PROCEEDINGS

SMS TPE'94

Moscow, Russia
September 19-23

Sponsored by
Office of Naval Research, USA
Russian Basic Research Foundation

Institute for System Programming of the Russian Ac. of Sci.
Computing Centre of the Russian Ac. of Sci.

DTIC QUALITY INSPECTED 3

19950104 072

Copyright © 1994 by Institute for System Programming of the Russian Academy of Sciences.

Second International Conference on

SOFTWARE FOR MULTIPROCESSORS AND SUPERCOMPUTERS
THEORY, PRACTICE, EXPERIENCE



DTIC
ELECTE
JAN 0 6 1994
S
D
F

Moscow

September 19-23 , 1994

PROCEEDINGS

Sponsored by

Office of Naval Research, USA

Russian Basic Research Foundation

This document has been approved for public release and sale; its distribution is unlimited.

Institute for System Programming of the Russian Ac. of Sci.

Computing Centre of the Russian Ac. of Sci.

Table of Contents

SMS TPE' 94 Program Commitee ____ 3

Preface ____ 4

Plenary Session : Invited Presentation

Parallel Simulated Annealing on Distributed Memory Systems ____ 10
F.H.Lee, G.S. Stiles, V.Swaminathan

Section I : Languages,Compilers and Programming Systems

TPML: Parallel Meta Language for Scientific and Engineering Computations Using Transputers ____ 22
J.Hartley, A.Bargiela

A rapid compiler prototyping system for fine-grained concurrence architectures ____ 32
V.Evstigneev,.V.Kasyanov

A Parallelizer for a Language without Variables ____ 39
R.Ortega, D.Bourge

SYNAPS/3 - An Extension of C for Scientific Computations ____ 48
V.A.Serebriakov, A.N.Bezdushny, C.G.Belov

The Meander Language and Programming Environment ____ 57
G. Wirtz

Data-Flow Language for MIMD Distributed Memory Multiprocessors ____ 67
J. Julliand, B.Markhoff.

Retargetable Compiler of ANSI C Superset for Vector and Superscalar Computers ____ 77
S.Katserov, A.Lastovetsky, S.Gaissaryan, D.Khaletsky, I.Ledovskih

Dataflow Assembly Language Programming System ____ 85
L.G.Tarasenko

Object-Oriented Language for Distributed Computations ____ 94
V.A.Torgashev, I.V.Tsaryov

A Data-Parallel Declarative Language for the Simulation of Large Dynamical Systems and its Compilation ____ 103
O.Michel, J.-L.Giavitto, J.-P.Sansonnet

Compiling a Producer-Consumer LEQ System in a Network of Communicating Processes ____ 112
M.-C. Eglin-Leclerc, J.Julliand

Fortran DVM - Language for Portable Parallel Program Development ____ 124
N.A.Konovalov, V.A.Krukov, S.N.Mihailov, A.A.Pogrebtsov

Section II : Neural Networks

NEUTRAM - A Transputer Based Neural Network Simulator _____ **136**
D.O.Gorodnichy, A.M.Reznik

A Transputer Based Robot Vision System to Evaluate Neuro-Control _____ **144**
D.Kuhn, J.P.Urban, S.Hagmann, H.Kihl

Multilayer Neural Networks on a Pipeline of Transputers. Application to Automatic All-Night Sleep Stages Quoting _____ **151**
A.Pinti, J.C.Grossetie

Section III: Applications

Real Applications on the TN300 T9000 Based Scalable Parallel Computer _____ **162**
D.Duval

Specification of Ditributed Hard Real-Time System _____ **169**
L.Carcagno, M. De Michiel, D.Dours, R. Facca, B.Sautet

A Flexible Approach to Parallel, Real Time Graphics _____ **179**
R.Cant

Transputer Network Implementation of Boolean Operations on Polyhedral Objects _____ **189**
Y.Chepaite

Section IV: Architectures

The Balanced Hypercube and Fault-Tolerant Ring Embeddings _____ **200**
J.Wu, K.Huang

On System Interconnection Strategies for a Parallel Cellular Machine _____ **210**
F.-F. Cai

On FPGAs as a New Hardware Support for Parallel Discrete Event Simulation __ **220**
C.Beaumont, J.Champeau, J.M.Filloque, B.Pottier, P.Boronat,

Control of CLOS Rearrangible Switching Networks on the Neimann Algorithm _ **230**
I.Sakho

Highly Parallel Asynchronous Computer System for the Large Data Arrays Processing _____ **238**
A.B.Barsky, V.V.Shilov

Section V : Enviroments, Debugging & Monitoring

XDSM - an X11 Based Virtual Distributed Shared Memory System _____ **250**
A.Argile, A.Bargiela

CCSP - A Formal System for Distributed Program Debugging _____ **260**
E.Arrowsmith, B.McMillin

A Friendly Data Flow Programming Environment _____ **270**
M.Menasche, D.M.Maia

A Message Density Monitoring Strategy for Distributed Memory Parallel Systems **280**
L.P.Santos, A.Chalmers, A.Proenca

Graphical Construction of Parallel Programs _____ **289**
G.R.Ribeiro Justo

Ada for Multicomputers ____ 299
B.Herrmann, G.-R. Perrin

Monitoring and Debugging in RAMPA ____ 309
V.Alexakhin, D.Arapov, V.Krukov, V.Ivanov, A.Petrenko, A.Vashakidze

Monitoring Parallel Programs for Detecting Access Anomalies Occured First ____ 317
Y.-K. Jun, K. Koh

An Appoach to the Scalable Software Development with QUICKPLAY System _ 327
A.Biriukov, V.Meliokhin

A simulation Environment for Finite-State Machine Model on Parallel Transputer-based Architectures: a Plant Automation Case Stady ____ 337
V.Filippi, G.L.Redaelli

Pipe_Lib : A Parallel Library for Writing Pipeline Applications ____ 346
L.M.Silva, J.G.Silva

PPS-Construction: a Methodology for Computer Aided Construction of Parallel Program Systems ____ 356
V.D.Ilyin

An Integrated Hardware-Software Development Environment for Transputer-Based Systems ____ 365
J.Spletukhov

Section VI: Parallelisation Techniques, Sheduling & Load Balancing

Dynamic Load Balancing in a Parallel Processing System ____ 370
N.Baykal, K.Erciyes

Mapping and Scheduling Data-Parallel Dataflow Programs on Massively Parallel Architectures ____ 380
A.Mahiout, J.-L.Giavitto, J.-P.Sansonnet

The Enhancement of a User-level Thread Package Scheduling on Multiprocessors 390
M.Gil, X.Martorell, N.Navarro

Use of V-ray System for Optimisation of a Serial Program for Cray Supercomputers ____ 399
V.V.Voevodin

Section VII: Numerical Algorithms

Analysis of Splitting Parallel Methods for Solving Block Tridiagonal Linear Systems ____ 406
E.Galligani, V.Ruggiero

Investigation of Parallel Algorithms for Solving the Boltzmann Equation and their Efficiency ____ 417
V.Aristov, I.Mamedova

A Multi-phase Gossip procedure : Application to Matrices Factorization ____ 426
A.Benaini, D.Laiymani

A Substructuring Method for Solving the Image Restoration Problem on Multiprocessor Systems _____ 435
I.Galligani, E.Loli Piccolomini, V.Ruggiero, F.Zama

Selected SMS TPE'93 Papers

Specification and Compilation of Distributed Algorithms for the ArMen Machine_____ 446
P.Dhaussy, S. Rubini

The Programming Language Modula-P_____ 463
J. Vollmer

Pact C - A Fault Tolerant Parallel Programming Environment_____ 475
J. Maier

Parallel Processing for Real-Time Image Analysis of Aircraft Engines_____ 489
N.W.Campbell, A.G.Chalmers, B.T.Thomas

A Parallel Processing Environment for Solving Problems with Global Communication Requirements _____ 498
A.G.Chalmers, D.J. Paddon

The Evolution of the Meiko Computing Surface _____ 511
D.Wilson

Nonprocedural Language NORMA and Problems of Its Implementation _ 523
A.N. Andrianov, K.N.Efimkin, I.B. Zadykhaylo

Author index_____. 531

SMS TPE'94 Program Committee

Chairman:

Prof. Victor Ivannikov, Corresponding Member of the Russian Ac. of Sci., Institute for System Programming, Moscow, Russia

Co-Chairmen:

Prof. Vladimir Serebriakov, Computing Centre of the Russian Ac. of Sci., Moscow, Russia

Prof. Constantine Polychronopoulos, University of Illinois, USA

Conference Secretary:

Dr. Alexander Biriukov, Computing Centre of the Russian Ac. of Sci., Moscow, Russia

Members:

Dr. Neill Campbell, University of Bristol, Bristol, UK

Dr. Alan Chalmers, University of Bristol, Bristol, UK

Prof. Yurii Evtushenko, Computing Centre of the Russian Ac. of Sci., Moscow, Russia

Prof. Stefan Jaehnichen, GMD-FIRST, Berlin, Germany

Prof. Victor Kasyanov, Institute of Informatics Sysytems, Novosibirsk, Russia

Dr. Bernard Pottier, Universite de Bretagne-Occidentale, France Prof. Dyke Stiles, Utah State University, USA

Mr. Dave Wilson, MEIKO Ltd., Bristol, UK

Prof. Larry Wittie, Computer Science, State University of New York at Stony Brook, USA

Accesion For
NTIS CRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

PREFACE

The problem of developing efficient parallel software is by the time one of the great challenges in modern computer science. As the number of commercially available parallel platforms increases, the interest to parallel programming is growing throughout the world. The activity in this area in Russia, that started in the early 60's and was that time mostly academic, has led to a number of theoretical results and experimental software and hardware systems implementations. Some of them have later been used to develop real industrial applications.

The primary aim of SMS TPE'93 and its successor SMS TPE'94 was to provide a forum for free exchange of ideas between Russian researchers and their colleagues abroad. The first event in the series was, in fact, a small workshop with only a limited number of participants. However, it allowed the forming of many more scientific contacts and demonstrated significant interest from the both sides. The SMS TPE'94 now was attracted many more participants and from more than 10 countries.

Unlike most conferences on the field, the SMS TPE'94 program was announced to cover fairly large spectrum of topics. Since only a few international conferences on parallel software design are held in Russia, we decided not to restrict the program, but to keep it potentially open to all work in the area.

The papers submitted to SMS TPE'94 hence cover a wide range of topics from parallel languages and compilers to new neural networks. We also have included into this book some selected papers from SMS TPE'93.

Many people have contributed to SMS TPE'94 organization and to preparation of this book. First, we would like to give our thanks to the Program Committee members for their refereeing efforts. Special thanks go to Grigory Amirdjanov, Scientific Council on Cybernetics, and Victor Meliokhin, The Computing Centre of the RAS, for their work on collecting the manuscripts and preparing the book for printing.

We are especially grateful to the Office of Naval Research (USA) that has kindly sponsored the event and significantly helped at all stages of its organization.

V.P.Ivannikov
V.A.Serebriakov

Plenary Session : Invited Talk

Parallel Annealing on Distributed Memory Systems

F. H. Lee, G. S. Stiles, and V. Swaminathan
Department of Electrical and Computer Engineering
Utah State University
Logan Utah 84322-4120
fhl@multi.ee.usu.edu

Summary

This paper summarizes efforts to develop parallel simulated annealing algorithms, and describes our work on an algorithm designed to efficiently handle a wide variety of problems on large (massively parallel) distributed memory systems. This algorithm is based on the use of multiple independent searches, and uses the results of many short exploratory runs to determine the best parameters for the main annealing phase.

Keywords: simulated annealing, parallel, randomized algorithms, distributed memory

1. Introduction

Simulated annealing is an approximation technique developed by Kirkpatrick [1,2] from work by Metropolis [3], which has proved useful for attacking complex combinatorial problems. The algorithm typically begins with a randomly chosen configuration. This configuration is then perturbed (a *move* is made), and the new configuration is accepted if the move results in a decrease in the cost or object function; if an increase results, the move is accepted with a probability decreasing exponentially with an increasing change in cost; the rate of decrease depends upon a parameter referred to as the *temperature*. As the algorithm progresses, the temperature decreases, thus decreasing the probability that a change with a given increase in cost will be accepted. Practical implementations often decrease the temperature in steps; a series of moves at one temperature may be referred to as a *chain*, in recognition of its similarity to a Markov chain. Under suitable conditions, and an appropriate schedule for decreasing the temperature (the *cooling schedule*), the algorithm converges to the global minimum [4-7].

The algorithm has been applied to practical tasks such as cell placement and routing in VLSI design [6,8-23], multi-processor task allocation [24-26], emission tomography [27], electrostatics [28], vector quantization [29], and routing [30] and to classic problems including the travelling salesman problem (*TSP*) [4,31-35], graph coloring [5,36], and linear placement [37].

Our work has concentrated on the design of distributed database networks [38,39]. The initial specifications of the problem include the geographical locations of the nodes, the databases, the query and update traffic from each node to each database, and the costs of communication links and other hardware. The goal is to allocate databases to nodes, connect nodes with links, and route queries and updates at minimum operating cost while maintaining a specified level of performance. This problem rapidly becomes very complex: a 20 node network with two databases has about 10^{1400} configurations. The problem is particularly difficult in that the cost calculations are global, and small changes in one section of the network may have significant effects on remote sections.

While annealing often out-performs other heuristics, it may still require long computation times; there have been many attempts to develop effective parallel versions. Since it now appears as if the most powerful parallel systems will have large numbers of processors with physically distributed memory, we have concentrated on developing a general approach which will run efficiently on such machines.

There are two basic approaches to developing parallel algorithms. When there are tasks which may be executed concurrently while preserving the correctness of the program, *task* (or *algorithmic*) parallelism may be used. The evaluation of complex cost functions or constraints are

tasks which might be carried out with algorithmic parallelism.

When the same operations are carried out independently on different members of a large data set, the *data parallel* (or *domain decomposition*) approach may be used: each processor applies the same algorithm to a different chunks of data. This method may be referred to as *SPMD* (single program multiple data) [40]. Classic examples include such problems as fluid dynamics, which is highly amenable to domain decomposition. In the annealing of image processing, where local cost calculations must be made over the entire image, the data parallel approach can be very successful. Data parallelism perhaps more often appears in annealing in the form of *parallel moves*, where multiple processors simultaneously apply moves to the same configuration. We see below there are a number of variations on this theme.

With randomized approximation algorithms, such as simulated annealing, in which both the execution time and the final result may be randomly distributed (and the average run time can often be adjusted), another version of data parallelism (*randomized parallelism*) is possible: we simply run multiple copies of the randomized algorithm simultaneously and select the best result, hoping that the best result of many short runs will be better than the result of one long run. The success of this approach depends upon the joint distribution of the run time and final result [41].

2. Previous Work

The papers reviewed below will be grouped according to whether data parallelism or algorithmic parallelism is the primary method (more than one method may be used in a single application). Please note that this represents only a very small sample of a large body of literature on parallel annealing; Greening [12] can be consulted for a more extensive survey.

A. Data Parallelism

Data parallel approaches to annealing may be separated into two main groups. The first contains those which use domain decomposition to calculate complex cost or constraint functions; the second contains those which employ move and/or randomized parallelism. Two or more types of data parallelism may, of course, be used on a single problem.

Azencott [7] discusses a class of problems which are ideally suited to domain decomposition. In image processing tasks such as restoration, edge detection, and segmentation, the cost calculations are essentially local. With large images, efficient use can be made of even massively parallel systems.

A divide-and-conquer form of data parallelism was used by Natarajan and Kirkpatrick [17] to solve the VLSI cell placement problem on a shared memory system. The results showed that the speedup tends to level off with more than 4 processors. The projected speedups, estimated from the authors' figures, were 1.6, 2.5, and 2.9 on 2, 4, and 7 processors, respectively.

We have investigated the use of divide-and-conquer on our network design problem [42,43]; however, the strongly global nature of the problem limits the success of the approach. The nodes of the network are divided into disjoint groups, which are annealed separately; the groups are then joined and the resulting network is annealed. The fact that only a small number of hops is allowed between communicating nodes requires that a large number of links (often lengthy and thus costly) will always need to be added to the combined network. Minimization of the costs of these links requires that the last annealing phase start at a fairly high temperature - which undoes the efforts of the first phase. The resulting algorithm actually runs more slowly than the sequential version.

In the parallel move approach, multiple processors make one or more moves on a single configuration, and the moves are then combined to produce one or more new configurations. The success of a method which requires the maintenance of one (or a few) configurations depends strongly on the nature of the cost and constraint calculations, and the degree to which moves interact. If the updates to the cost calculations require only local information, then each processor can accurately evaluate the result of its move. If one move rarely conflicts with another move, then the results of several moves can be easily combined. If either of these conditions does not hold,

substantial computational or communication overhead may be incurred or moves may be wasted.

We classify various parallel move schemes by generalizing Azencott's [7] definition of periodically interacting multiple searches (*PIMS*). We assign a 4-tuple (n, r, a, s) to each scheme, where n represents the period in iterations between interactions, r represents the number of simultaneously running configurations, a represents the maximum number of moves accepted at each interaction, and s represents the number of unique configurations at the beginning of each period. Note that r and s may be no greater than the number of simultaneous searches - which is typically equal to the number of available processors; a must be no greater than nr, the total number of moves attempted per period. We use $a = 0$ to indicate schemes which do not combined moves but select one (or more) of the multiple configurations for the next round of searches.

In the simplest version of parallel moves, in which all processors work on one configuration, and the configuration is updated after one move by each processor, we have a (1, p, p, 1) PIMS implementation, where p is the number of processors. The performance of this approach will be limited by the fact that a single consistent configuration of the problem must be maintained. With shared-memory systems, a single copy of the configuration may be updated by all processors, but memory contention will ultimately limit scalability. On distributed-memory systems, the communication required to maintain a single configuration can rapidly become prohibitive. On both shared- and distributed-memory machines, the number of useful simultaneous moves may be limited by conflicts among moves which result in invalid configurations. The (1, p, p, 1) approach can be effective in the waning phase of annealing (*tail parallelism*), when very few moves are accepted; with few accepted moves, little communication is required. Unfortunately, this phase typically does not contribute significantly to the overall result.

Felten et al. [31] implemented parallel moves on a MIMD machine to solve the TSP; the method is (1, p, p, 1). Since moves on different portions of the salesman's journey do not interact, cost or object function calculations are local and data parallelism in the form of parallel moves can be beneficial. Efficiencies of 86%, or more, were reported on 64, or fewer, processors.

Rutenbar and Kravitz [21] combined task parallelism and data parallelism, switching from one to the other as annealing progressed. The task parallelism evaluated the results of a single move. When the move acceptance rate dropped to a low value, the algorithm switched to data parallelism in the form of parallel moves - a (1, p, p, 1) version. The implementation on a 4-node shared memory VAX 11/784 achieved speedups of about 1.8. The low efficiency resulted from limited parallelism in task decomposition and the low impact of tail parallelism.

Parallel move annealing of the placement problem was studied by Darema-Rogers et al. [9,10]. They decrease the amount of shared memory traffic by not updating the common configuration after every move, thus allowing errors in the object function calculation to accumulate on each processor. The common configuration and object function are updated periodically. A 32 node IBM 3081 multiprocessor yielded speedups up to 14.1, and the IBM RP3 research machine attained a speedup of about 6.5 with 8 processors.

Greening and Darema [11] developed "random rectangles spatial decomposition" to implement parallel moves on cell placement problems. This method partitions the cell map into disjoint rectangular regions; each processor makes moves only within a distinct region. This method, as well as other spatial decomposition methods [8,14,44], results in no or very light interaction among processors if the problems are such that moves in one domain have little effect on moves in other domains. Implementation was on an IBM RP3 machine. No speedup information was provided, but statistics indicated that convergence improves as processors are added.

Jeong and Kim [32,33] utilized parallel move and divide and conquer techniques to implement parallel annealing on a MIMD machine consisting of T414 transputers to solve the TSP.

The cities are grouped and parallel moves are performed on disjoint groups. Speedups (figured on a per-move computation time) relative to a 2 processor implementation on a 2048 city problem were 1.63, 2.85, and 5.7 on 4, 8, and 16 processors. For a 1024 city problem, speedups of about 1.7, 2.5, 3.8, and 4.3 were achieved by 2, 4, 8, and 16 processors, respectively.

Girodias et al. [27] implemented a parallel annealing algorithm on 54 T800 transputers. The target problem was emission tomography image reconstruction; tomography differs significantly from such problems as the TSP in that the cost calculations are essentially global. The authors were able to develop an object function which was more suitable for data domain decomposition, however. The performance was quite encouraging, yielding speedups of about 15, 27, and 37 on 18, 36, and 54 processors respectively.

An interesting version of parallel moves is proposed by Roussel-Ragot and Dreyfus [20]. It differs from other methods in that only one accepted move is used to update the configuration (and object function), thus maintaining the properties of sequential annealing; hence this is a (1, p, 1, 1) approach to PIMS. The acceptance rate of moves clearly plays a very important role. Within the high temperature region, the acceptance rate is high, and there are many accepted moves per set of parallel moves. Thus, many of these accepted moves are essentially wasted, and the benefit of parallel moves is not great. However, in the low temperature region, the story reverses as tail parallelism becomes effective. The speedup of this method is then dependent of the duration of the low temperature phase. For their placement problem running on a transputer network, the speedups (estimated from the figures) are about 2.2 and 3.0 with 3 and 6 T800 transputers.

Systolic parallel annealing techniques were introduced by Aarts et al. [45]. Each constant-temperature chain is executed on a different processor. Each chain is divided into a number of subchains equal to the number of processors. Neighboring processors communicate at the end of subchains to determine which configuration will be used to continue. Note that this is a (n, p, 0, s) PIMS system, where n is the subchain length, the 0 indicates that moves are not combined but rather complete configurations are chosen, and s can vary from 1 to p depending on the implementation. The advantages of this approach are 1) loads among processors are nearly balanced because of the equal subchain lengths; 2) the communication overhead is relatively small, since communication occurs at the end of each subchain. Performance initially improves as the number of processors is increased, but eventually levels off as the subchains become very short. With short subchains, equilibrium is more difficult to attain and the temperature decrement must decrease, resulting ultimately in decreasing efficiency. Furthermore, as the subchains shrink, communication overhead becomes more significant.

Aarts et al. [45] also proposed a "cluster" version of the above algorithm. In this case multiple processors work on the same subchain. At the end of the subchain, one result is probabilistically selected to be the starting configuration for the next subchain. As the probability of acceptance decreases, more processors can efficiently be assigned to a subchain (essentially PIMS within PIMS). This approach can be efficient for small numbers of processors, but increasing communication is expected to limit efficiency as the number of processors grows.

Kim and Kim [34] proposed a systolic technique called "step-wise-overlapped parallel annealing". This scheme also breaks the chain into subchains, and assigns a subchain to each processor until all processors are busy; whenever a processor finishes its subchain, it grabs another subchain to work on. Each processor maintains its own cooling schedule. An implementation was simulated on a CRAY-2S. With 128 simulated processors, a speedup of 70.8 was obtained.

A parallel annealing algorithm combining the systolic approach and data parallelism on an electrostatics problem was proposed by ter Laak et al. [28]. They adopted the step-wise-overlapped version of Kim and Kim [34] and incorporated data decomposition for the cost calculation. Efficiencies of greater than 86% were reported on a 57 transputer system.

We have tried a variety of (n, r, nr, s) PIMS approaches on the network problem with

several combinations of r and s [46-48]. None of the methods was very promising. The major performance limitations result from the facts that simultaneous moves have a fairly high probability of conflicting, thus negating the value of parallel moves on one configuration, and that communication of the entire configuration is very expensive.

If we take the PIMS method to the limit, we have multiple independent instances of complete annealing runs (MIR), and we select the best result at completion; this can be viewed as $(N, p, 0, p)$, where N is the total length of the runs and the 0 again indicates that we do not combine moves but pick the best final configuration. When Azencott [7] analyzed PIMS under the assumption that, for a single sequential annealing run, the probability of being in a non-optimal state falls off as a power of the number of iterations, he discovered that the best approach is to interact only at the conclusion of complete runs - which is exactly the $(N, p, 0, p)$, or MIR, version. Furthermore, for a given amount of computation time and a given problem, he has shown that it may be more efficient to have multiple runs on each processor as well. We will refer hereafter to the $(N, p, 0, p)$ PIMS version as MIR, and will reserve PIMS to refer to those implementations which interact more frequently.

Dodd [49] reached a similar conclusion when investigating the effective difference between one long run and MIR. For problems in which the probability of finding the optimal solution is proportional to the log of the number of iterations, he showed that the total number of iterations is the only parameter affecting the performance. If n iterations are to be performed by sequential annealing to attain a certain quality result, then p copies of runs of length n/p should yield the same result. Thus, since there is nearly no communication overhead, a speedup of nearly p can be achieved with p processors.

The "ensemble" method of [50] is another implementation of MIR. The principle is to explore the energy landscape quickly with multiple processors, and to determine the equilibrium adaptively. Ensemble annealing explores the energy landscape by generating random walkers in parallel. The "best so far energies" (BSFE) are recorded by each instance of the algorithm. The equilibrium conditions and the annealing parameters for the next temperature step are determined from the global distribution of BSFE. The determination of the optimal ensemble size (number of parallel random walkers) has been studied by Hoffmann et al. [51].

Natarajan and Kirkpatrick [18] proposed a combination of parallel moves and MIR to solve the placement problem. Groups of m processors work together to perform parallel move simulated annealing. The best outcome of k groups (mk processors total) is selected. Unlike data decomposition, all processors are allowed to work on the entire data domain. Their implementation was on a shared memory system; 2, 4, 6, and 7 processors yielded speedups (estimated from figures) of 2.0, 3.7, 5.5, and 5.5, respectively.

We have tested the MIR approach extensively on our network problem. For a given amount of computation time, this method produces results at least as good as any other method we have tried. It has the obvious additional advantages that there is very little communication overhead and it is very easy to program. This method is the basis of our new work described in Section 3.

B. Algorithmic Parallelism

Algorithmic parallelism may be used when a substantial portion of the annealing routine is spent in tasks which can be efficiently spread across processors. After a move is made, for example, constraints may need to be checked and the change in cost must be calculated. For some problems these tasks may require a substantial amount of computation; if they can be parallelized, efficient use may be made of algorithmic parallelism. In the work of Rutenbar and Kravitz [21] discussed above, the cost calculation was executed in parallel. The cost and constraint calculations on our network problem could be done in parallel if they required more work.

Witte et al. [26] used an approach called *speculative parallelism*, in which the operations of

annealing were pipelined, with the moves, cost calculation, and housekeeping overlapping. The term *speculative* refers to the fact that later decisions can reject the results of prior calculations. Speedups of 2.6 were obtained on an 8-node NCUBE system.

C. Summary

The results discussed above indicate that the task parallel and data parallel approaches can provide decent speedup in some cases. Successful task parallelism, however, requires that sufficient parallelism be available in the problem; furthermore, problem-specific coding will be required to take advantage of task parallelism. The parallel move approach does not require problem-specific code, but does suffer from the requirement that a global, consistent configuration be maintained; this can be computationally expensive on both shared- and distributed-memory machines. Global effects can cause multiple moves to yield inconsistent configurations, resulting in either many rejected moves or expensive repairs. Programming of both data and task parallel annealing algorithms can be fairly complex, since significant communication among processors may be required. The MIR form of randomized parallelism is, on the other hand, extremely easy to program since there is a minimum of communication required, and the same shell can be used for all problems. With appropriate time-cost distributions (which may be adjustable by varying the annealing parameters), substantial speedup may be obtained with MIR.

3. Recent Efforts at Utah State

We have recently concentrated on various versions of PIMS and MIR, and have extended our test problems to include multiprocessor task allocation [25], min cut, and the electrostatic problem discussed by ter Laak et al. [28]. We have been particularly interested in determining whether Azencott's result (that the most efficient PIMS approach is MIR) holds for a variety of real problems. If this is indeed the case, then this approach may be the method of choice for a general parallel annealing algorithm.

Thus far we have applied PIMS to a wide range of network design, task allocation, and min cut problems, using 1 to 40 transputers on a Meiko CS-1.5; code has been developed with Edinburgh's CHIMP message-passing package. We have also collected statistics from multiple independent runs with a broad range of computation times. From these statistics we can calculate, following the methods described by Stiles [41], the expected minimum cost and running time obtained by selecting the best of multiple independent runs.

Figure 1 presents the average final cost versus the number of iterations of Kim and Kim's [34] version of step-wise systolic PIMS on a 7-node network problem for 2, 4, 8, 16, and 32 processors; the results of the sequential algorithm are also shown. Each sequential point is the average of 50 runs; the PIMS data are averages of 20 runs. The run lengths were varied by changing the chain length, and a geometric cooling schedule was used. The unconnected points are the expected best cost from MIR runs on the specified number of processors; these values were calculated from the distributions obtained from the sequential runs.

The plot demonstrates that substantial improvement can be obtained over the sequential algorithm by using PIMS. As the number of processors is increased, the performance initially improves. The performance increase gained by adding processors eventually falls off, which is to be expected, since the subchain length on each processor becomes very short and there is less time to approach equilibrium. The most important result is that, for a given number of processors and a given number of iterations (on this problem), the PIMS approach is no better than running MIR and selecting the best result - as Azencott predicts. Furthermore, when communication overhead (which does not appear when the results are plotted vs. iterations) is added, the curves for the PIMS method, which must communicate at each interaction, will shift to the right; the curves for MIR will not move significantly since that algorithm communicates only at its conclusion.

We note finally that the expected best result of 128 independent runs is not significantly better than that for 32 processors, suggesting that the underlying distribution will limit the maximum

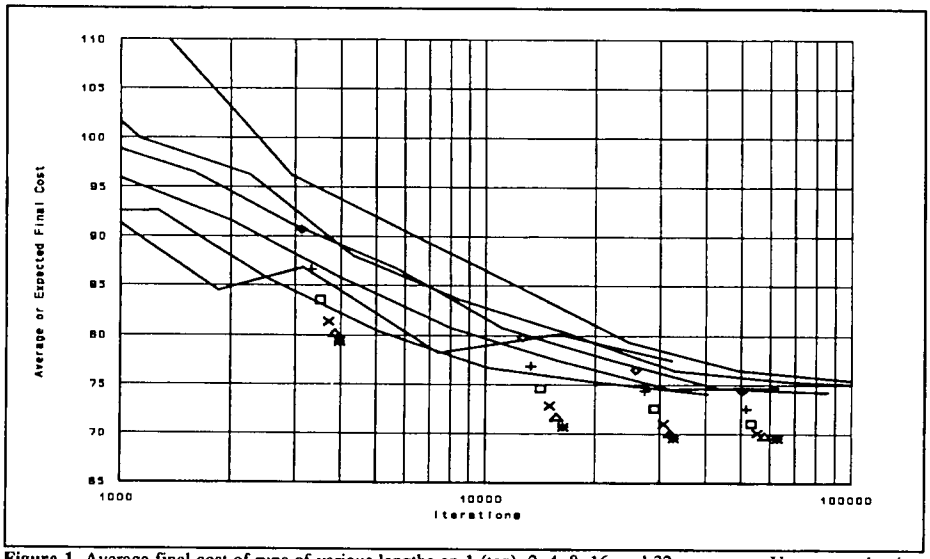


Figure 1. Average final cost of runs of various lengths on 1 (top), 2, 4, 8, 16, and 32 processors. Unconnected points are expected final costs of MIR implementations on 2, 4, 8, 16, 32 and 128 processors.

speedup obtainable. On larger problems we find that more processors may be used efficiently.

We have become increasingly convinced that MIR may be the most efficient and most general method of parallel annealing, as well as the easiest. We are now developing a parallel implementation based on MIR which we expect will be applicable, and adaptive, to a wide variety of problems. An important feature is the use of statistics accumulated from many runs to determine the optimal number of runs on each processor and the optimal cooling schedule for the runs.

The algorithm consists of three phases. In Phase 1, each processor starts with a randomly selected configuration and makes a small number of moves, accepting all changes regardless of the cost. Then the maximum (ΔC_{max}) and the minimum (ΔC_{min}) absolute cost changes are used to determine the starting (T_s) and final (T_f) temperatures, respectively:

$$T_s = -\frac{\Delta C_{max}}{\ln(0.5)} \; ; \quad T_f = -\frac{\Delta C_{min}}{\ln(0.5)} \qquad (1)$$

The second phase is used to determine 1) the length of a single run which, on the average, will yield a suitable solution; 2) the total number of shorter runs which will yield the same result; and 3) the initial cooling schedule for the third phase. We begin Phase 2 with several rapid annealing runs of different lengths on each available processor, using T_s and T_f together with a simple geometric cooling schedule ($T_{i+1} = \lambda T_i$, $\lambda < 1$). (Note that T_s and T_f will be updated as the algorithm progresses.)

The length of a single acceptable run is then determined with the use of Azencott's assumption [7] that the annealing algorithm converges according to equation (2), where X_n is the configuration after n iterations, E_{min} is the set of minimum cost configurations, and $\alpha > 0$, $K > 0$ are constants estimated from the Phase 2 runs.

$$P[X_n \notin E_{min}] \propto \left(\frac{K}{n}\right)^{\alpha} \tag{2}$$

From this relation we can derive an expression for the difference δ_n between the average and the true minimum cost after n iterations:

$$\delta_n = \langle C_n \rangle - C_{min} \propto \left(\frac{K}{n}\right)^{\alpha} \tag{3}$$

We elect to terminate the algorithm at $n = N$ when this difference is sufficiently small:

$$\delta_n \leq \epsilon \tag{4}$$

where ϵ is a small positive number. This effectively gives us the number of iterations N of a single sequential run which will, on the average, provide us with a result of sufficient quality.

Azencott [7] has further shown that, under the assumption on convergence stated above, the optimal use of the N iterations is to distribute them across s_N shorter runs, where

$$s_N = \left\lfloor \frac{N}{\alpha K} \right\rfloor \tag{5}$$

The s_N runs are then allocated to the available processors.

Finally we determine an appropriate cooling schedule. A number of researchers [see, e.g., the summary in 16] have suggested that the temperature should be decreased so that the equilibrium cost decreases by no more than a standard deviation or so at each step; this should permit a more rapid approach to equilibrium at each temperature. In practice, when dealing with a small number of runs, the scatter in the cost is often such that it is difficult to determine the average and standard deviation at each temperature. However, by utilizing many runs on many processors we can estimate the cooling curve more accurately.

Cooling histories from the Phase 2 runs are accumulated and an average cooling curve $(T,\langle C \rangle,\sigma_C)$ is computed. From this curve we can construct a cooling schedule specifying a sequence of temperatures such that the expected change in cost from one temperature to the next is equal to $\alpha\sigma_C$, where σ_C is the standard deviation of the cost and α is near 1.0.

The actual annealing takes place in Phase 3; the specified number of runs are made on each processor according to the cooling schedule obtained in Phase 2, and the best result is selected at the end. Note that the cooling schedule can be continually updated, if desired, based on the cooling histories of each intermediate run; this update could be done locally on each processor or globally at the cost of collecting the histories and disseminating the updated schedule.

Preliminary runs of this method appear very promising. The scheme of estimating N, the length of a single run which should produce acceptable results, yields values close to what we would expect based on previous experience. The results of Phase 3 produce very good final costs. For a five node network, the overhead of Phases 1 and 2 is about 50% of the total execution time; for a 10 node network, the overhead shrinks to about 25% of the total. For problems of realistic size the overhead should be significantly less.

4. Conclusions

Our work with the annealing of complex problems has led us to conclude that, in such cases, the most efficient parallel implementation consists in multiple independent runs, from which one selects the best result. This approach avoids problems that may plague other methods, including

the need for problem-specific code, heavy communication overhead, and the complications induced by global cost functions and conflicting parallel moves. A key feature of this approach is the use of information gained during the run, both to select the parameters of the parallel implementation and to tune the cooling schedule; a large parallel system, and many runs, allow reliable information to be obtained quickly.

Acknowledgements

We thank the National Center for the Design of Molecular Function at Utah State for use of computing facilities; these facilities are supported by the National Institutes of Health. The second author would like to thank Arne ter Laak, and the Edinburgh Parallel Computing Center, for their support during his sabbatical.

References

[1] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, 220, pp. 671-680, 1983.

[2] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing: Quantitative Study," *J. Statistical Physics*, 34, pp. 975-986, 1984.

[3] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computer Machines," *J. Chemical Physics*, 21, 6, pp. 1087-1092, June, 1953.

[4] E. H. L. Aarts and P. J. M. van Laarhoven, "Statistical Cooling: A General Approach to Combinatorial Optimization Problems," *Philips J. Research*, 40, pp. 193-226, 1985

[5] E. H. L. Aarts and P. J. M. van Laarhoven, "A New Polynomial Time Cooling Schedule," *Proc. IEEE Int'l. Conf. Computer Aided Design*, pp. 206-208, Nov., 1985.

[6] E. H. L. Aarts and J. H. M. Korst, Simulated Annealing and Boltzmann Machines, Wiley, New York, 1989.

[7] R. Azencott, ed., Simulated Annealing: Parallelization Techniques, John Wiley & Sons, Inc. 1992.

[8] A. Casotto, F. Romeo, and A. L. Sangiovanni-Vincentelli, "Placement of Standard Cells using Simulated Annealing Algorithm on the Connection Machine," *Proc. IEEE Int'l. Conf. Computer-Aided Design*, pp. 350-353, 1986.

[9] F. Darema-Rogers, S. Kirkpatrick, and V. A. Norton, "Simulated Annealing on Shared Memory Parallel Systems," Research Report RC 12195, IBM T. J. Watson Research Center, Oct. 1986.

[10] F. Darema-Rogers and G. F. Pfister, "Multipurpose Parallelism in VLSI Computer-Aided Design: A Case Study," Research Report RC 12516, IBM T. J. Watson Research Center, Feb. 1987.

[11] D. Greening and F. Darema, "Rectangular Spatial Decomposition Methods for Parallel Simulated Annealing," *Proc. Int'l. Conf. Supercomputing, ACM*, pp. 295-302, Crete, 1989.

[12] D. R. Greening, "Parallel Simulated Annealing Techniques", in Emergent Computation, ed. S. Forrest, MIT/North Holland, pp. 293-306, 1991.

[13] M. D. Huang, F. Romeo, and A. L. Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated Annealing," *Proc. IEEE Int'l. Conf. Computer-Aided Design*, pp. 381-384, Nov., 1986.

[14] R. Kling and P. Banerjee, "Concurrent esp: a Placement Algorithm for Execution on Distributed Processors," *Proc. International Conf. Computer Design*, pp. 354-357, Oct. 1987.

[15] S. A. Kravitz and R. A. Rutenbar, "Multiprocessor-Based Placement by Simulated Annealing", *Proc. Design Automation Conf.*, IEEE & ACM, June, 1986.

[16] P. J. M. van Laarhoven and E. H. L. Aarts, Simulated Annealing: Theory and Applications, D. Reidel Publishing Company, Dordrecht, Holland, 1987.

[17] K. S. Natarajan and S. Kirkpatrick, "Evaluation of Parallel Placement by Simulated Annealing: Part I - The Decomposition Approach," Research report RC 15246, IBM T. J. Watson Research Center, Oct. 1989.

[18] K. S. Natarajan and S. Kirkpatrick, "Evaluation of Parallel Placement by Simulated Annealing: Part II - The FLAT Approach," Research report RC 15247, IBM T. J. Watson Research Center, Dec. 1989.

[19] F. Romeo and A.L. Sangiovanni-Vencentelli, "Probabilistic Hill Climbing Algorithms: Properties and Applications," *Proc. Chapel Hill Conf. VLSI*, pp. 393-417, Chapel Hill, May, 1985.

[20] P. Roussel-Ragot and G. Dreyfus, "A Problem-Independent Parallel Implementation of Simulated Annealing: Models and Experiments," *IEEE Trans. Computer Aided Design*, 9, 7, pp. 197-214, 1990.

[21] R. A. Rutenbar and S. A. Kravitz, "Layout by Annealing in a Parallel Environment," *Proc. IEEE Int'l. Conf. Computer Design*, pp. 434-437, Port Chester, Oct. 1986.

[22] S. R. White, "Concepts of Scale in Simulated Annealing," *Proc. IEEE Int'l. Conf. Computer Design*, pp. 646-651, Port Chester, Nov., 1984.

[23] C. R. Wouters, E. H. L. Aarts, and C. H. van Berkel, "Optimization of Gate Matrix Layouts Using Simulated

Annealing," *Philips Research Report*, 1986.
[24] J. L. Gaudiot, J. I. Pi, and M. L. Campbell, "Program Graph Allocation in Distributed Multicomputers," *Parallel Computing*, 7, pp. 227-247, 1988.
[25] N. Udiavar and G. S. Stiles, "A Simple but Flexible Model for Determining Optimal Task Allocation and Configuration on a Network of Transputers," *Proc. 1st Conf. NATUG*, IOS Press, Amsterdam, pp. 24-32, 1989.
[26] E. E. Witte, R. D. Chamberlain, and M. A. Franklin, "Parallel Simulated Annealing Using Speculative Computation," *IEEE Trans. Parallel & Distributed Sys.*, 2, 4, pp. 483-494, Oct. 1991.
[27] K. A. Girodias, H. H. Barrett, and R. L. Shoemaker, "Parallel Simulated Annealing for Emission Tomography," *Phys. Med. Biol.*, 36, 7, pp 921-938, 1991.
[28] A. ter Laak, L. O. Hertzberger, and P. M. A. Sloot, "NonConvex Continuous Optimization Experiments on a Transputer System," *Proc. 15th WOTUG Technical Meeting*, IOS Press, Amsterdam, pp. 251-265, April, 1992.
[29] J. Vaisey and A. Gersho, "Simulated Annealing and Codebook Design," IEEE CH2561-9, pp. 1176-1179, 1988.
[30] T. Yoshimura and E. S. Kuh, "Efficient Algorithms for Channel Routing," *IEEE Trans. Computer-Aided Design*, 1, pp. 25-35, 1982.
[31] E. Felten, S. Karlin, and S. W. Otto, "The Traveling Salesman Problem on a Hypercubic MIMD Computer," *Proc. Int'l. Conf. Parallel Processing*, pp. 6-10, St. Charles, Aug. 1985.
[32] C. Jeong and M. Kim, "Fast Parallel Simulated Annealing for Traveling Salesman Problem on SIMD Machines with Linear Interconnections," *Parallel Computing*, 17, pp. 221-228, 1991.
[33] C. Jeong and M. Kim, "Parallel Algorithm for Traveling Salesman Problem on SIMD Machines Using Simulated Annealing," *Proc. Int'l. Conf. Application Specific Array Processors*, IEEE, pp. 712-721, 1991.
[34] Y. Kim and M. Kim, "A Step-Wise-Overlapped Parallel Annealing Algorithm on a Message-Passing Multiprocessor System," *Concurrency: Practice & Experience*, pp.123-148, 2, 2, June 1990.
[35] S. Rees and R. C. Ball, "Criteria for an Optimum Simulated Annealing Schedule for Problems of the Travelling Salesman Type," *J. Physics, A*, 20, pp. 1239-1249, 1987.
[36] E. H. L. Aarts and J. H. M. Korst, "Boltzmann Machines as a Model for Parallel Annealing," *Algorithmica*, pp. 437-465, 6, 1991.
[37] E. H. L. Aarts, P. J. M. van Laarhoven, R. Burgess, and B. Culleton, "Linear Arrangement Using Statistical Cooling," *Philips Research Report*, 1984.
[38] P. Julien-Laferriere, F. H. Lee, G. S. Stiles, A. Raghuram, T. W. Morgan, "Stochastic Optimization of Distributed Database Networks," *Proc. IEEE Phoenix Conf. Comp. & Commun.*, pp. 452-460, Mar., 1989.
[39] A. Raghuram, T. W. Morgan, P. Julien-Laferriere, and G. S. Stiles, "A Model for Determining the Optimal Topology and Database allocation in Computer Networks," *Proc. IEEE Phoenix Conf. Comp. & Commun.*, pp. 461-472, Mar. 1989.
[40] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, Vol. 1, General Techniques and Regular Problems, Prentice Hall, 1988.
[41] G. S. Stiles, "On the Speedup of Simultaneously Executed Randomized Algorithms," Submitted to *IEEE Trans. Parallel & Distributed Sys.*, Oct. 1992.
[42] G. S. Stiles, P. Julien-Laferriere, F. H. A. Lee, and J. M. Favre "Stochastic Optimization of Distributed Database Networks on a System of Transputers," Occam Users Group Meeting, March, 1987.
[43] G. S. Stiles, F. H. Lee, and P. Julien-Laferriere, "Parallel Simulated Annealing Optimization of Distributed Database Networks," Third SIAM Conference on Parallel Processing for Scientific Computing, Dec., 1987.
[44] R. Jayaraman and F. Darema, "Error Tolerance of Parallel Simulated Annealing Technique," *Proc. Int'l. Conf. Computer-Aided Design*, IEEE, 1988.
[45] E. H. L. Aarts, E. M. J. de Bont, J. H. A. Habors, and P. J. M. van Laarhoven, "Parallel Implementations of the Statistical Cooling Algorithm," *Integration*, 4, pp. 209-238, 1986.
[46] F. H. Lee, Parallel Stochastic Optimization of Distributed Database Network, MS Thesis, EE, USU, 1988.
[47] G. S. Stiles, K. W. Bosworth, T. W. Morgan, F. H. Lee, and R. J. Pennington, "Parallel Optimization of Distributed Database Networks," *Proc. 1st Int'l. Conf. Appl. Transputers*, pp. 1-19, Aug. 1989.
[48] F. H. Lee and G. S. Stiles, "Parallel Simulated Annealing: Several Approaches," *Proc. 2nd Conf. NATUG*, IOS Press, Amsterdam, pp. 201-214, Oct. 1989.
[49] N. Dodd, "Slow Annealing Versus Multiple Fast Annealing Runs - An Emperical Investigation," *Parallel Computing*, 16, IOS Press, Amsterdam, pp. 269-272, 1990.
[50] G. Ruppeiner, J. M. Pedersen, and P. Salamon, "Ensemble Approach to Simulated Annealing," *J. de Physique I*, 1, pp. 455-470, 1991.
[51] K. H. Hoffmann, P. Sibani, J. M. Pedersen, and P. Salamon, "Optimal Ensemble Size for Parallel Implementation of Simulated Annealing," *Applied Mathematics Letter*, 3, 3, pp. 53-56, 1990.

Section I : Languages, Compilers and Programming Systems

TPML: Parallel Meta Language for Scientific and Engineering Computations using Transputers

J.K. Hartley, A. Bargiela

Department of Computing
The Nottingham Trent University
Burton St, Nottingham NG1 4BU, U.K.
jkh@uk.ac.ntu.doc, andre@uk.ac.ntu.doc

Summary

The experience with the transputer implementation of parallel processing algorithms, in the field of real-time process control, has led to the development of a parallel meta language (TPML) which offers a generic tool for programming transputer platforms. The meta language complies with the Bulk Synchronous Parallel (BSP) processing model proposed by Valiant [1], and it is seen as a prototype for implementations on other distributed hardware architectures. This paper argues that the TPML offers a simple, yet general, solution to the problems associated with programming parallel hardware architectures.

The TPML simplifies the development of parallel programs by offering the facility of a logical specification of inter-task communications that abstracts from the physical detail of processor connectivity and task placement. Any alteration to the processor connectivity is handled internally by the TPML, so that the specification of communications in the application programs remains unchanged. The complete specification of the parallel processing is possible through a small set of TPML commands.

A prototype of the TPML has been implemented in 3L Parallel Fortran 77 targetted at a transputer platform. In order to assess the overheads associated with the use of TPML an existing application code has been converted to TPML.

Keywords

Distributed architectures, Parallel programming, Bulk Synchronous Parallel processing model.

1. Introduction

Parallel processing has always held a great promise of the high efficiency and scalability of computations. This is particularly attractive in scientific and engineering application domains. Most real-time systems, for example, require a fast response time alongwith high computational power - parallel processing is seen as a means for the achievement of these goals [2,3,4]. However, it must be noticed that, to date, parallel processing has been used to a lesser degree than initially expected. One reason for such a situation is the inherent complexity, stemming from the hardware dependent nature of programming parallel computers [5]. Scientists and engineers want to concentrate on solving problems from their application domains - and do not wish to be distracted by any low level parallel programming considerations [6].

Another, possibly more important, reason for the limited use of parallel hardware is the lack of portability of applications between various parallel computers. The diversity of parallel hardware designs means that it is exceedingly difficult to develop and test parallel applications on hardware that is different from the target computer. What appears to be a missing ingredient, in parallel systems development, is a logical model of parallel computations which would act as a reference model for both the parallel hardware designers and the software developers in much the same way as the von Neumann model acts as a reference for sequential processing.

The challenge is that the model needs to be sufficiently general to encompass a full spectrum of parallel processing architectures, from shared memory multiprocessors through to local area networks. In particular, the model needs to encompass the message passing and the shared memory programming paradigms of parallel systems [7]. A recently proposed candidate to such a role is a Bulk Synchronous Parallel (BSP) processing model [1]. Although it has been originally intended to be a tool for the evaluation of computational complexity of parallel algorithms, it seems to be well suited for a more general role of a logical model of parallel computation. In essence, the BSP model assumes that any parallel processing can be expressed as an alternating sequence of parallel computations and communications. In order to maintain the integrity of the data communicated between tasks, all parallel computational processes synchronise before the communication begins - thus, the Bulk Synchronous Processing name. The BSP model is general in that it does not specify the nature of interprocess communications which can be implemented either by means of sending/receiving messages or reading/writing global variables.

A parallel meta language - TPML, developed at The Nottingham Trent University, represents an implementation of a BSP processing model. It has been originally developed and implemented as an extension to the parallel (3L) Fortran for a transputer platform. This development can be easily extended to any MIMD (multiple instruction, multiple data streams) computer and to other programming languages.

A meta language approach to the implementation of the BSP processing model was adopted in order to make full use of the available software development tools. However, it is envisaged that in future the TPML functionality will be available through a dedicated parallel compiler [8].

2. Parallel Meta Language (TPML)

The parallel meta language enables a logical specification of parallel tasks and intertask communications. This operation is performed while hiding the message transport complexities inherent to a specific physical processor connectivity and/or processor-memory arrangements. The execution of the meta language statements has the effect of generating application code in 3L Parallel Fortran and generating all the hardware specific information that is necessary to support the execution of parallel tasks. Since the major TPML design objective was to facilitate easy transition from hardware specific to hardware independent parallel software development, the design decisions favour simplicity in preference of optimality.

This TPML implementation assumes a coarse grained model of parallel processing with individual tasks typically allocated to separate processing nodes - although processors may possess a number of tasks, or indeed be redundant. Communication between these tasks is performed by sending messages. Within this model of computation each processor is assigned an automatically generated auxiliary message routing task which is an agent facilitating inter-task data communication. This task implements a virtual channel of communication between its associated computational tasks and every other computational task. The routing task is individualised by the inclusion of information about physical processor connectivity, which is optimised so as to provide the most efficient links between the computing nodes.

If the number of processors is less than the number of computational tasks, more than one task must be placed on the same processor. On such a processor, TPML generates just one router. While, conceptually, any computational task could be placed on any processing node, in the case of 3L Fortran, one needs to ensure that the I/O statements are contained only within the tasks placed on the root processor. This is because the I/O statements must have access to the afserver process via the filter task, which is positioned on the root processor. If the number of processors exceeds the number of tasks, the tasks can be remotely placed on these processors.

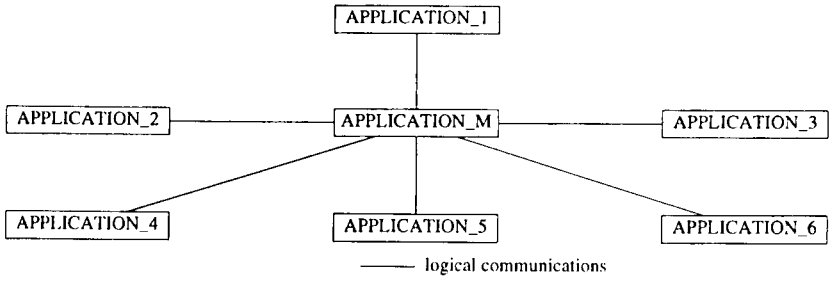


Figure 1a Logical inter-task communications

Figure 1a illustrates the logical view of a parallel program as specified with the TPML. In this example, the master task APPLICATION_M is in communication with every other application task (APPLICATION_1 through to APPLICATION_6). There is no communication between these slave tasks, thus the logical commmunication network has a star topology.

TPML derives appropriate switching vectors for each router, which ensures that the optimal route is followed from one task to another, given the placement of these tasks on the processors. The connectivity between the routers mirrors the physical connectivity of the processing nodes, while the logical connectivity of tasks is modelled by the appropriate values of the switching vectors. Figure 1b illustrates the configuration of the routers. It is worth noting, that in order to map the star topology of the logical connectivity of tasks, onto the physical connectivity of processors, only a subset of entries in the switching vectors is used, as indicated in Figure 1b.

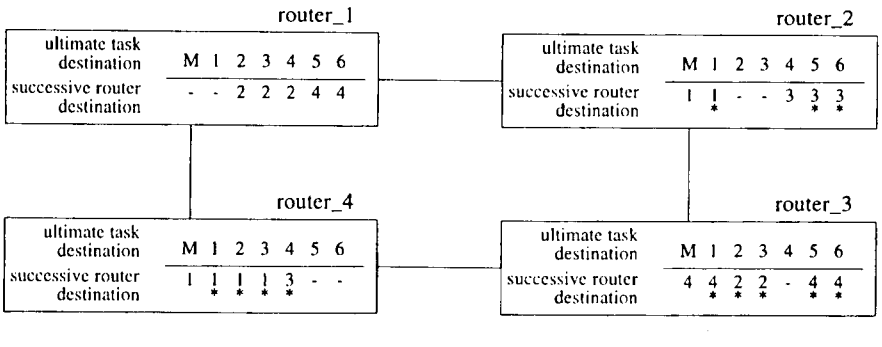


Figure 1b Routing tasks configuration

The configuration of the physical system, in terms of task placement and the processor connectivity is illustrated in Figure 1c. Clearly this configuration is dependent on the number of available processors and the links between them. The advantage of using the TPML is that it enables the application programmer to abstract from the hardware dependent considerations (Figures 1b and 1c) and to concentrate on the hardware independent logical view (Figure 1a).

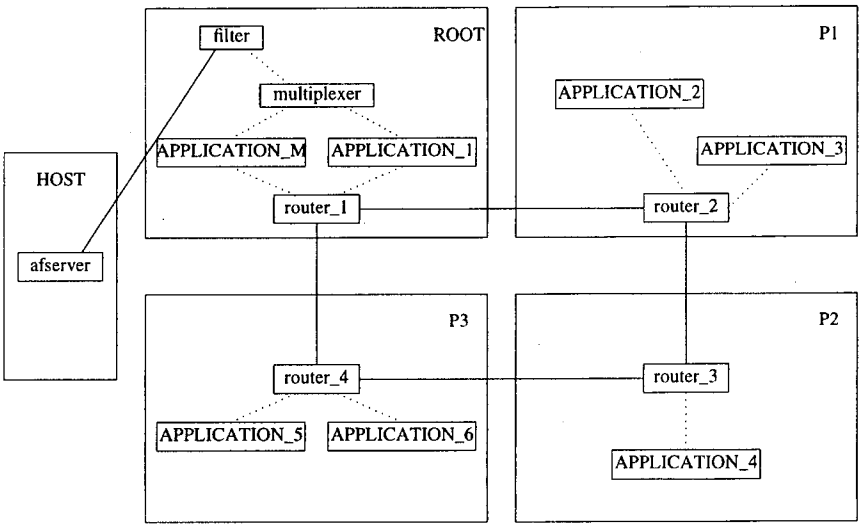


Figure 1c. Physical system configuration

3. TPML Syntax

The syntax of the TPML, provided for use within the application code, includes three sets of commands for the respective management of: tasks, intertask communication and superstep synchronisation.

Task management:

Each task is bracketted by a pair

c>>tpml_task_begin(task_id) or **c>>tpml_task_begin(task_id, same_task_id)**

and

c>>tpml_task_end

The user includes the code for the task named 'task_id' within these expressions. The code must include all declarations of variables found within the task, alongwith any subroutines or functions that might be accessed during the execution of the program. In the case of a number of identical tasks being created, it is not necessary to duplicate the code for each task. Instead, the initialising command is given two arguments - the first naming the task, and the second defining the task to be copied.

Intertask communication:

c>>tpml_send(destination_task_id, data, length)
and
c>>tpml_receive(data)

The tpml_send command specifies to which task a data packet is to be sent. In order to keep the implementation of the meta language simple, we have allowed that the data packet is of an integer array type only. It has been left to the programmer to pack/unpack the data of other types into this array. By explicitly specifying the number of elements in that array, we minimise the volume of data on the communication links. To receive a data packet, TPML requires the programmer to specify only the name of the array to which the incoming data packet is to be assigned. This is so, because the data packets always arrive from the same source - the router.

Superstep synchronization:

c>>tpml_step_begin(step_id)
and
c>>tpml_step_end

The superstep facility serves to ensure that the access to the shared data by the parallel tasks is properly synchronised. The 'step_id' parameter, identifies a superstep within each parallel task. Execution of parallel tasks proceeds through all the supersteps in the same order. This implies that the feasible interleaving sequences of the concurrent tasks are well controlled, but the application programmer is required to ensure that all the tasks proceed through the supersteps in the same order. Within the brackets: **c>>tpml_step_begin** and **c>>tpml_step_end**, each task may perform different computations but they must all reach the end of their current supersteps before any one can proceed to the next. The superstep facility provides a barrier synchronisation for the participating tasks.

The superstep has been implemented as a separate task which holds information concerning the amount of supersteps of each identity, positioned within the application code. The superstep task receives messages from the application tasks, indicating that that particular task's individual superstep, with identity 'step_id', has been completed. These messages are queued internally and counted by the superstep task and if they balance out with the number of expected supersteps of type 'step_id', the permissions to enter the next superstep are granted.

For programming reasons, an auxiliary facility of an explicit label to mark the end of the application code has been included within the TPML. Its syntax is:

c>>tpml_end.

Also, the command

c>>tpml_kill

has been introduced to ensure that the execution of continual loops in the application tasks is discontinued if required.

4. Meta Language Compilation

The compilation of a parallel application code written using the TPML syntax includes a number of effects:

- extraction of the code (duplication if necessary),
- extraction of information on processor topology,
- generation of router tasks and their customisation,
- augmentation of the user code by the communication commands,
- generation of a configuration file.

4.1 Generation of tasks

The application tasks are generated by a single pass along the parallel code, specified in TPML. The beginning and end of each task's code are indicated by the inclusion of the commands **c>>tpml_task_begin** and **c>>tpml_task_end**. The code contained between these brackets is then written to a file headed by the declarations required by the TPML commands for communication and superstep implementation. Subroutines which take care of the execution of communications between the tasks are included at the end of the file. A library of subroutines is available, and specific versions are selected according to the location of the task on a particular processor.

The router tasks are constructed by adding details of a physical hardware to the logical specification of inter-task communication (as programmed in TPML). The additional information is retrieved automatically by the TPML compiler and consists of: the number of available processors, the processor connectivity and the number of communication ports needed. Based on this, the TPML provides automatic generation of routing paths from one task to any other, ensuring that the minimum length path between tasks is followed. These paths are generated by building reachability trees from each task to all other tasks, thus eliminating the possibility of loops within the designated paths. An individual array, indicating the output port to be selected given the destination task of the message, is included within the code for each router (see Figure 1b).

The creation of the superstep task is implicit in the reference to the **c>>tpml_step_begin/end** commands. The structure of this task is determined by analysing the TPML source code and counting the number of times a superstep of identity 'step_id' is encountered in the corresponding tasks.

4.2 Augmentation of code by communication commands

The TPML model of parallel processing is that of the communicating sequential processes. All data is exchanged between the application tasks by means of explicit unsynchronised send and receive calls. The sending and receiving tasks are decoupled by means of the communication routers which are responsible for this message delivery. The implicit synchronisation between the application and the communication router task, makes it possible to raise more than one communication request in a single BSP superstep. All of these router tasks are optimised to be structurally deadlock free, ensuring the completion of program executions.

This logical model of communication is mapped, within the transputer systems environment onto the physical communications over channels. A channel connects exactly one process to exactly one other process, carrying messages in one direction only. If a bi-directional communication between two processes is required, two channels must be used. Each process can have any number of input and output channnels, but the channels in a system are fixed - new channels cannot be created dynamically.

An important property of transputer channel communications is that they are synchronised. A process wanting to send a message over a channel is always forced to wait until the receiving process reads the message. So, the TPML must ascertain that the parallel program is structurally free from deadlock. This problem is resolved by the inclusion of a buffer space in each router so that the sending and receiving tasks are de-synchronised.

The programmer must supply the array to be communicated ('data'), alongwith its size ('length') and logical destination ('destination_task_id'), within the command
c>>tpml_send(destination_task_id, data, length).

No specification of physical addresses or routing information is required. Similarly, within the code of the destination task, the programmer must indicate to which array ('data') the received message must be written -
c>>tpml_receive(data).

The commands **c>>tpml_send** and **c>>tpml_receive** are substituted by appropriate calls to the corresponding subroutines, specified below. These subroutines are included at the end of each application task's code:

```
      subroutine tpml_send(task,data,length)
c     ==============================
c --- Send integer array 'data' to 'task' via a router
      CALL F77_CHAN_OUT_WORD(4*length, outport1)
      CALL F77_CHAN_OUT_MESSAGE(4*length, data, outport1)
      CALL F77_CHAN_OUT_WORD(task, outport1)
      return
      end
c
      subroutine tpml_receive(data,length)
c     ============================
c --- Send message to router to indicate that the task is ready to receive a packet
      CALL F77_CHAN_OUT_WORD(continue, outport1)
c --- Task receives a packet in the form of an integer array 'data'
      CALL F77_CHAN_IN_WORD(length*4, inport1)
      CALL F77_CHAN_IN_MESSAGE(length*4, data, inport1)
      CALL F77_CHAN_IN_WORD(task, inport1)
      return
      end
```

The formal parameter 'length' in the tpml_receive subroutine is private to TPML and is used whilst unpacking the data array.

The command **c>>tpml_step_end** also indicates a call to the tpml_send subroutine, with 'task' being denoted by the superstep task.

4.3 Generation of the configuration file

The configuration file is a file read by the afserver program determining which computing nodes of the transputer platform should be used by the specific application tasks.

The configuration file specifies:
- the processors, and the wires connecting them
- the names of the files containing the component tasks of the application
- the connections between the various tasks' ports
- the placement of particular tasks onto particular processors in the physical network.

The TPML is capable of generating any appropriate configuration file according to the present architecture. Arbitrary networks of transputers can be constructed simply by wiring together the transputers into a selected topology. Tasks placed on the same processor can have any number of interconnecting channels. Tasks placed on different processors can only be connected where physical wires connect the processors' links. Each logical connection, between two tasks placed on different processors, is assigned exclusive use of one of the physical link channels connecting these processors. The number of interconnections between tasks on different processors is therefore limited by the number of hardware links each one has (four). There are two channels assigned to communicate along each of these links, one in each direction. The implication of this

limitation for the TPML application is that all tasks requiring access to the afserver, to enable data to be sent along the I/O channels, must lie on the root processor. An example of the TPML-generated configuration file, tpml.cfg, is given below:

```
processor HOST
processor ROOT
wire ? ROOT[0] HOST[0]                          ! physical wire connection between the root and host
processor P001
wire ? P001[1] ROOT[2]
processor P002
wire ? P002[1] P001[2]
processor P002
wire ? P003[1] ROOT[2]
processor P003
wire ? P003[1] P002[2]
!
Task tafserver  Ins=1  Outs=1
Task filter     Ins=2  Outs=2  Data=50k
Task multiplexer   File=filemux    Ins=3  Outs=3  Data=  8k
Task tpml_router1   File=router_1   Ins=4  Outs=4     ! tpml_router1 has task image file router_1.b4
Task tpml_router2   File=router_2   Ins=4  Outs=4     ! tpml_router2 has 4 input and output ports
Task tpml_router3   File=router_3   Ins=3  Outs=3
Task tpml_router4   File=router_4   Ins=4  Outs=4
Task application_m  File=application_m  Ins=3  Outs=3   Data=500k
Task application_1  File=application_1  Ins=3  Outs=3   Data=500k
:
Task application_6  File=application_6  Ins=1  Outs=1   Data=500k
!
Place tafserver   Host
Place filter      Root
Place application_m   Root                        ! application_m is placed on the root processor
Place application_1   Root
:
Place application_6   p003
Place tpml_router1    Root
Place tpml_router2    p001
Place tpml_router3    p002
Place tpml_router4    p003
!
Connect ? tafserver[0]      filter[0]
Connect ? filter[0]         tafserver[0]
Connect ? filter[1]         multiplexer[0]
Connect ? multiplexer[0]    filter[1]
Connect ? multiplexer[1]    application_m[1]
Connect ? application_m[1]  multiplexer[1]        ! logical connection from application_m to task multiplexer
Connect ? multiplexer[2]    application_1[1]
Connect ? application_1[1]  multiplexer[2]
Connect ? tpml_router1[0]   application_m[2]
Connect ? application_m[2]  tpml_router1[0]
Connect ? tpml_router1[1]   application_1[2]
Connect ? application_[2]   tpml_router1[1]
Connect ? tpml_router1[2]   tpml_router2[0]       ! tpml_router1 has output port 2, tpml_router2 has input port 0
Connect ? tpml_router2[0]   tpml_router1[2]
:
Connect ? application_6[0]  tpml_router4[0]
Connect ? tpml_router4[0]   application_6[0]
```

The information about the number and the interconnections between the transputers is extracted by the TPML compiler automatically by means of the 'tworm' utility. According to how the processors are connected, the TPML compiler generates an appropriate configuration file. For example, when the transputer network is connected into a pipeline, each processor - other than the root and the last processor in the line - has associated two pairs of communication ports. However, when the transputer network is connected into a tree structure, the processors may use up to four pairs of communication ports. This requires customisation of the sending and receiving procedures which map the logical task connectivity onto the variants of physical processor connectivity.

The TPML assigns a data value of 50k to the filter task and 500k to each of the application tasks. These values can be altered by the programmer if these values are not appropriate. The router tasks use the remaining memory available to the corresponding processor. If required, the multiplexer task is placed on the root processor, allowing more than one task to use the standard file I/O, by merging a number of request streams into a single stream. The multiplexer task is typically allocated less than 10k of memory.

5. Compilation of the TPML Program

When the user types in the command line **tpml**, a list of system commands are automatically executed.

```
tworm -c> tpml.cfg          # generates processor interconnection information and writes it to the file tpml.cfg
trun t_parse.app            # invokes the afserver process to execute t_parse.app instigating compilation of
                              the programmer's code
tff -t -g application_m.f   # compiles application_m.f as a task to be later configured
tff -t -g application_1.f
tff -t -s -g application_2.f # compiles application_2.f with the stand-alone library as a task to be later
:                             configured
tff -t -s -g application_6.f
tff -t -s -g router_1.f
tff -t -s -g router_2.f
tff -t -s -g router_3.f
tff -t -s -g router_4.f
tconfig tpml.cfg tpml.app   # generates tpml.app from the interconnected task images specified in the file
                              tpml.cfg
trun tpml.app               # invokes the afserver process to execute tpml.app across the transputer network
```

6. Concluding Remarks

The parallel meta language (TPML) is an implementation of the BSP model of parallel computations, on the transputer platforms. The design of the language, however, is general and it is eminently implementable on other parallel computers. The TPML affords the programmer a logical specification of inter-task communications and itself elaborates on the detail of physical processor connectivity and task placement. Any alteration to the processor connectivity is handled internally by the TPML, so that the specification of communications in the application programs remains unchanged. The TPML has been applied to an existing distributed state estimator [9] - the development of an earlier diakoptical simulation algorithm implemented for use with nonlinear networks [10,11]. In implementing this task, TPML has proven itself to be a convenient means of portable parallel programming.

References

1. VALIANT, L.G. "A Bridging Model for Parallel Computation" *Communications of the ACM*, Vol. 33, No. 8, August 1990.

2. CHAJAKIS, E.D., ZENIOS, S.A. "Synchronous and Asynchronous Implementations of Relaxation Algorithms for Nonlinear Network Optimisation" *Parallel Computing*, Vol. 17, pp 873-94, 1991.

3. FU-CHAW YU, YAN-PEI WU "A Multilevel Diakoptics Algorithm for Large Scale Networks" *Journal of the Chinese Institute of Engineers*, Vol. 11, Iss. 6, pp 667-79, Nov. 1988.

4. LIU, Z., THORELLI, L-E. "HSIM: A Distributed Simulation Environment Based on Trollius and Transputers" Parallel Computing and Transputer Applications, edited by M.Valero et al, IOS Press/CIMNE, Barcelona, 1992.

5. PAZZINI, M., NAVAUX, P. "TRIX, A Multiprocessor Transputer-Based Operating System" Parallel Computing and Transputer Applications, edited by M.Valero et al, IOS Press/CIMNE, Barcelona, 1992.

6. GAJDAROV, D. "LYNX - A New Language for Parallel Programming" Parallel Computing and Transputer Applications, edited by M.Valero et al, IOS Press/CIMNE, Barcelona, 1992.

7. ARGILE, A., BARGIELA, A. "Using X11 Windows to Provide Shared Task-Memory in Distributed Systems" Integrated Computer Applications in Water Supply, Vol. 1, edited by B. Coulbeck, Research Studies Press, John Wiley, 1993, pp 305-16. Leicester, Sept. 1993.

8. ANDERSON, J., LAM, M. "Compiler Optimisations for Scalable Parallel Machines" *Parallel Update*, Vol. 17, Jan. 1994.

9. BARGIELA, A., HARTLEY, J.K. "Diakoptical State Estimation of Nonlinear Networks" in preparation *(preprints available from the authors)*.

10. HOSSEINZAMAN, A., BARGIELA, A. "Parallel Simulation of Nonlinear Networks using Diakoptics" *Proc. PACTA '92*, Sept. 1992.

11. BARGIELA, A. "Nonlinear Network Tearing Algorithm for Transputer System Implementation" *Proc. TAPA '92*, pp 19-24, Nov. 1992.

A rapid compiler prototyping system for fine-grained concurrence architectures

V. Evstigneev, V. Kasyanov
Institute of Informatics Systems,
Siberian Division, Russian Academy of Sciences,
Novosibirsk, 630090, Russia
E-mail: {eva,kvn}@isi.itfs.nsk.su

Summary

One of problems from parallel processing is the problem of a rapid development of a compiler prototype for a new computer. Having such compiler prototype one can begin to exploit the computer, to reveal its merits and demerits, and to evaluate its performance. To achieve this goal, Institute of Informatics Systems began to develop the tool for a rapid designing a compiler prototype oriented on a few input languages (Fortran 77, C, Modula 2, Pascal, SISAL) and on a few architecture families, in particular on the family of architectures exploited fine-grained parallelism. This family includes the VLIW, the superscalar, and other architectures.

Second goal that we want to achieve is to develop a tool for an investigation optimizing and restructuring transformations of programs to be parallelized. Our system called PROGRESS is similar to the DELTA system of Illinois University and to the TWINE system of Livermore National Laboratory.

This work is supported by the Russian Foundation for Fundamental Research under grant 93-012-576.

Key words

Transformational approach, parallel architecture, fine-grained parallelism, program optimization, program restructuring

1. Introduction

New parallel architectures require the new compilers directed on this architecture. One of problems here is the problem of a rapid development of a compiler prototype for a new computer. Having such compiler prototype one can begin to exploit the computer, to reveal its merits and demerits, and to evaluate its performance. To achieve this goal, Institute of Informatics Systems began to develop the tool for a rapid designing a compiler prototype oriented on a few input languages (Fortran 77, C, Modula 2, Pascal, SISAL) and on a few architecture families, in particular on the family of architectures exploited fine-grained parallelism. This family includes the VLIW, the superscalar one, the SCISM and other architectures.

Second goal that we want to achieve is to develop a tool for an investigation optimizing and restructuring transformations of programs to be parallelized. Our system called PROGRESS is similar to DELTA system of Illinois University and to TWINE system of Livermore National Laboratory.

Like DELTA our system is intended to manipulate with high-level language programs and to give as the output the modified program. It is designing to include a large collection of analysis and transformation functions which can be combined using a very-high-level language. In despite of DELTA and other similar systems, the system uses for programs transformed a number of intermediate representations (such as the abstract syntax tree, the control flow graph, the program dependence graph, etc) and manipulate with programs annotated.

The PROGRESS system as a whole is intended for supporting the following opportunities: to convert a sequential program into a parallel one written on a parallel dialect of the input

language, to optimize programs on input language level using various criterions of quality, to compare versions of the programs with respect to complexity, to include much of the capabilities of modern symbolic algebra systems to tune the translation process upon a given architecture to find systems of optimizing transformations.

The PROGRESS system consists of a number of subsystems; each subsystem is designing as an abstract data type — through the set of operations supported by this subsystem.

It is clear that program improvement can be viewed as successive transformations on source language representations of a program visible to programmer and subject to his direction and guidance. However, many authors (e.g. [1]) indicate the following. First, transformations are not applied in random order; the successful application of a transformation suggests successor transformations. Indeed,in a practical application, the most difficult software engineering problems are concerned with transformation ordering and information gathering. Second, although a transformation may be applicable, it may not win or cause an improvement in the program. Third, the distinction between machine-dependent and machine-independent portions of a compiler is more subtle than usually thought; a transformation on a program may be machine independent, in the usual sense, but the reason for applying it may well depend on the target machine architecture.

The approach we favor following [1] is to allow the programmer to be the strategist and to provide a mechanical assistant to perform the optimization itself. A canned set of transformations may improve the program sufficiently. Otherwise the programmer may have to experiment by applying transformations and measuring the resulting program repetitively until a sufficiently improved program results. The programmer indicates the transformations to perform; the system performs them mechanically, verifying that program equivalence is preserved or requesting that the programmer so verify.

2. Input languages

We plan to have as input such imperative languages as Fortran 77, C, Pascal, Modula-2. They are well known. Most of supercomputers have at least one from those languages.

We want to include in our system as an input language also the applicative language SISAL [2]. SISAL is strongly typed, general purpose functional language that supports data types and operations for scientific computing. SISAL has several important semantic properties. First, the language is mathematically sound-functions map inputs to outputs without side effects. Second, names are referentially transparent; that is, they stand for values rather than memory locations. Third, the language is single-assignment. A name may be assigned a value only once within each scope. A SISAL tenet, not required by its functional semantics but enforced by the compiler to aid readability, is that all names must be defined before they are used.

SISAL version 1.2 was introduced at 1985 and it has been realized for such computers as Cray Y-MP, Sun workstation and etc. We want to realize SISAL version 2.0 [3] or SISAL 90. In the paper [4], the comparision of SISAL and Fortran shows that successor of Fortran for supercomputing should be functional and performance is no longer an issue.

3. Target architectures

Our current interests are multiple-instruction-issue architectures and scalable distributed memory multiprocessors (multicomputers). Multiple-instruction-issue architectures or fine-grained parallelism exploited ones include VLIW, superscalar, superpipelined, and others machines. The importance those architectures counts of the existence of a tendency to offer both kinds parallelism: coarse-grained and fine-grained [5].

Very Large (or Long) Instruction Word (VLIW) architectures and superscalar architectures have the potential to execute multiple operations in a single machine cycle. The main challenge in these architectures is to detect and exploit as much parallelism as it is available in the appli-

cation program. Surprisingly, the reported values of instruction-level parallelism exploited in many applications have been rather low.

VLIW machines are the tightly-coupled synchronous parallel machines. Many commercial machines fall into this class, including horizontally microcoded engines, the Mars-432 (Numerix Corp.), FPS-164/264, Multiflow's series and Cydrome's Cydra. These machines have several distinguishing features. There is a single flow of control (i.e., a single program counter) and a global address space. Parallelism is achieved by multiple (synchronous) functional units. Finally, these machines are statically scheduled — virtually all runtime behavior is specified by the program.

In contrast with vector processors, which are capable of producing many results from a single (vector) instruction, the super-scalar machines can execute different types of operations per clock cycle. For this reason, a wider class of applications can be processed more efficiently by a super-scalar architecture whose performance can be comparable with the processing power of a vector processor. Classical examples of super-scalar machines are: the i860 and i960 from Intel, the RS/6000 from IBM, the MIPS-X, etc.

4. The PROGRESS System

We present here the PROGRESS system project for transforming programs where the programs are represented in different intermediate forms [6]. The main goal of this project is the development of a formal theory of a semantic analysis and transformations of programs and specifications. This goal includes: investigation of equivalence relations on various program models and transformations preserving such equivalencies; development of new algorithms for semantic analysis of programs and specifications; systematic description of program transformations and construction of a catalogue of transformation rules for programs and specifications; validation of the elaborated theory on some experimental systems for transformational development of efficient programs and for other semantic based program manipulations.

As result of this project a new method for construction of algorithms for semantic analysis, semantic error detection and program optimization will be proposed.

Systematic description of program transformations is the important problem in the area of program optimization and program restructuring.There are several catalogues of program optimizing transformation for sequential programs. For parallel programs, each transformation has a few version for various architectures and various forms of intermediate representations of programs. Since there are many parallel architectures and many different forms of representation programs finding of an optimal order of application of program transformations is very complicated. As a matter of fact there are following open problems in the parallel program transformation theory:

a) the choice of a program optimality criterion;

b) the finding an optimal order to apply program transformations;

c) the development of effective approximate methods for the solving of discrete optimization problems that arise when global program transformations are applied;

d) the choice of an optimal form of the intermediate representation of a program to be transformed.

These problems require to study known program transformations and to create a catalogue of transformation rules and its versions. Without that catalogue, solving the problem of a composite transformation design and an evaluation of its effort on the parallel program quality is almost impossible.

The PROGRESS system is intended to study currently available transformations and to create new ones (and composite transformations consisting from existing ones) for programs written in high-level languages (Fortran 77, C, Modula-2, SISAL and so on) which are extended by facilities to annotate programs. These facilities are intended to allow:

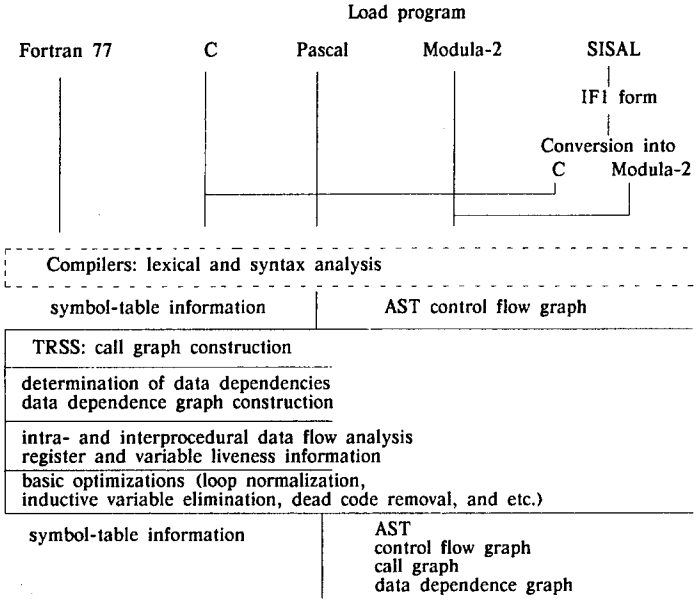
a) (user) to control program manipulation process and to supply the missing information about program (for example, specific knowledge about the range of data values which will be input to the program);

b) (system) to comment the resulting program by some information about program manipulation process.

Since we also want to use this system for teaching goals and for experiments with program transformations PROGRESS as a whole system for program manipulations at the programming language level will consist of following main subsystems: the translating subsystem TRSS which converts a source program into the basic intermediate form; the intermediate representation subsystem IRSS which converts the basic program representation into a given form (a program dependence graph, a hierarchical task graph, ideograph and so on); the transformation subsystem TSS which implements program analysis and transformations, both machine-independent and machine-dependent ones; the retranslating subsystem RTSS which converts an intermediate program into high-level language program to pretty print; the evaluation subsystem EVSS which estimates a resulting program quality by static analysis of the program and simulation of it run on the hardware; a set of code generators to generate cod for the given target machine; a set of simulators of target machines etc.

4.1. Translating subsystem TRSS

The translating subsystem TRSS plays a role of the system manager which organizes and manipulates system components. As input, TRSS has the original program text written in Fortran, C, Modula-2, Pascal or SISAL. A program or a program fragment to be investigated (parallelized) is analyzed and transformed by TRSS compilers into a basic intermediate representation which is used as an input for other subsystems such as IRSS or TSS. For support further manipulations, the TRSS performs also the dependency analysis. As result we obtain symbol-table and data-dependency information for reference by the IRSS and TSS subsystems. TRSS performs also a data flow analysis, built the call graph and performs intra- and interprocedural analysis. In order to make possible the data dependency analysis the TRSS performs some standard optimizations, e.g. loop normalization, dead code removal, and etc. For processing SISAL programs, TRSS includes the special tool to convert a SISAL program into the IF1 form and then into the Modula-2 (or C) program.

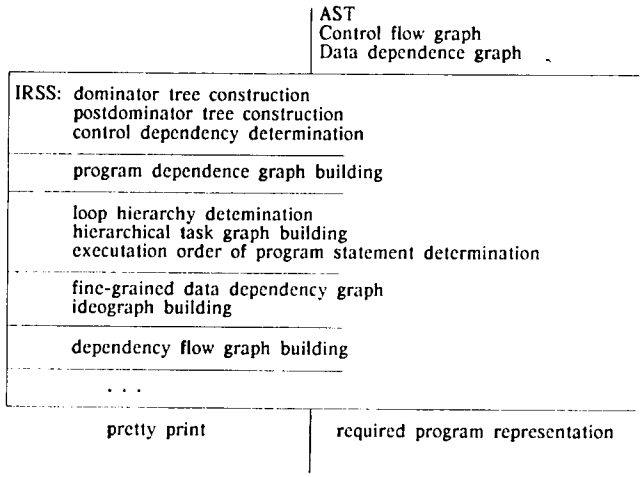


4.2. Intermediate representation subsystem IRSS

As an input, IRSS obtains from TRSS a program in the form of an AST and a control flow graph, and also an information about a data dependency. IRSS builts the representation to be required:

a) the program dependence graph [7];
b) the hierarchical task graph [8,9];
c) the ideograph [10];
d) the dependence flow graph [11];
e) the program dependence web [12]; and etc.

To achieve this goal IRSS designs dominator and postdominator trees, finds control dependencies, and determine the execution order of program operators.

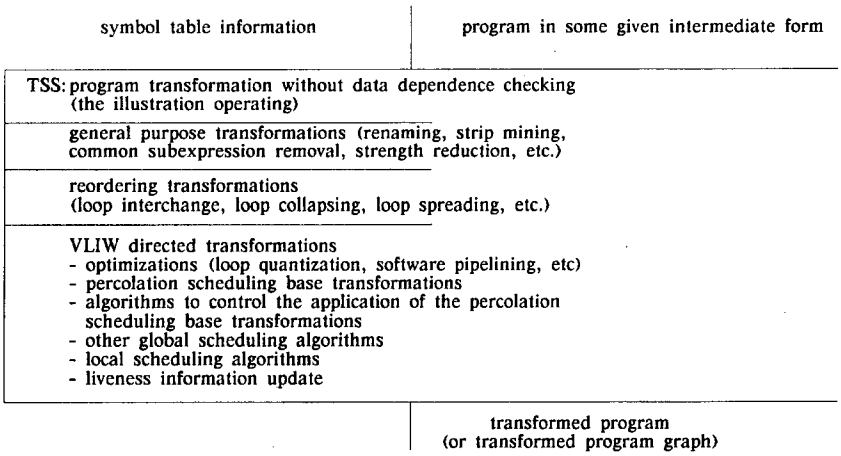


4.3. Transformation subsystem TSS (transformer)

As an input, TSS obtains a program in the given intermediate form and executes transformations which are indicated by an user. Transformations are storing in the transformation file. A transformations may be applied both to a whole program and to a program part such as a basic block or a loop nest.

For teaching aim, a source program written on a high level language can be transformed by a symbolic algebra technique into the output program written in same language without a program dependencies checking. In general case, a transformation is applied if it is possible.

TSS includes also the set of fine-grained restructuring transformations for the creation of long instruction words. For example, they are Percolation Scheduling base transformations [13, 14] (Aiken's version of Trace scheduling and Compact Global [15], resource-constrained version of Percolation Scheduling [16] and others). These methods perform scheduling on the program graph (control flow graph) and provide semantics-preserving transformations for reordering. Percolation Scheduling can be extended by loop unrolling and software pipelining.



5. Current state and related works

Now, subsystems TRSS, IRSS and TSS are under development; we make also the program transformation catalogue.

Today, many tools and environments are being constructed to coordinate the disjoint activities of editing, debugging, and tuning complex applications designed to run on parallel architectures. These are Faust [17], PTOPP [18, 19], ParaScope [20], and others. From our point of view, the DELTA system (University of Illinois at Urbana-Champaign) [21] is the most suitable prototype for our project.

References

[1]. D.B. Loveman. Program improvement by source-to-source transformation // J. ACM. — 1977. — Vol. 24, N 1. — P. 121—145.

[2]. J.T. Feo. Sisal. — LLNL. Preprint UCRL-JC-110915, July 1992.

[3]. A. P. W. Boehm, R. R. Oldenhoeft, D. C. Cann, J.T. Feo. The SISAL 2.0 Reference Manual. — LLNL, Preprint UCRL-MA-109098, Dec. 1991.

[4]. D. Cann. Retire Fortran? A debate rekindled // Comm. ACM. — 1992. — Vol. 35, N 8. — P. 81—89.

[5]. V. Evstigneev. Parallel processing: architectures and computers. — ISI of. Sib. Div. of the RAS. Preprint 22, Novosibirsk, 1993.

[6]. V. Evstigneev, V. Kasyanov. The PROGRESS program manipulation system // Proc. of the Intern. Conf. "Parallel Computing Techn." — Obninsk, Aug. 30—Sept. 4, 1993. — Vol. III, 1993. — P. 651-656.

[7]. J. Ferrante, K.J. Ottenstein, J.D. Warren. The program dependence graph and its use in optimization // ACM TOPLAS. — 1987. — Vol. 9, N 3. — P. 319—349.

[8]. M. B. Gircar. Functional parallelism: theoretical foundations and implementation. — PhD thesis. Univ. of Illinois, CSRD Rep. 1182. Dec. 1991.

[9]. M. Gircar, C.D. Polychronopoulos. The HTG: An intermediate representation for programs based on control and data dependencies. — Univ. of Illinois, CSRD Rep. 1046, May 1991.

[10]. S. S. Wang, A. K. Uht. Program optimization with ideograph // 1989 Intern. Conf. on Parallel Processing. — P. 153—159.

[11] K. Pingali, M. Beck, et al. Dependence flow graphs: An algebraical approach to program dependencies. — Cornell Univ. Techn. Rep. N 90-1151. Sept. 1990.

[12]. R. A. Balance, A. B. Maccabe, K. J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages // Proc. of the 1990 SIGPLAN Conf. on Progr. Lang. Design and Implementation. — SIGPLAN Notices. — 1990. — Vol. 25, N 6. — P. 2-57 — 2-71.

[13]. A. Nicolau. Percolation Scheduling: A Parallel Compilation Technique. Cornell Univ. Techn. Rep. TR 85-678. May 1985.

[14]. V. Evstigneev. Some peculiarities of the software for the computers with large instruction word // Programmirovanije. — 1991. — N 2. — 69-80. (In Russian).

[15]. A. Aiken. Compaction-based parallelization. PhD Dissertation. Cornell Univ. Techn. Rep. N TR 88-922. June 1988.

[16]. K. Ebcioglu, A. Nicolau. A global resource-constrained parallelization technique // Proc. Int. Conf. on Supercomputing, Crete 1989.

[17]. V.A. Guarna,Jr., D. Gannon, D. Jablonowski, A. D. Malony, Y. Gaur. Faust: An integrated environment for parallel programming // IEEE Software. — 1989. — July. — 20-27. (CSRD Rep. 825, Nov. 1988)

[18]. P. E. McClaughry. PTOPP — A practical toollset for the optimization of parallel programs. CSRD Rep. 1225, May 1992.

[19]. R. Eigenmann, P. McClaughry. Practical tools for optimizing parallel programs. CSRD Rep. 1278, Jan. 1993.

[20]. K. Kennedy, K.S. McKKinley, C.-W. Tseng. Interactive parallel programming using the ParaScope editor // IEEE Trans. Parallel and Distrib. Syst. — 1991. — Vol. 2. — P.329—341.

[21]. D. Padua. The Delta program manipulation system. Preliminary design. CSRD Rep. 880. June 1989.

A Parallelizer for a Language without Variables

Raymundo Ortega (e-mail: ortega@enstb.enst-bretagne.fr)
Daniel Bourget (e-mail: bourget@enstb.enst-bretagne.fr)
Laboratoire d'Informatique de Brest
Télécom Bretagne
BP 832, 29285 Brest Cedex, France
phone: (33) 98.00.14.30 fax: (33) 98.00.12.82

Abstract

This paper describes the GraalP parallelizer for a language without variables: ParGraal[6]. This language is a parallel version of the Graal language[3], which is an applicative language based on the FP systems principles[2]. It contains annotations to indicate the instructions to be evaluated in parallel. Annotations are as simple as the future instruction of some parallel Lisp languages. GraalP makes a static analysis of the program to be divided up. It constructs a dependences graph of the program being analyzed. The restrictions imposed on the model enable cyclic graphs to be transformed into acyclic ones. They are the most commonly treated in works[19], [17], [8], [12], [16], [20], [4], [13]. The constructed dependences graph is divided in order to find portions to be evaluated in parallel; a heuristic is used in order to achieve this goal. Finally, the whole program is rewritten with annotations, which indicate the portions of parallel evaluation.

Keywords: parallelizer, applicative language, FP systems, dependences graph, parallel evaluation.

1 Introduction

An automatic program parallelization mechanism implies program transformation. This is the conversion of sequential programs into ones that can be executed in parallel. This transformation is not always as easy as can be imagined. What is "parallelizable"? This question poses a really big problem in carrying out the transformation; answers are numerous, and sometimes no answer can be found.

From the instructions point of view, any sequential program contains an execution order. But it is possible to find suborders that can be calculated in parallel by a processor or a generated task. Program transformation consists in finding these instruction suborders for which the execution can be done in parallel.

In order to parallelize a program, not only is function and data dependences analysis necessary, but the number and the size of tasks(into which the program is being divided) should also be the most suitable for efficient execution; this means granularity should be optimal. Program efficiency depends completely on an adequated granularity: if it is not optimal then performances will not be optimal either.

When a program has been fragmented into many small tasks, parallel execution time can be greater than sequential execution time. This efficiency loss is because the time passed to communicate data has been increased. On the other hand, when the number of task divisions has been limited, there will be just a few large tasks and so less time dedicated to data communication. However, there can still be some instruction suborders which could have been evaluated in parallel.

This shows the necessity for research on models for grouping instructions into suborders. These models should minimize the parallel execution time in relation to the sequential one. Yet, they give only a static approximation of

program functioning, specially when its execution time depends on its input data[1].

In this article we present the GraalP parallelizer which is based on a heuristic model, in section 1, we give an introduction with a brief description of the parallelizer and some other models, in section 2, the Graal language description and the representation of programs written in Graal is given, and in section 3, the heuristic used. Finally, in section 4 we give some conclusions.

1.1 Global view of its operation

We present a parallelizer (GraalP) for Graal language[3], which is a language based on the principles of Backus FP systems [2]. The main characteristic of this language is the absence of variables, which means the non existence of data dependences and so of a data dependences graph.

GraalP makes a static analysis of the program. The way it functions is easy:

Firstly, user function definitions are separated from user defined function applications.

Secondly, the function dependences global graph is constructed. This is done by using the correspondence between programs in FP systems and tree forests[10]; this consideration can be extended to Graal programs because they belong to FP systems too.

Thirdly, the constructed graph (actually a tree forest) is explored from left to right and bottom-up using a heuristic to determine parallel execution regions. The graph to be explored can contain cycles, in the case of recursive functions, but restrictions imposed on the model may cause it to become acyclic. This is the graph type most commonly used in: [19], [17], [8], [12], [16], [20], [4], [13], whilst in [14] cyclic graphs are transformed into acyclic ones.

Graph exploration gives a function costs final environment (dictionary), by contrast with the initial environment which contains only costs for primitive functions. This initial environment is pre-defined and may be changed by the user, in order to adapt it to the machine used. For each primitive function, it defines a sequential execution cost and an average communications cost. This latter is useful for calculating the parallel execution cost of the whole application in which the function appears. The final environment computation enables to find sequential and parallel execution costs for the user defined functions; they are unknown at the beginning. So, at the graph exploration end, all functions have cost execution estimations.

Then, applications are explored to find those of parallel evaluation. The costs final environment is used in this exploration.

Finally, the whole program is rewritten with the annotation **par** before the instructions which will be executed in parallel. An annotation **par** indicates to the compiler the creation of a parallel task to evaluate the functional expression where it appears. The stated principles are similar to those used in some other functional languages for example Lisp MaRS[9].

1.2 Other models

In[19] an algorithm to divide a program is given. It takes into account factors such as execution and communication time costs. They are evaluated by deep first exploration of the program graph, which must be acyclic. The program graph is partitioned a lot of times, until it is found that no more partitions can be made. For each graph partition a function is evaluated. The divided program graph which minimizes the function (the best function value) is selected as that of optimal execution. A similar algorithm is used in [13].

In[17] program tree nodes are grouped into internal nodes called "clans". After this grouping, the program tree is explored, in a bottom-up way, to find leaves costs: they will be useful in determining node costs. They are determined as a function of the execution time and input-output time, in the same way as the assignment of nodes to processors is.

In our model, we use cost evaluation from left to right and bottom-up, but costs represent the sequential and parallel execution times: they are propagated from the bottom to the top in just one exploration. A very important characteristic of the compiler for the ParGraal language[6] is its capacity to generate a theoretical undefined number of tasks. In reality it is limited for recursive calls[5]; However, this fact makes it

possible to divide a program without regrouping the parts to adapt the program to a particular architecture, limited by a number of processors.

2 Graal language description

Graal[3] is a language based on the theoretical language defined by FP systems[2]. So, it inherits all properties defined for them.

2.1 Graal Principles

The basic principles of the FP systems and the Graal language are:

1) A set of objets O.

An X element belongs to the set if one of the following propositions is true:

$X = \perp$	the undefined object
X= atom	(of "Lisp type")
X= T or X=F	the boolean values: true and false
X=< >	the empty sequence
$X = < X_1, ..., X_n >$	where: $X_1, .., X_n$ are objets

2) A functions set F. It is the union of three other sets: $\mathbf{F = PF \cup FF \cup DF}$

- **PF** is the primitive functions set. It contains the arithmetic and boolean functions and all the basic functions for working with lists.
- **FF** is the functional forms set. They are functions more complicated than primitive functions. One of them (named **o**) enables functions to be composed, in order to create new functions belonging to F.
  f:g:<x> = (f o g):<x>
- **DF** is the user defined functions set. They must have the following form:
  def func = m
  func is the function name. **m** is the function body, which can be a function of func (the case of a recursive function).

3) An operation: "the function application". In FP systems all functions are 1-ary (the argument is always a sequence) and there is only one operation to execute them: the function application. If f is a function and x is an argument, then the application of f to x is represented by f:x.

A program is defined as the non empty set of function definitions given by the user. This is expressed by:

$P = \{ \text{def } f_1, \text{def } f_2, ...\text{def } f_n \} \neq \{\}$
where def f_1 is the main definition

The main Graal language characteristic is the absence of variables. This is possible because there is a primitive function which allows an element of a sequence to be selected:

$k:< X_1, ..., X_n > = X_k$ and $1 \leq k \leq n$.

A function to calculate the factorial is given here, as a Graal program example:

```
(de fac
    (if (zerop o 1)
        '1
        {mul 1 fac o sub1} ))
```

See [3] for a better description of the Graal language.

2.2 The Graal parallel language

The Graal parallel language (ParGraal)[6] is a parallel implementation of Graal. It inherits all functions and properties of the Graal language. It uses the **par** annotation [6] to indicate the parallel evaluation of function arguments. The ParGraal language compiler was designed in Ada. It generates a task to evaluate each parallel argument. Nevertheless, Graal polyadicity[3] has caused serious problems as far as tasks administration (arguments) is concerned. To avoid these problems, a task responsible for the arguments arrival is automatically generated. Its structure depends on the number of parallel arguments and it must wait for them to arrive before applying the main function.

The Graal language was implemented in "C". Each user defined function has an associated symbol. Each symbol has four basic fields[3]:

Pname	This is the symbol indentifier
Ftyp	This is the symbol type
Fval	This contains the symbol value

In addition to these base fields, four others[5] were added in the design of Graal parallel implementation:

Parallel	This indicates if the function can be parallelized.
Verrou	This forbids modifications to a symbol. It plays an important role in the case of shared elements.
Nbproc	Number of **Process**, allows recursive calls to be limited.
Pack	This specifies the package name where the symbol has been defined.

The user can use the form **par**[5], followed by a function name, to set the symbol parallel field to true . The parallel field is set to false by the form **no_par** [5].

So, the user is responsible for defining which functions will be evaluated in parallel. He controls the parallelization level, but the parallelization of mutually exclusive instructions is forbidden [5]. This is because only one result will be useful for the rest of the program. In any case, a minor parameter modification, at compiler installation time, removes this restriction for all conditional execution instructions.

2.3 The graph of a Graal program

The function dependences program graph should be constructed to determine the instructions set which can be executed in parallel. By definition, there is dependence between two instructions if there is an edge or an edges path that allows them to be joined. Program execution should maintain the order defined by the different edges.

Two instructions can be executed in parallel if they are not dependent and if their results will be useful to the rest of the program. The second part of the above condition is not necessary, and it can be bypassed, but in this case there will be a waste of resources.

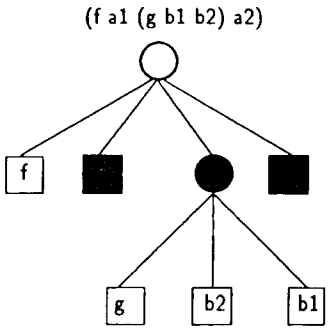


Figure 1: g must be evaluated before f, but it can be evaluated in parallel with a1 and a2

So the first thing to do is to find the Graal program graphical representation. This representation is based on the fact that in FP systems any program definition f can be seen as a functions composition[10]. This is expressed by:

$f = f_n \circ f_{n-1} \circ \ldots \circ f_1$
$f \in FP \cup FF \cup DF$
$f_i \in P \cup F \cup FF\text{-}$
FF- = is the functional forms set without the form for composing functions.

The previous formula is expressed in Graal[3] (braces mean n-ary function composition) by:

$f = \{ f_n\ f_{n-1} \ldots f_1 \}$

This formula represents a tree forest. The first element is a function, which can be represented by a tree, and the other elements are also functions that can be represented by trees.

A function application $f :< t_1, \ldots, t_n >$ can be expressed as a tree forest too. The first element represents the function and each sequence element, which can also be sequences, represents a tree subforest.

Because a Graal program consists only of function definitions and function applications, previous considerations allow us to consider any Graal program as a tree forest.

The graph G of a Graal program is made up of nodes and edges; the latter connects two

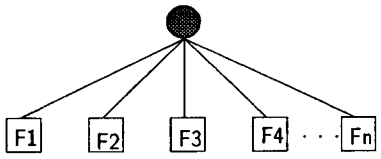


Figure 2: The representation of a forest

nodes. A node can also be constituted by a tree.

A Graal program graph G is:

G= (N E) where:

$N = \{ F_1, ..., F_n \}$
N is the nodes set, and its elements are trees or forests.

$E \subseteq N \times N$
E is the edges set, a subset of the Cartesian product of nodes.

Any elements of a forest can be one of the following:

- an irreducible element
- a tree
- another forest

The first element can be one of the following:

- a primitive function
- a functional form
- a user defined function

We consider a program as a forests set, in which edges of each tree and forest are labeled with execution costs (see 3). We assume too that user defined functions cannot be improved or changed. This restriction preserves the original functions semantics and is the reason why we do not modify or improve the given original function definitions. But the order in which the user has given definitions can be rearranged. This is done to ensure that all function costs will be found. We construct the function dependences program tree. It ensures that lowest level nodes, for example leaves, depend only on nodes of a higher level. In this way, when the program forest (which may be composed of trees) is explored, all costs of the used functions will be sure to be found.

3 The heuristic for cost assignation

Program transformation means grouping instructions into tasks. This grouping is done using the function dependences program graph. This graph contains the program execution order. However, the grouping must be done in such way that the parallel execution time is less than the sequential execution time.

3.1 Cost assignation

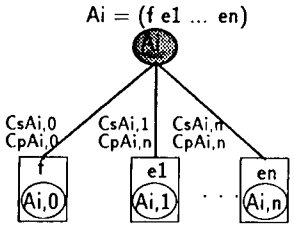
In the constructed dependences graph, each of the lowest level elements, an atom or a function, should contain its sequential and parallel execution time. But time is a parameter that depends on processor load, disk access, interruptions, processor speed[16] ... So, instead of using time to label the graph, global estimations of sequential and parallel times will be used. They will be named sequential and parallel execution costs.

At the beginning, all primitive functions have an associated sequential execution cost and an average communication cost. These costs are relative to the costs of a very primitive function: addition. The parallel execution cost of the application of a primitive function to arguments is found by using a heuristic. Sequential and parallel execution costs will follow a bottom-up propagation to determine nodes of parallel execution and to find the costs of user defined functions. These costs will constitute the final environment, in which each user defined function will have an associated sequential and parallel execution cost. This environment will be useful in treating applications of user defined functions.

3.2 Forest cost calculations

The forest sequential cost is calculated by adding to the first subforest sequential cost, which can be a function, the costs of all other subforests: these can be function arguments.

The usual way to calculate parallel cost is to take the greatest parallel costs of all subforests (this changes in the case of a primitive function



Ai,0 ≠ a primitive function

The sequential cost of the Ai forest is:

CsAi=sum(CsAi,j) --(I)

The parallel cost of the Ai forest is:

CpAi=max(CpAi,j) for (0 < j < n) --(II)

Figure 3: The calculations of parallel and sequential costs for a forest

application), because it is necessary to select the slowest subforest from the least fast. The slowest will take more time than the others to execute, so is therefore the worst case. The slowest subforest execution cost represents the whole forest execution time. This is illustrated in figure 3.

Let us consider the main subforest $A_{i,0}$, from the previous example. Let us suppose that $A_{i,0}$ is a primitive function f and that $e_1, e_2, ..., e_n$ are its arguments. Let us also to suppose that the sequential cost and the parallel cost have been determined for each argument, there is an associated forest $A_{i,j}$ $(1 \leq j \leq n)$ for each one. f function parallelization entails communications between the different tasks generated. So, we defined an average communication cost for each primitive function. It can be adjusted for every architecture and makes cost calculations easier. The product of this cost times the number of parallel tasks will give the total forest communication cost. But, to obtain the total forest parallel execution cost, it is necessary to add the forest sequential cost. The latter is equal to the division of the function sequential cost by the number of parallel tasks. This is because work will be shared by all different tasks.

The number of parallel tasks is the number of forests with a quotient (parallel cost/sequential cost) less than 1. This number is considered to be 1 when there is no forest which conforms to the previous criterion.

The mathematical expression for the formula expressed is:

$$A_{i,0} = f \qquad f = a\ primitive\ function$$

$$CpA_i = (CsA_{i,0}/np_i) + (np_i * Comms_i)$$

$CsA_i = Sequential\ cost\ of\ forest\ A_i$
$CpA_i = Parallel\ cost\ of\ forest\ A_i$
$Comms_i = Average\ communication\ cost$
$np_i = The\ number\ of\ parallel\ tasks$

$$np_i = 1 \qquad if\ \ \forall j \in [0..n]\ (CpA_{i,j}/CsA_{i,j}) \geq 1$$

in other cases:

$$np_i = \sum_{j=1}^{n} p_{i,j}$$

where:

$$p_{i,j} = 1 \qquad if\ \ (CpA_{i,j}/CsA_{i,j}) < 1$$
$$p_{i,j} = 0 \qquad if\ \ (CpA_{i,j}/CsA_{i,j}) \geq 1$$

3.3 The case of a recursive function

In the case of a recursive definition, a graph with cycles, sequential cost computation gives an equation of the following form:

$$CsA_i = k + CsA_i$$

This equation has only a trivial solution CsA_i=*0 et* $k = 0$. The equation for parallel cost computation has the following form:

$$CpA_i = max(k, CpA_i)$$

and its solution is: $CpA_i = k + m$, $m = [0, 1., ...]$

In order to avoid the drawbacks of these recursive equations, we assume the hypothesis that the sequential and parallel costs of a recursive function call are zero. This condition implies transforming a cyclic program graph into an acyclic one, which this is the case most commonly treated in [19], [17], [8], [12], [16], [20], [4], [13]. As a consequence of this, the sequential cost is equal to the sum of the sequential costs of all branches; and the parallel cost is equal to the greatest parallel costs of all branches. In both cases, the recursive calls costs are not considered due to the restrictions imposed before. A

simple representation of this is shown in figure 4. The A_i forest sequential is:

$$CsA_i = \sum_{j=0}^{n} CsA_{i,j}$$

$$CsA_i = \sum_{j=0}^{k-1} CsA_{i,j} + CsA_{i,k} + \sum_{q=k+1}^{n} CsA_{i,q}$$

$$CsA_i = \sum_{j=0}^{k-1} CsA_{i,j} + CsB_k + \sum_{q=k+1}^{n} CsA_{i,q}$$

$$CsA_i = \sum_{j=0}^{k-1} CsA_{i,j} + \sum_{r=0}^{n} CsB_{k,r} + \sum_{q=k+1}^{n} CsA_{i,q}$$

but $CsA_{i,0} = CsB_{k,0} = 0$ so it becomes:

$$CsA_i = \sum_{j=1}^{k-1} CsA_{i,j} + \sum_{r=1}^{n} CsB_{k,r} + \sum_{q=k+1}^{n} CsA_{i,q}$$

The parallel cost of the A_i forest is:

$$CpA_i = max(CpA_{i,j}) \; with \; (1 \leq j \leq n)$$
$$CpA_i = max(CpA_{i,0} \; ... \; CpA_{i,k} \; ... \; CpA_{i,n})$$
$$CpA_i = max(CpA_{i,0} \; ... \; CpB_k \; ... \; CpA_{i,n})$$
$$CpA_i = max(CpA_{i,j}, \; (max(CpB_{k,r})), \; CpA_{i,q})$$
$$with: \; (0 \leq j \leq k-1) \; ; (0 \leq r \leq n) \; ; (k \leq q \leq n)$$

but $CpA_{i,0} = CpB_{k,0} = 0$ so it becomes:

$$CpA_i = max(CpA_{i,j}, \; (max(CpB_{k,r})), \; CpA_{i,q})$$
$$with: \; (1 \leq j \leq k-1) \; ; (1 \leq p \leq n) \; ; (k \leq q \leq n)$$

$$CpA_i = max(CpA_{i,j}, CpB_{k,r}, CpA_{i,q})$$
$$with: \; (1 \leq j \leq k-1) \; ; (1 \leq p \leq n) \; ; (k \leq q \leq n)$$

This enables body function exploration, to find instructions to be parallelized, without the problem represented by recursive function calls. The exploration will give the costs of these kind of functions, they will included in the final environment. The final environment is equal to the initial environment plus the costs of all user defined functions. When it has been constructed, functions applications are processed. Finally the whole program is rewritten with the **par** annotation[18] to indicate those instructions of parallel execution.

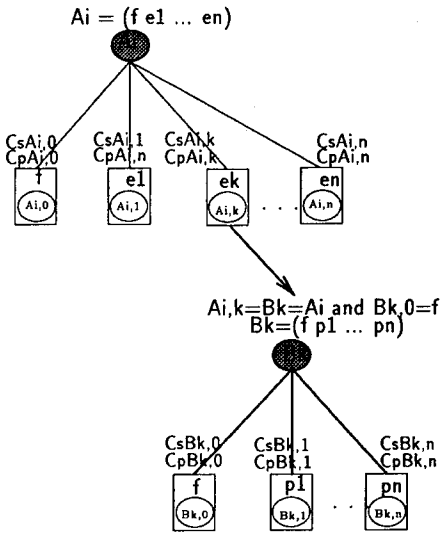


Figure 4: Cost calculations for a recursive function

3.4 Inherent parallelism

Conditional instructions parallelization is not recommended. On the other hand, in Graal there are some instructions which have an inherent parallel. They conform to Backus' original idea of changing sequential computation thinking[2]. The following instructions belong to this case:

- the application of a function to a sequence: α f <x1, ... xn>= <f:x1, ... f:xn>

  The distribution and reduction of a function application to every sequence element can be made in parallel. Nevertheless, the following condition should be satisfied: the number of processors or generated tasks should be equal to the number of sequence elements; and there should be also an equal number of function copies.

- The sequence construction: [f1, ..., fn]:x = < f1:x, ..., fn:x >

  The parallelization of this instruction is similar to the previous one. The only difference is that the number of processors or generated tasks should be equal to the number of functions and there should be an equal number of argument copies.

This kind of inherent parallelism is due to an element repetition, the function or the argument when constructing a new sequence. So, sub evaluations can be done in parallel by a task or processor.

The parallelism of these instructions has been shown by[10]. They have shown that a program in FP systems has regions of sequential and parallel execution. The aforementioned instructions have been defined as parallelizable from the outset, when the environment was first defined. This is achieved by defining their parallel cost as being less than their sequential cost. These instructions have been considered as exceptions to the given rule for primitive functions. The first two rules *I* and *II* given in figure 3, for the evaluation of forest costs are suitable for this case.

4 Conclusions

The automatic parallelization mechanism has shown that the scheme used can give significant results. It is certain that manual parallelization will give better results, because all details of a given configuration will be considered.

The parallelizer has been applied to different programs, and the initial environment has been constructed with rough estimations of primitive functions, for sequential and parallel costs. A good parallelization level has been found especially in instructions containing boolean operators. This shows a potential application to language programming for relational logic, implemented in Graal by [7]&[15]. Instructions considered as implicitly parallelizable have been found to be parallelizable after program transformation. This agrees completely with the initial idea given by Backus[2].

These preliminary results show that the aforementioned heuristic allows the automatic parallelization of programs. Nevertheless, it can be improved to take into account other factors like variable communications costs or scheduling overheads, this being a way of obtaining better levels of parallelization.

Another future research topic will be the extension of the results to a particular parallel architecture. The one chose is the ArMen architecture[11]. The basic node in this architecture is made up of a transputer, an LCA (Logic Cell Array) and a RAM memory.

References

[1] F. André and JL. Pazat. Le placement de tâches sur des architectures parallèles. *Techniques et Sciences Informatiques*, 7(4):385-401, 1988.

[2] J. Backus. Can programming be liberated from the von neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613-641, 1977.

[3] P. Bellot. *Sur les sentiers du Graal. Etude conception et réalisation d'un langage de Programmation sans variable.* PhD thesis, Université de Paris VI, 1986.

[4] F. Bieler. Partitioning programs into process. In *Lecture Notes in Computer Science*, volume 457, pages 513–524, 1990.

[5] D. Bourget. *Compilation et parallélisation d'un langage sans variable.* PhD thesis, Université de Paris VI, 1989.

[6] D. Bourget. Parallelisation of a language without variables, par_graal. *PPCC3*, 1989.

[7] D. Bourget and R. Legrand. Une machine de réduction d'un langage logique exploitant le parallélisme d'Ada. In S. Bourgault and M. Dibcbas, editors, *Programmation en Logique*, pages 441–458. FRANCE TELECOM and CNET, 1989.

[8] M. Burke, R. Cytron, and J. Ferrante. Automatic discovery of parallelism: a tool and an experiment (extended abstract). *Communications of the ACM*, pages 77–84, 1988.

[9] M. Castan, E. Cousin, and C. Coustet. *MaRS_Lisp. Manuel de l'utilisateur.* ONERA-CERT, 1990.

[10] N. Devesa, M. Lecoiffe, and B. Toursel. Un modèle formel d'exécution parallèle de programmes FP. *Bigre Programmes pour machines parallèles*, 66:163–174, 1989.

[11] J. M. Filloque, E. Gautrin, and B. Pottier. Efficient global computations on a processor network with programmable logic. In *Lecture Notes in Computer Science*, volume 505, pages 55–63. Parle, Springer Verlag, 1991.

[12] A. Gupta and A. Tucker. Exploiting variable grain parallelism at run time. *Communications of the ACM*, pages 212–221, 1988.

[13] S. Kalogeropulos. On partitioning lisp programs. In *High Performance and Parallel Computing in Lisp.* Europal Workshop, 1990.

[14] J. R. Larus and P. N. Hilfinger. Restructuring lisp programs for concurrent execution. *Communications of the ACM*, pages 100–1110, 1988.

[15] R. Legrand. *Calcul Relationnel et Programmation en Logique.* PhD thesis, Université de Paris VI, 1986.

[16] T. Lewis and B. Kruatrachue. Grain size determination for parallel processing. *IEEE software*, pages 23–32, 1988.

[17] C. McCreary and H. Hill. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32(9):1073–1078, 1989.

[18] R. Ortega. Mécanismes de parallélisation automatique pour un langage sans variable. Rapport du DEA Lap, 1991.

[19] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Conference on LISP and Functional Programming*, pages 202–211. ACM, 1986.

[20] H. P. Zima, H. J. Bast, and M. Gerdnt. Superb: a tool for semi-automatic mimd/simd parallelisation. *Parallel Computing*, 6:1–18, 1988.

SYNAPS/3 - AN EXTENSION OF C FOR SCIENTIFIC COMPUTATIONS

V.A.SEREBRIAKOV, A.N.BEZDUSHNY, C.G.BELOV

Computing Centre of the Russian Academy of Sciences
Moscow, 117967, Vavilova 40
e-mail:serebr@sms.ccas.msk.su
fax: (095) 135-61-59

Abstract

Extensions for conventional programming languages are suggested that allow to write parallel algorithms for distributed memory machines. The main goal of these extensions is to supply a programmer with a strict, precise and transparent language for scientific applications. Main language constructs are discussed with appropriate examples. Performance results for some applications are given.

1 Introduction

It is well known that programming for distributed memory computers is a very hard problem, especially for a nonprofessional (applied) programmer. We suggest a language (SYNAPS/3) that can be viewed as an extension for conventional programming languages (the first of all to FORTRAN and C), and is very simple, transparent, and natural for scientific programming where loops are the main control structure and vectors and matrices are the main data structure.

2 Problem overview

There are two extreme points of view on programming style for MIMD machines: the first assumes that the user must describe parallelization in all details, and the second is based on the idea that user can know nothing about parallel processing and all the work has to be done by the compiler. The most popular approach is to suggest the user to define a data distribution between processors and to leave the task of program parallelization for the compiler. The most well known projects in this direction are High Performance FORTRAN (HPF) and Vienna FORTRAN.

But the problem of data distribution is not the only one that have to be overcome. Another important problem is the usage of common (replicated) variables: either each processor must assign the same value to the variable on all iterations, so the program cannot be parallelized, or compiler can recognize, that the variable value is used only in

```
Sequential program          Vienna FORTRAN program

real A(N,N)                 real A(N,N) dist(*)
real temp                   real temp (N) dist(=A.2)
real ipivot                 real ipivot (N) dist(=A.2)
do 30 k=1,N-1               do 30 k=1,N-1
.....                       .....
temp=...                    temp(k)=...
 a)                          b)
```

Figure 1: Comparison of a sequential and Vienna FORTRAN program specifications

the current loop iteration, and partition its assignments(privatize). The HPF does not give rules that manage these assignments, because there is no a distinction between private and replicated variables in HPF.

Partially the notion of reduction overcomes this problem. But due to the absence of the explicit division between private and replicated variables it is impossible to define and program **reduce**-statement. Language offers only a fixed set of reduce operations.

As an example, let us consider LU-decomposition with pivoting. Do programs for sequential computers fit for parallel ones? In the Vienna FORTRAN language specification one of the examples of LU-decomposition ([5]*pp.*72) is given (Figure 1 b). Sequential form of the program is shown in Figure 1 a. It is easily to see that this is not the pure sequential variant, but it is tuned for parallel computers in a way absolutely not evident from the sequential point of view.

Instead of having scalar variable k array k is introduced to partition assignments of temporary value. This example shows that in most cases to get an appropriate result it is not sufficient simply to define data distribution, but it is necessary to do some more deep changes in the program.

All this leads to very diffused and vague language. The language is not now a strict tool to define a program, because it does not supply the programmer with understanding of the process of computations expressed in appropriate terms. To overcome this problem authors propose to introduce another, the second level of programming, the *environment* based upon text editor. This environment helps the user to understand what peculiarities of his program prevents parallelization. For example user can see data dependence graphs, sequence of calls etc. This is the typical case of double thinking: you formally consider the program as sequential, and the rules of the language does not supply you with some mechanisms to control the situation, but you cannot manage without this control, and you are forced to go out of these rules and attract another notions to understand the situation.

So, our purpose is to construct a language that must satisfy the following conditions:

- it must be a simple extension of conventional language (FORTRAN or C);
- it must not contain complex and unusual notions such as data communication, processes and so on;
- it must be strict in the sense that the language itself must be sufficient to express all notions that are needed; it should not be necessary to attract tools outside the language;

- it must supply the user with a possibility to manage the structure of the program and to control the quality of its program.

The only one principle difference that distinguish our language from HPF like languages is that the user must divide the program into two kinds of regions, the first can not contain assignments to replicated values that depend on its private and distributed variables, the second has no this restriction. Regions of the first kind we name *parallel*, regions of the second kind - *sequentional*. This requirement is sufficient to parallelize the program. Another result of this partitioning of the program body is that it makes possible to define **reduce**-statement precisely.

As a resume of the discussion we can say that we don't introduce any new concept in the language, but instead combine familiar concepts in a strict and precise mode.

SYNAPS/3 programs have following structure

```
  templ Tmpl[100:block];/* distribution scheme declaration */
  float A[Tmpl];        /* distributed array definition */
  float s;              /* replicated variable */
  par {float d; int i;/* private variables definition */
    for (;;;i in Tmpl)/* distribute iterations between processors*/
    { d=A[i];         /* Assignment to replicated variable s
                      (s=A[i]) the value that depends on private
                      or distributed variables is forbidden */
    }
    s=0;              /* assign the same value */
    reduce s+=d;      /* compute replicated variable value
                         on private data. */
}
```

2.1 Data distribution

Data that take part in parallel execution can be divided into two classes: *private* data that are declared in a parallel part and other ones that are declared in a sequential part, but are visible in a parallel part. The latter can be *distributed* and *non − distributed*. Distributed data are arrays for which some distribution rules are given. Other data are non-distributed. Private data are declared in a parallel block with usual rules.

From the **SPMD** point of view non-distributed variables are replicated variables and a parallel part differs from a sequential only in one restriction on use of replicated data: it is forbidden to assign them values that are dependent on private or distributed data.

Data distribution scheme is defined with *template* declaration **templ**. Data are distributed according with templates attached to them. For example,

```
templ t_a[100:block];
templ t_b[100:cyclic];
templ t_c[100:cyclic:10];
```

In this example data template *t_a* describes block distribution. Block size is determined by compiler. Distribution scheme may be defined as cyclic as for *t_b* template. Distribution scheme can be defined as block-cyclic as for *t_c* template. In this case blocks of data are distributed cyclically.

If templates are declared then distributed arrays can be defined in the following way:

```
float A[t_a];
float D[100][t_d];
float C[t_c][t_a];
```

Arrays are automatically dimentioned by being distributed. We name data that are declared as distributed *statically distributed* data.

3 Parallel control statements

3.1 Parallel block statement

par-statement allows to describe parallel execution of the same group of statements with distributed data. It is forbidden to assign to replicated variables values that depend on private or distributed variables. From the **SPMD** point of view **par**-statement serves to introduce private variables for processors and to limit assignments for replicated variables. To describe parallel execution it is necessary to point out the data and possible computations distribution. For this purpose the head of **par**-statement must contain templates according to which computations can be distributed in the par-statement and may define data that are dynamically distributed before the body of **par**-statement is executed and are collected after it. If **SPMD** model is used, data are not distributed before the statement, but it is necessary to "unify" data after the statement, i.e. to collect them on each processor.

If the head does not contain a list of dynamically distributed data then it has the form:

```
par (template_name,template_name...)
```

The list of templates in the head must contains all templates that define schemes of a processors network for parallel **for**-statements of this **par**-statement. Statement sequence of the statement body is executed with replicated data and its own private and distributed data.

The head can contain a list of *dynamically* distributed data:

```
par (t1, t2, t3; A[][t1], B[t2][t3])
```

The dynamically distributed variable must not be statically distributed. Sizes of each dimension must be equal to corresponded sizes of the template.

3.2 Parallel form of the for-statement

The list of statements is executed in a **for**-loop. This statement is intended to distribute loop iterations between processors. To do this the head of the loop contains template that defines this distribution. Data that are used inside the statement as distributed can be as statically, as dynamically distributed. In the latter case the distribution is determined by the surrounding **par**-statement.

In general case the head of the **for**-statement has the following form:

```
for (I=L;I<=U;I+=S;I in template_name)
    { statement list}
```

The range defined by the template is distributed between processors. On each processor the loop variable gets values from intersection of the range assigned to the processor and the range $L..U$ from the head. The bounds $L..U$ must lie in the range determined by the template. Loop variable must be private with respect to parallel block.

If the loop variable range is equal to the range determined by the template, the definition of bounds can be skipped:

```
for (;;;I in template_name)
    {statement list}
```

Example 1. *jki*-form of LU-decomposition with pivoting is given below. In this example data are distributed by columns.

```
templ tmpl[N:cyclic];
int ipivot[N]; float A[N][N];
main ()
{
 par (tmpl; A[][tmpl], ipivot[tmpl]  )
 { int j:
   for (;;;j in tmpl)/* distribute [0..N-1] between processors*/
   { int i,k, pivrow, tr[N];   float pivot;
     for (i=0; i<n; i++) tr[i]=i;
     if (j>0)
     {  for ( k=0; k<j; k++ )
        {   pivrow=tr[k];
            tr[ k       ] = tr[ipivot[k]];
            tr[ipivot[k]] = pivrow;
            for ( i=k+1; i<N; i+=1 )
              A[ tr[i] ][j] = A[ tr[i] ][j] - A[ tr[k] ][j] *
                              A[ tr[i] ][k] / A[ tr[k] ][k];
     } }
     if (j<N-1)
     {  pivot=0.0;           pivrow=-1;
        for (i=j; i<N; i++)
            if (abs(A[ tr[i] ][j]) > pivot)
            {   pivot=abs(A[ tr[i] ][j]);     pivrow=i;
            }
        if (pivrow>=0)      ipivot[j]=piv;
        else                printf("Cannot solve the system\n");
}} } }
```

3.3 Reduce statement

reduce-statement ensures synchronous computation of replicated data on private data. **reduce**-statement has the form:

```
reduce statement_sequence
```

The statement sequence executes sequentially in undefined order on every processor with its own private data. The statement body may contain assignments to replicated variables but not to private ones. We can compute the conjunction of private variable values in the following way:

```
int dis;
par (tmpl)
{ int flag;
  ....
  dis=0;
  reduce { dis |= flag; };
}
```

Example 2. Solving of linear system by Jakobi iterations :

```
templ  Tmpl  [N:block];
double A     [Tmpl][N];
double B[N], Y[N], X[N];
double Delta = 0.0001;  int j, dis;
do { for (j=0; j<N; j++) Y[j] = X[j];
     par (Tmpl; X[Tmpl] )
     { double s, Eps=0.0;  int i;
       for (;;; i in Tmpl)
       { s  = B[i];
         for (j=0; j<N; j+=1)  s -= A[i][j]*Y[j];
         s  /= A[i][i];
         Eps = fmax( Eps, fabs(X[i]-s) );
         X[i]= s;
       }
       dis=0;
       reduce dis |= (Eps > Delta);
     }
} while (dis);
```

Example 3. Fast Fourier Transform:

```
 enum    {      k = 13,   N = 8192 };
#define  M_PI   3.14159
 templ   Tmpl[N:block]; /* assumed that there exists */
 complex work_seq[Tmpl],/* the operations of complex type*/
         save[Tmpl];    /* to simplify program */
void main ()
{ int i, step, Power=1, Bit=1<<(k-1);
  par(Tmpl)
     for (;;; i in Tmpl) work_seq[i]= { (double)i/(double)N, 0. };

  for (step=0; step < k; step++, Power *= 2, Bit >>= 1)
  {
```

```
    par (Tmpl)
    {  int e_vals[k];       /*The values of the e(k,j)*/
       complex omega_power; /*The value of omega^e(r,j)*/
       for (;;;i in Tmpl) save[i] = work_seq[i];

       for (;;; i in Tmpl)
       { int p=i, j;    complex shift_value;
         e_vals[k-1]=0;
         for (j=0;j<k;j++,p>>=1) e_vals[k-1  ]=(e_vals[k-1])+p%2;
         for (j=1;j<k;j++     ) e_vals[k-1-j]=(e_vals[k-j]<<1)%N;

         omega_power = { cos(2.0*M_PI*e_vals[k-1-step]/N),
                         sin(2.0*M_PI*e_vals[k-1-step]/N) };
         shift_value = (i % (N/Power) >= (N / (2*Power)))
                  ? save[i - N/(2*Power)] : save[i + N/(2*Power)];
         if (i & Bit)
           work_seq[i] = save[i] * omega_power + shift_value;
         else
           work_seq[i] = save[i] + shift_value * omega_power;
       }
    }
  }
  for (;;; i in Tmpl)
  {    printf("Value %d, real      part =%g\n",i,work_seq[i].re);
       printf("Value %d, imaginary part =%g\n",i,work_seq[i].im);
} }
```

4 Subroutines

We impose some restrictions on the use of parallel constructions in subroutines. The main is the following: if a function can be called from a parallel loop it must not have parallel loops inside. Distributed array in this case can be passed into the function on the place of the following formal parameter:

```
type A[i in tmpl]
```

where *i* is the formal parameter on the place of which loop control variable have to be passed, and *tmpl* is the template according to which actual parameter and parallel loop are distributed.

If the function is not called from parallel loops, it can have parallel loops inside. In this case distributed arrays can be passed into the function on the place of formal parameters that are described as usual distributed variables:

```
type A[tmpl];
```

and these arrays can be used in parallel loops inside the function.

Size/Method	jkipb	ikjb	kjipb	kijb	kjipc	kijc	ikjc	jkipc
50	*0.18*	*0.19*	*0.46*	*0.55*	*0.43*	*0.55*	*0.62*	*0.60*
100	*0.18*	*0.20*	*0.61*	*0.69*	*0.52*	*0.86*	*0.92*	*0.87*
150	*0.18*	*0.20*	*0.66*	*0.74*	*0.78*	*0.98*	*0.98*	*0.97*
200	*0.18*	*0.20*	*0.68*	*0.75*	*0.84*	*1.00*	*0.99*	*0.99*
250	*0.18*	*0.20*	*0.69*	*0.76*	*0.88*	*1.00*	*0.99*	*1.00*
300	*0.18*	*0.20*	*0.70*	*0.76*	*0.91*	*1.00*	*1.00*	*1.00*

Table 1: The results of parallelizing LU-decomposition

lg_2Size	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
Without vectorization	*0.18*	*0.28*	*0.33*	*0.35*	*0.40*	*0.40*	*0.44*	*0.41*	*0.41*
With vectorization	*0.54*	*0.90*	*0.93*	*0.96*	*0.98*	*1.0*	*1.0*	*1.0*	*1.0*

Table 2: The results of parallelizing FFT

5 Implementation and Performance Results

The first version of the compiler from the SYNAPS/3 language with some restrictions on the form of nonlocal index expressions is implemented. So each index of left-hand side distributed variable of an assignment must be a loop variable.

Some performance results for different forms of LU-decomposition and FFT are given in Table 1 and Table 2. The tables show the ratio speedup/number of processors. In the table 1 **p**, **b** and **c** mean pivoting, block distribution, and cyclic distribution, correspondingly. The first row of the table 2 corresponds to the use of not vectorized form of send/receive, the second - to the vectorized one. This measurements were done on the transputer board IMS B008 [7][8] connected with IBM PC. The transputer board has six IMS T805 transputer nodes, each equipped with 1 or 2 Mbytes of local memory.

6 Related works

There were proposed some extensions of C language for distributed memory multiprocessors. For example in Kali[1] the user must specify data distributions and after that he must associate this data distribution with loop iterations distribution.

The language DINO[2] requires the user to specify a distribution of data to an environment. The programmer does not specify communication explicitly, but must mark nonlocal references.

The most distinctive systems for distributed memory multiprocessors are based on FORTRAN: Fortran-D[3], HPF[4],Vienna Fortran[5]. The main idea of all of them is to save FORTRAN for its users. In principal, only one construction is added to pure FORTRAN, namely data distribution. From this data distribution description compiler extracts information about the distribution of computations. Of course, this approach simplifies programming (especially for FORTRAN programmers), but it violates one of the main principle of programming: program structure has to be in accordance with program data structures. As a result a user can't see the difference between different variants of an algorithm. For example, there are a lot of scheme for LU-decomposition, but only few of them are well suitable for distributed memory computers. In this case

user have to draw some other tools to estimate his decision, for example a programming system that surrounds the compiler. In some cases such a system can suggest user a better decision. So, the construction of the program has at least two levels.

Another consequence of this disparity is the unnatural structure of an object program. To support this difference between a data structure and a structure of computations, a lot of statements must be guarded by conditions [6].

Essentially our approach differs from that mentioned earlier in the following:

1) The user must to point out explicitly what loops are distributed; on one hand it simplifies a compiler and makes the object program more efficient, on the other hand - it stimulates the user to make the program structure more appropriate for parallel execution;

2) It is forbidden to assign replicated variables values that depend on private and distributed variables inside **par**-statement. Because this it is possible to avoid problems connected to parallelization of loop bodies and again force the user to care of the structure of its program;

3) **reduce**-statement coupled with **par**-statement allow to overcome easily the problem of computations of reductions;

4) Dynamic data distribution is performed in structural way but not so freely as in dialects of FORTRAN; this allows to avoid dynamic redistribution.

References

[1] P.Mehrotra and J.Van Rosendale, Advances in Languages and Compilers for Parallel Processing. Cambridge, MA:Pitman/MIT Press, 1991, pp. 364-384.

[2] M.Rosing, R.W.Schnabel, and R.P.Weaver, Journal of Parallel and Distributed Computing, v.13, pp. 30-42 (1991).

[3] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. *FORTRAN D Language Specification.* Report COMP TR90-141, Rice University, Huston, Texas, April 1991.

[4] DRAFT, *High Performance FORTRAN Language Specification.* High Performance FORTRAN Forum, May 3, 1993, Version 1.0.

[5] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. *Vienna FORTRAN - a Language Specification*, ICASE Interium Report 21, ICASE NASA Langley Research Center, Hampton, Virginia 23665, March 1992.

[6] S. Hiranandani, K. Kennedy, C.-W. Tseng.*Compiling FORTRAN D for MIMD Distributed Memory Machines.* Communications of the ACM. August 1992, V. 35, No. 8.

[7] Inmos Ltd. Transputer Reference Manual. Prentice-Hall, 1988.

[8] Inmos Ltd. Transputer technical notes. Prentice-Hall, 1989.

The Meander Language and Programming Environment

Guido Wirtz
FB Elektrotechnik und Informatik, Universität–GHS–Siegen,
Hölderlinstraße 3, 57068 Siegen, GERMANY
guido@server.informatik.uni-siegen.de

Abstract

Explicit parallel programs are error-prone during coding and hard to understand. Besides the complexity of parallelism, these problems stem partly from the fact that entirely text-based languages are not the best choice for describing parallelism. The linear order of text hides the parallel structure of a program. Graphical constructs ease programming as well as understanding. We describe a hybrid programming language and the accompanied programming environment which integrates textual and graphical descriptions by clearly distinguishing sequential from parallel aspects. The Meander environment supports a wide range of programming steps and transforms hybrid programs into ANSI-C programs executable on transputer networks and workstation clusters. The major benefit is that program design, coding, mapping and visualization can be done in one single formalism.

Keywords: parallel programming environments, message–passing languages, graph–based specifications, visual programming, mapping support, visualization

1. Introduction and Overview

Text-based parallel languages have serious drawbacks which stem primarily from the fact that textual representations are always written down in some sequential order which hides the parallel structure of a program. To overcome this problem, we propose the usage of graphical languages to specify parallel programs. In order to manage the problem of graphical complexity and to broaden the acceptance of our approach, we do not use graphics for all parts of a parallel program: purely sequential parts should be formulated in the familiar textual manner. Hence, we use a *hybrid approach* integrating textual and graphical representations into one language. Such a hybrid language provides the ideal basis for an integrated programming environment: the same graph which is used for drawing a sketch of the planned process system is used for coding and for the visualization of program behaviour. Moreover, processor configuration, process-processor mapping, program execution and performance analysis can be specified and/or visualized using abstractions of graphical programs and hardware configurations. Due to the hybrid approach, the user is able to test sequential components in isolation as well as to re-use sequential programs on the familiar ANSI-C [1] function call level.

Meander is an explicit parallel language for the MIMD message–passing paradigm based on a CSP–like philosophy [2]. It is supported by an integrated programming environment including a graphical editor, static analysis of graph structures and C code, a graphical mapping component, automatic transformation of specifications into programs executable on transputer networks using Helios [3] and workstation clusters using PVM [4]. A visualization component permits the offline visualization of program runs on the graphical program representation.

The aim of this paper is to describe the different steps of support, the Meander system offers during program development. The underlying philosophy of parallel software development

is introduced more detailed in [5]; the principal sources for exploiting visual methods in a programming environment are discussed in [6]. Section 2 introduces the language by stepwise developing an example which illustrates the levels of notational support provided by the language. In section 3, the possibilities for specifying the desired target architecture, performing the process-processor-mapping and looking at the results of a program run in the visualization component are presented. We end up the description of `Meander` with a sketch of the remaining system components (section 4). Afterwards, some related projects are discussed and compared to `Meander` (section 5). We close with some remarks on the next development steps and future work.

2. Programming in `Meander`

For the sake of simplicity and in order to describe the basic language as well as some built-in abstraction mechanisms, we choose a simple Jacobi-like relaxation on an equidistant two-dimensional grid which may occur e.g. during the approximation of partial differential equations based on a finite-difference discretization as our running example [7]. The steps to a data-parallel version of that algorithm are quite simple:

1. The grid has to be set up with initial values w.r.t the wanted boundary conditions as well as the righthandside of the equation. This can be done by sequential `C` functions.
2. An iteration of the sequential algorithm can be formulated by a simple `C` function which takes a reference to the grid as its basic parameter. It behaves very locally by applying a small operator to all interior grid points based on values of a former iteration.
3. *Design of data structures suitable for parallelism*: parallelism is based on geometrically partitioning the first dimension of the grid into two-dimensional subgrids. In order to support communication among neighbouring subgrids w.r.t. overlapping read accesses, we provide interior subgrids also with an overlap region in this dimension. For the relaxation function this looks much the same as the handling of the boundary grid points: such points are only read but not written during the relaxation. In order to overcome problems with the handling of two-dimensional `C` arrays, we use a 1-dimensional work-array plus the required indices as our grid data `struct`.
4. *Assignment to logical processes*: initialization is performed by a `master` process prior to distributing the grid to a couple of `worker` processes. The same process is also responsible for sampling the grid afterwards and performing I/O. A single `worker` gets its subgrid, performs the desired iterations by exchanging overlapping values before each iteration and returns the final subgrid to the `master`.

The steps 1 and 2 are purely sequential and the only thing which is of further interest w.r.t. them is their *parallel decomposition* formulated in 3 and 4. The logical partitioning itself which has to compute equally-sized subgrids and provide the indices for all segments can also be formulated by sequential functions. We do not go into the details of the program code w.r.t. steps 1–3 because the methods used here are common for each data-parallel version of a relaxation. The latter step is performed by each process in combination with a logical process index in order to reduce the amount of exchanged data.

The methodology descibed so far reveals the advantage that the parts of the final program which implement the numerical algorithm as well as the logical data distribution can be developed and tested in a *pure sequential setting* without running the entire program in parallel. Moreover, these components may be substituted by different sequential routines conforming to

the same prototypes by seperate compilation and linking rather than changing, transforming and re-compiling the entire parallel program. Thus, our approach provides an interface for re-use, modification and sequential testing which fits well in the typical methods of developing C programs.

What is left, is the specification of the master and worker processes and the communication between them for exchanging data. This part is done in the context of Meander via a so-called *specification graph*. An entire *specification program* consists of three parts: the *specification graph* describes the global structure of a parallel process system by means of a *directed, loosely connected graph* build up from a fixed set of *graph fragments* and *3 disjoint types of edges* (causal, sync, async) representing the order of computation in the graph as well as synchronized and asynchronous communication connections. The correspondence to the sequential code parts is defined by an *annotation* function which provides an appropriate *sequential code fragment* for each node of the specification graph (executable statements, storage manipulation or expressions). In order to avoid the repetition of function implementations and typedefs, a third component – the *global base environment* – holds all parts of a specification which are not directly executable. This part is available at each node of the specification graph and hence may be used in each code annotation.

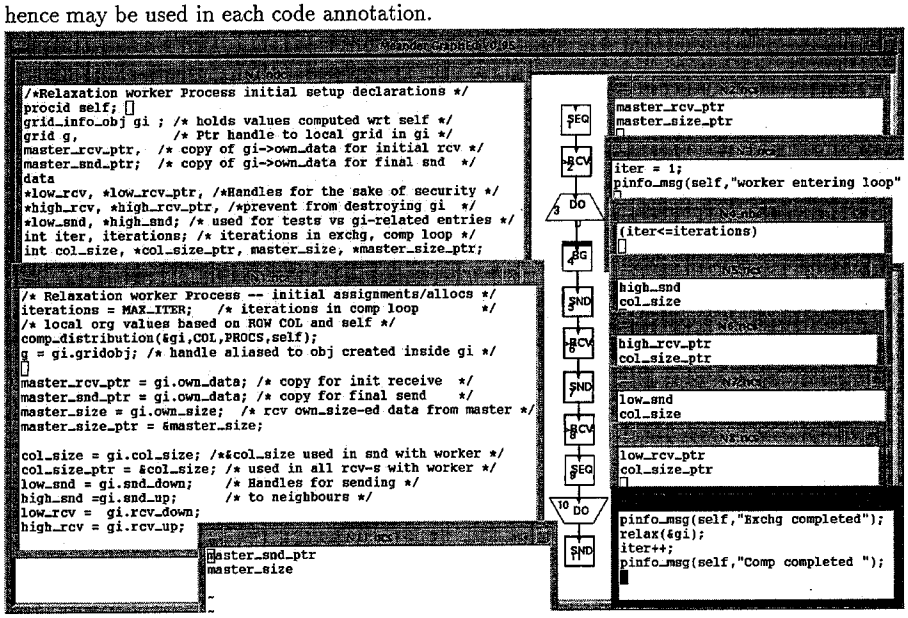


Figure 1. The graph of a worker process and its code annotations

Fig.1 shows the graph for a single worker process: node (1) which is a simple sequential node has annotated the needed declarations N1.ndc as well as the initialization of all data in its executable part N1.nca. The design of the grid data structure is hidden in the globally available base environment as well as the implementation of the comp_distribution function computing the overall grid partitioning because this information is used later on in many processes. After this initialization, the receive node (2) provides the interface for receiving data from another process (the master as shown in Fig.2). A rcv as well as the complementary send node are

annotated by a *reference* to the data which are to be sent (where data are to be received) and the `sizeof` the message. The `rcv` may only take place afterwards iff a corresponding process is ready to execute a matching `snd` and vice versa in the case of synchronous communication. Until this time, the process executing its construct first is blocked.

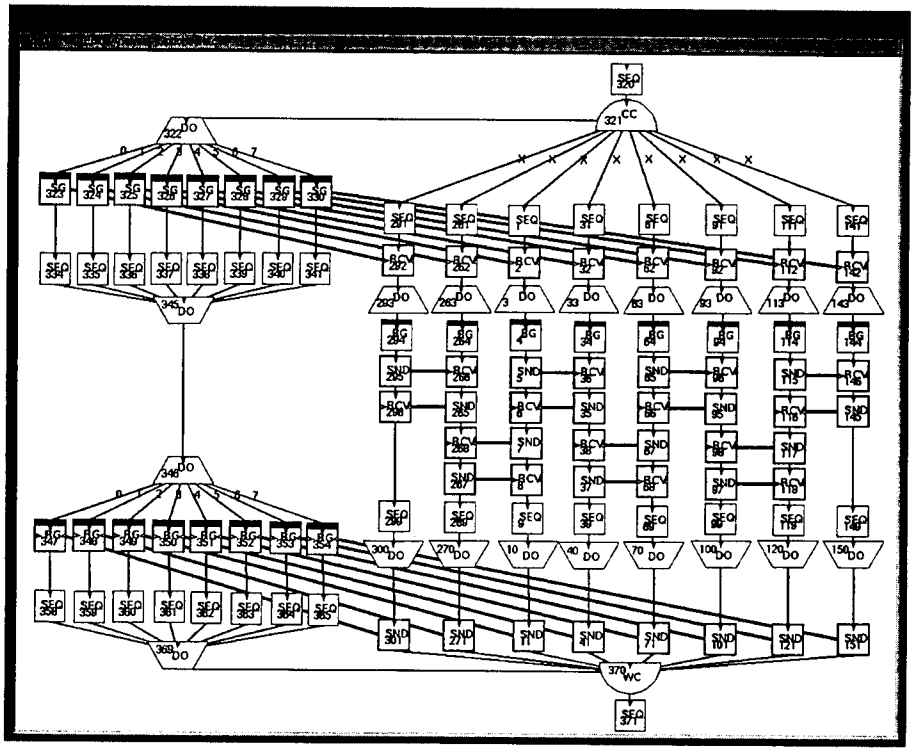


Figure 2. The specification graph for a `master` and 8 `worker`

After receiving its own chunk of data, the `worker` starts a `do` (3) which initializes the iteration count `iter` and is then controlled by a simple, single `bool guard` (4) holding a boolean expression in order to execute a fixed set of iterations. The semantics of this `do` resembles a while loop which means executing the annotated code of nodes (5)-(9) until the condition in (4) becomes false. The loop is not formulated by textual code because it contains further communication and hence has influence on the parallel behaviour of the process. This is the global rule for deciding which parts of a `Meander` program are formulated graphically and which parts are annotated: *sequential code must not hold communication or other parallel activities.*

The communication inside the `do` is used for exchanging in each iteration the overlapping data with its upper and lower neighbour before performing the next relaxation (9). The process (1)-(11) is a typical interior `worker` which has a neighouring process at the right (31)-(41) as well as at the left (261)-(271). The connectivity of communication nodes is represented explicitly by means of special `sync` communication edges. The two processes at the boundary

are somewhat special because they have exactly one neighbour and their communication part is reduced to a single exchange. Note, that there are two kinds of processes in order to avoid deadlocks during the communication which synchronizes both partners. After performing all iterations, the do of a **worker** terminates by executing the end-do node (10) and the resulting set of local data is send back to the **master** (11). Afterwards, the **worker** process terminates automatically because its chain of activities has been executed completely.

The entire example process system consists of a **master** (320)-(371) and 8 **worker** processes. The **master** performs the work of our step 1 in node (320) and then executes a **create-child** construct (321) which starts all processes which are connected to this node by means of an X-marked process creation edge. The corresponding **wait-child** (370) is used to synchronize the **master** with all **workers** because this node can only be executed iff all processes started at the cc have been terminated. The notion of *process* is also defined on the basis of these constructs: each maximal chain of nodes built up by *causal* edges constitutes a process of its own. Thus, a process starts either at a node which has no causal predecessor (the **master** at (320)) or at a node with an incoming process creation edge. The code of the **master** annotated to the nodes between these brackets is executed in parallel with all **worker** processes. That means, the termination information of all started processes is internally available at the **master** when the **wait-child** node (370) is executed and leads to the completion of this construct and after executing the last **seq** (371) (performing I/O of the result) to the termination of the **master** and the entire system.

The only thing which is of further interest in the **master**, is the structure of the do loops which are used for distributing (322)-(345) and sampling (346)-(369) the subgrids to/from the **workers**. Here, the power of the CSP-like [2] *guarded commands* is shown better than in the simple **worker** loop: each do consists of 8 **guards** which are used to send/receive the data to/from the **workers**. Using communication constructs as conditions is highly useful because the loops have no entirely predefined order of execution: there is a *priority* assigned to each edge leading to a guard (here: 0 upto 7) which *rules the order of inspecting* the guards. Nevertheless, if in the case of the sampling loop, an inspected **rcv** is not yet ready to execute because the **worker** holding the matching **snd** has not yet finished its relaxation loop, the next guard is examined until all have been executed once in this case. Hence, always that order of sending/receiving is choosen which implies the shortest waiting times for the **master**.

Note, that the example is set up in a way that all data which are used in more than one process have to be exchanged explicitly via **snd/rcv**. There is *no data sharing between distinct processes* and we work in a pure distributed memory paradigm.

It should be clear that the explicit formulation of each **worker** process is not the best way to describe a data-parallel program consisting of many similiar processes and we have used this approach only to illustrate as many features as possible within a single example. Normally, such a system is formulated by means of an abstraction mechanism which is called *process replication*. Fig.3 (left) shows a **master** process as well as a single **worker** *template process* for a relaxation. However, there are additional annotations which describe the intended process system: the **create-child** (31) is annotated by **grid(p,8)** which specifies that 8 processes of the templates kind have to be created. The *internal communication structure* between the replicated processes is also specified by the **grid** keyword which leads to the structure shown in the upper right part of Fig.3 including the correct setup of the boundary processes. Additional supported replication schemes are **torus** (connecting also the boundary processes), **pipe** (only one-way connections), **tree** etc. as well as higher dimensions for **grid** and **torus**. Note, that

we use asynchronous inter-worker communication this time (represented via dashed lines) and need only one kind of worker template process.

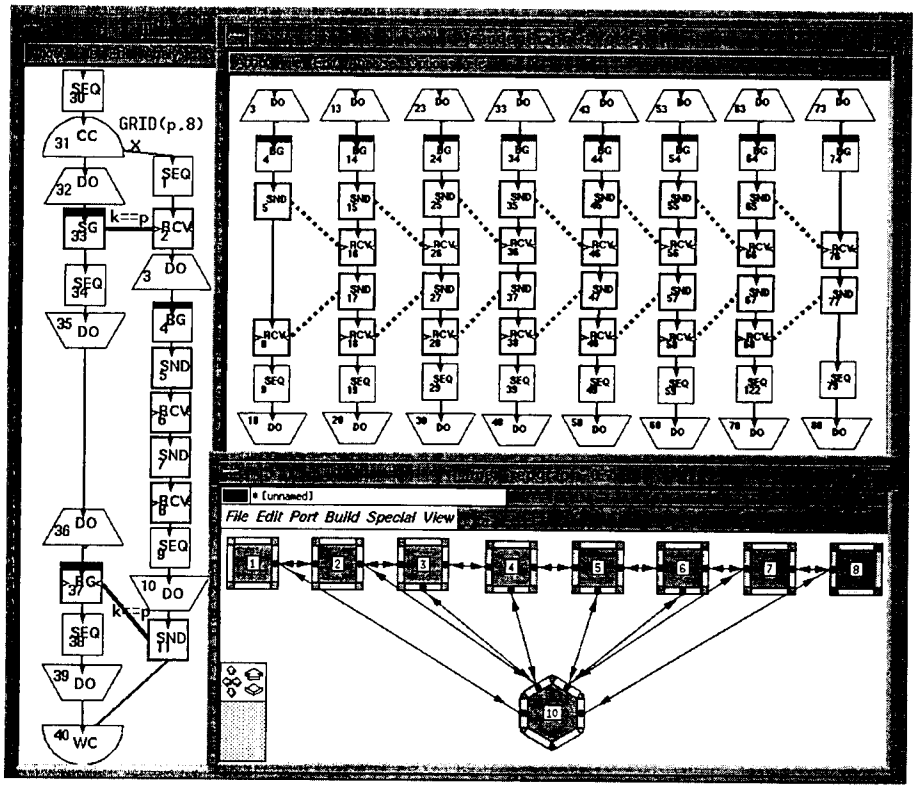


Figure 3. Relaxation template and replication graph

The last parameter p of the replication specification introduces a *meta-variable* which is helpful for specifying the communication between the master and the replicated processes. It is used in the distributing (32)-(35) and sampling (36)-(39) loops in order to get rid of specifying a guard for each replicated process. Instead, one single guard is used and the connecting communication edge, e.g. (33-2), is annotated by a C expression which identifies the subgrid count k in the masters loop with the number of the replicated processes p.

3. Hardware Specification, Mapping and Visualization

After coding the program, some additional steps are needed to run the program in parallel. The most important step is deciding for the *target hardware* for a run and *mapping* the processes to the available processors. Support for this step in Meander is based on two graphs. The *process graph* is an automatically obtained abstraction of the specification graph which is computed by collapsing all nodes which make up a single process to a single *process node*. The communication edges of the specification graph are inherited by the process nodes and

collapsed, too. In our examples, this results in the graph shown in the right lower window of Fig.3. It represents the overall resulting processes as well as the needed communication connections which would result from the replicated process system as well as the explicit formulated one shown in Fig.2. The target of the mapping is the so-called *attributed hardware graph* which specifies the available computing nodes (transputers and/or workstations) and their interconnection structures. An example of a configuration consisting of a frontend workstation (13) and 12 transputer nodes is shown in Fig.4. The graph is obtained automatically via the `Helios resource map` [3] when working with the transputer backend. In the case of `PVM` [4], the user has to specify it explicitly in order to describe which workstations should be used in a program run.

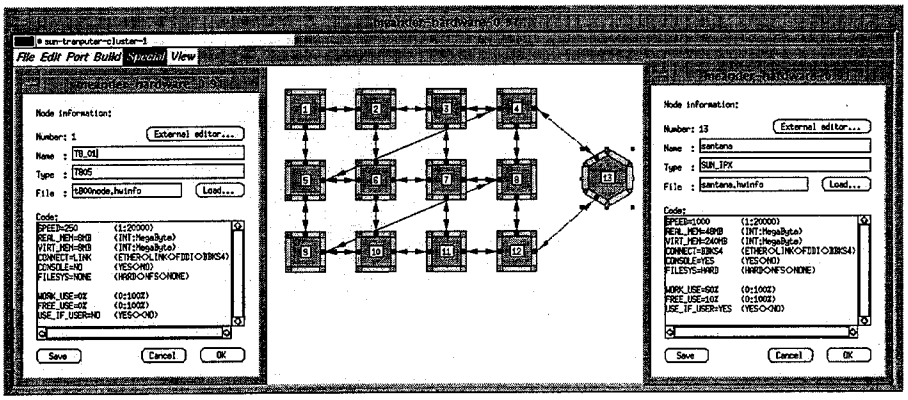


Figure 4. Hardware configuration as mapping basis

Each node of the graph holds *attributes* which are used to describe the characteristics of the corresponding processor in terms of relative speed, memory etc. Typical entries for a transputer node (1) and the frontend (13) are shown in the extra windows. This information is used as the basis of the mapping which can done either automatically or manually by assoziating (colouring or numbering) the nodes of the process and hardware graph. Part of the current work w.r.t. `Meander` is due to developing new mapping strategies based on genetic algorithms. It should be noted that the same attribute mechanism and supporting tool is also used to specify and/or display the *requirements* of the process graph. For example, the user may annotate the maximum memory requirement of a process in order to avoid its mapping to a transputer if it exceeds the 8MB limit present in the configuration shown in Fig.4. Moreover, this method is used to display results from previous program runs on the same hardware in order to provide information for performance tuning.

Besides the usage of statistics from previous runs, we support the offline visualization of the run based on tracefiles: if the user annotates the specification graph by visualization attributes, function-calls to our `visualization library` are generated during the transformation phase. The program run produces a tracefile which is used in the graphical editor to animate the program execution on the basis of the specification graph and/or the mapping graph. Visualization is controlled by the user through temporal and logical stepping using a recorder-like interface. A typical situation of a run of the program from Fig.2 is shown in Fig.5. The state of the visualization is represented by the colour of the nodes: white nodes

are not yet executed, grey nodes have been executed at least once and the dark grey nodes are the currently active nodes. The situation shown in Fig.5 occurs after distributing all data during an iteration of the `worker` loop.

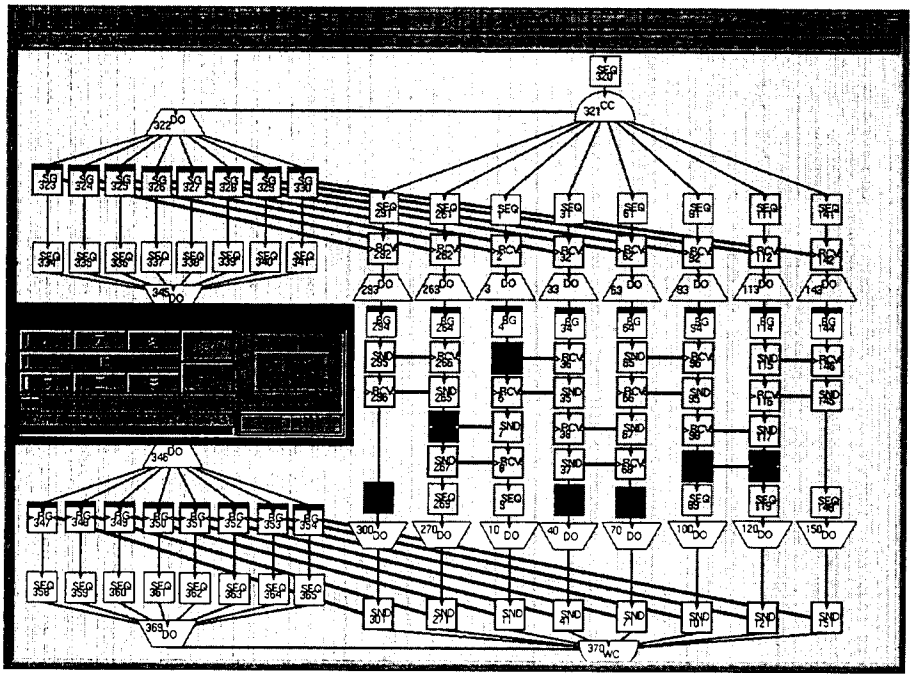


Figure 5. Visualization of the relaxation process system

4. The Remaining Components of the Meander System

The central component is a X-based *graphical editor* running on DecStations (C++/InterViews-3.1 [8]) which supports the interactive construction of specification graphs, the annotation of sequential code to nodes and the analysis of graphs (cf. the screenshots in Fig.1–3). An incremental *analysis component* checks sequential C code annotations. We use the lcc [9] C-frontend equipped with functions to generate an annotated syntax tree as starting point of our analysis. The *transformation component* generates C source code for a main program for each specified process by combining code annotations with template code implementing the graphical constructs. These can be compiled on two backends: a SUN-hosted transputer system running Helios [3] or workstations using PVM-3.2.x [4].
Code generation is supported by the *Meander library*. It provides the units to organize process systems as well as the communication services which are not directly present in the used target software architecture. Realizing the library has been a challenge for the transputer backend where it is implemented on top of the socket level of Helios. Besides its use as Meander–transputer backend, the library is also useful as a Helios communication library of its own [10]. In the case of PVM, most of the needed functionality was already available.

5. Related Work

A lot of tools support some steps in the development of explicit parallel programs (for a recent overview for network-based systems cf. [11]). The area of *visualizing* the concrete race of a program has been tackled in many approaches, e.g. [12] or [13]. In contrast to **Meander**, almost all approaches start with the textual coded parallel program, give no direct support for the core program development and share one common drawback: the representation used to develop a program is completely different from those of the tools which are needed to understand the program. Moreover, most of these tools put their focus on performance measurement, not on a better program understanding.
In the area of *visual programming* some attention has been spent on parallel programming for years (c.f. [14]). Some of the graphical formalisms used are close to that of **Meander**.
The **Schedule** environment [15] is explicitly dedicated to the development of large scale numerical programs but restricted to shared-memory machines and based on **Fortran**. For the Petri-net based work of [16], the focus lies in performance prediction.
The basic atomic entities of many graphical approaches are coarse-grained entities like processes and the focus lies on support for defining process and processor configurations and mapping (e.g. **GRACIA** [17]). The **PVM**-based [4] **HeNCE** tool [18] offers a graphical interface for describing coarse-grained parallel **PVM** tasks. Nodes are connected via edges describing dependencies between the data produced in the nodes and contain subroutine calls as well as input/output declarations in order to specify which data are to be imported/exported in a single node. Graphical patterns for defining conditionals, loops and pipes ease the specification of process systems. The **Meander** language seems to be more appropriate for describing parallel programs which make use of explicit parallel components. In **HeNCE**, the incorporation of graphical constructs, node code and function calls works on three distinct language levels whereas **Meander** combines the textual and graphical level in a more concise manner.
Built-in graphical patterns for typical parallel process structures are wide-spread used through almost all approaches mentioned so far. In the P^3L language [19], pattern constructors for pipes, recursion, farms etc are utilized on the more detailed level of source code rather than to combine entire processes. In the **ADL** data-flow language [20], special nodes like communication channels and semaphores are introduced in order to combine basic sequential activity-nodes. Standardized graphical patterns for common parallel situations are very useful. Relying solely on such structures, however, may become a problem when a programmer has to formulate a non-standard communication pattern. A powerful general language supported by a set of built-in patterns seems to be the better choice.

6. Conclusions

The **Meander** system as described in this paper is functioning as a prototype. However, some of the components are not as comfortable as they should be. Enhancements based on the prototype experience currently under way are due to the better integration of the different components, a more powerful description of process and hardware graphs and better algorithms for the automatic mapping component (cf. section 3).
The focus of future work lies in implementing a third backend as soon as a stable multi-platform library conforming to the recent *message passing interface* standard is available. Conceptual work is especially due to the handling of data which is purely textual at the moment. Incorporating visual mechanisms for data distribution specification as well as their visualization will improve the applicability of **Meander** to a great extent.

REFERENCES

1. ANSI. *American National Standards for Information Systems, Programming Language C*, volume X3.159-1989 of *ANSI*. ANSI, New York, 1990.
2. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, N.J., USA, 1985.
3. Software Ltd. Perihelion. *The Helios Operating System*. Prentice-Hall, N.J., USA, 1989.
4. V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.
5. G. Wirtz. Graph-Based Software Construction for Explicit Parallel Message-Passing Programs. *Information and Software Technology*, 36(5), August 1994.
6. G. Wirtz. A Visual Approach for Developing, Understanding and Analyzing Parallel Programs. In E.P. Glinert, editor, *Proc. Int. Symp. on Visual Programming, Bergen, Norway*, pages 261–266. IEEE, August 1993.
7. W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T Vetterling. *Numerical Recipes - The Art of Scientific Programming*. Cambridge University Press, Cambridge, New York, 1986.
8. M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1990.
9. C. W. Frase and D. R. Hanson. A Code Generation Interface for ANSI-C. *Software—Practice & Experience*, 21(9):963–988, September 1991.
10. P. Böckmann, H. Giese, and G. Wirtz. Providing CSP-like functionality in a Helios^{TM} environment. In *Proc. 17th WoTUG Tech. Meeting, Bristol, UK*. IOS Press, April 1994.
11. Louis H. Turcott. A survey of software environments for exploiting networked computing resources. Technical Report MSU-EIRS-ERC-93-2, Mississippi State U., Starkville, MS, February 1993.
12. M. Heath. Visual animation of parallel algorithms for matrix computations. In D. Walker and Q. Stout, editors, *Proc. Fifth Distributed Memory Conference*, pages 1213–1222. IEEE, April 1990.
13. Thomas Bemmerl and Peter Braun. Visualization of message passing parallel programs. In Bouge J. et al., editor, *CONPAR92, LNCS 634 , Lyon, France*, pages 79-90, Sept. 1992.
14. Ephraim P. Glinert. *Visual Programming Environments - Paradigms and Systems*. IEEE Society Press, Los Alamitos, CA, 1990.
15. J. Dongarra, D. Sorensen, and O. Brewer. Tools to aid in the design, implementation, and understanding of algorithms for parallel processors. In R. H. Perrot, editor, *Software for Parallel Computers*, pages 195–220. UNICOM Applied Information Technology, Jan 1992.
16. A. Ferscha. A petri net approach for performance oriented parallel program design. *Journal of Parallel and Distributed Computing*, 15(4):188–206, August 1992.
17. O. Krämer-Fuhrmann and Th. Brandes. Gracia - a software environment for graphical specification, automatic configuration and animation of parallel programs. In *Transputer Anwender Treffen TAT , Aachen, Germany*, September 1991.
18. A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proc. of Supercomputing 91, Albuquerque, NM*, pages 435–444, August 1991.
19. M. Danelutto, S. Pelagatti, and M Vanneschi. High Level Languages for Easy Massively Parallel Programming. *Computer and Information Science*, 6(3), Oktober 1991.
20. M. van Steen, T. Vogel, and A. ten Dam. ADL: A Graphical Design Language for Real-Time Parallel Applications. In *Proc. WoTUG 93, Aachen, Germany*, September 1993.

Data-flow language for MIMD distributed memory multiprocessors

Jacques JULLIAND and Béatrice MARKHOFF
University of Franche-Comte, Laboratoire d'Informatique, 25 030 Besançon Cedex, France
Tel : 81.66.64.51, Fax : 81.66.64.50 - Email : julliand@univ-fcomte.fr

Abstract. We present a data-flow language for communicating process networks programming. We define a small step reduction semantics that corresponds closely to our operational intuitions about communicating process networks. Since it is able to describe infinite streams of data, function applications are evaluated in a lazy way while applications annotated with the symbol ‖ are evaluated in an angelic '*parallel*' way ; that induces a weak non-determinism. We do this by assigning to each branch a finite amount of evaluation resources, allocated by a fair scheduler. We demonstrate that our parallel operational semantics is sound and fair in regards of a lazy one. It is sound and fair in the following sense : any parallel applicative expression which is reduced by '*call-by-value*' can be reduced by our '*parallel*' strategy for any allocation. This operational semantics is asynchronous and parallel instead of lazy and sequential as in LUCID or synchronous and parallel as in LUSTRE. It modelizes that each branch of a parallel expression is associated to a communicating interpreter. This method avoids the main drawbacks of parallel implementation of usual functional languages : automatic detection of too small parallel tasks and evaluation of data that do not satisfy the property of locality.

Keywords. Data-flow language, Parallel programming, Functional programming, Parallel operational semantics, Distributed computation, Declarative programming.

1. Introduction

The present paper investigates an applicative programming for MIMD architectures. The aim is to define a high level programming model that leads to good qualities of programming such as reliability, better productivity, portability, ...and allows us to develop applications without execution model details. We agree with BACKUS [2] who thinks that the functional programming paradigm is a good one to reach this goal. But the good properties of the high level programming model do not undermine neither the efficiency of implementation, nor the expression power.

There exists two functional approaches for parallel architectures programming :

- to use classical functional languages which favours implementations of the "natural" parallelism,
- to extend them with parallel expressions.

The first approach [6, 17, 23] defines an application strategy of reduction rules in such a way that many processes can reduce independant subexpressions. But it clashes on two drawbacks which decrease the efficiency : automatic detection of too small parallel tasks and evaluation of data that do not satisfy the locality property. Moreover, these languages do not reach the expression power of parallel imperative languages such as C.S.P [11].

In the second approach, we distinguish three ways to extend functional languages :

- to add annotations so as to lead towards parallel implementations,
- to add process, message passing by rendez-vous and non determinism notions,

• to introduce time in language.

Annotations do not modify the host language semantics. They describe task partitions and/or data allocations. The language CONCURRENT-CLEAN [22], based on MIRANDA [26], adds annotations that exhibit parallel tasks to settle their size. CALIBAN [16] describes static communicating process networks, every process computing a function on streams.

PFL [12] and LCS [3] are based on CCS [19] and ML [18]. These extensions modify the semantics of ML deeply : synchronization, rendez-vous and non-determinism are introduced. Unfortunately these extensions put together the imperative and functional paradigms and induce complexity (loss of the unfolding and referential transparency properties [24]) and bad lisibility of programs.

RUTH [9] and ARCTIC [5] are based on timestamped streams which preserve the unfolding and referential transparency properties in contrast with a nondeterministic operator. But, time expression represents a very detailed execution scheduling. This one is difficult to implement and it is probably superflous outside the real-time domain, for which such languages are designed.

In this paper we focus attention on the pragmatic aspects to define a parallel operational semantics for an applicative language whose expressions describe communicating process networks. To do so, we need a semantics that reflects the parallel evaluation we make in practice. We assume that the reader is acquainted with operational definitions of functional languages and with reduction techniques.

The plan of paper and the main points to notice are as follows. Our goal is to investigate a parallel operator in a data-flow language to define a sound operational semantics that reflects parallel evaluation of communicating processes. Section 2 introduces and justifies the main foundations of our programming model in an informal way, defines a small applicative language and illustrates it by some examples. Section 3 defines its lazy operational semantics. Section 4 defines a parallel operational semantics that we prove sound and fair in regards of the lazy one. In section 5, we conclude by interests and limits of such proposals.

2. A small parallel and applicative language

In this section we define a very simple applicative language called AP^{2}L (for Asynchronous Parallel Programming Language). Section 2.1 presents it in an informal way, section 2.2 defines its syntax and section 2.3 gives some examples.

2.1. Informal presentation

In this section we introduce a number of concepts and distinctions related to parallel applicative programming. This is done in a rather informal way, but many of the concepts are treated in greater depth in subsequent sections.

The functional programming model proposed below is based on the following principles :

• expressions describe streams of values as in LUCID [1] and LUSTRE [8],

• the proposed language is a first order one,

• it contains a parallel annotation with an actual parallel operational semantics.

The stream concept allows us to express communicating processes as functions in a data-flow way. Intuitively, functions describe iterative processes that compute stepwise an output stream from many input streams. AP^{2}L is a first order language so that stream items are values and never functions. This limit does not undermine our goal and gives a domain where the semantics are simpler, specially for the parallel operator.

The parallel annotation allows us not to confuse the functional decomposition and the process structure. This operator is an annotation that reaches two goals : first it allows to settle the grain of parallel tasks and second it leads to define an operational semantics related to the communicating processes execution model. Since denotational semantics of functions is no strict , this is an annotation related to the applicative order called '*call-by-value*'. The syntactical distinction allows us to describe the same expressions in two ways : *x(e)* or *x ‖ (e)* where *x* identifies a function. In the latter, every expressions *x* and *e* are two processes in an operational point of view. In such a way, the programmer settles the task grain. In section 4, we define different operational semantics for these two kinds of expressions : expressions *x(e)* are lazily evaluated with a normal order called '*call-by-name*', whereas expressions *x ‖ (e)* are evaluated according to an applicative order usually called '*call-by-value*' that is computed by slice of streams called '*call-by-slice-of-value*'. In this case, expression *e* defines a producer process of a stream and *x* a consumer of that stream. This is an asynchronous strategy in the following sense : producer and consumer can compute simultaneously or concurrently without global scheduling at each step. This means that at any moment of the evaluation, the expression *e* defines the output stream of a producer process whose a prefix called *pe* is computed. *x ‖ (e)* defines the output stream of the consumer process *x* (which is a function written $\lambda y.e'$) applied to *e*. A prefix of *x ‖ (e)* is computed from a subprefix of *pe* which has been already substituted in place of the formal parameter *y* in $\lambda y.e'$. In other words the reduction of each expression can be computed by communicating, asynchronous and parallel interpreters. The subprefix of *pe* substituted in *e'* represents the communicated values, the remaining values in *e'* are the unused values and the remaining values of *pe* in *e* are the computed and uncommunicated values. These substreams represent *fifo* buffers such as in a KAHN's networks of functions [15].

2.2. Syntax

The syntax of AP²L is given in definition 2.1 and an informal semantic description is given in the annotations following. A program is an expression. In this paper, values are only integer and boolean streams to simplify the definitions.

Definition 2.1 The syntax of AP²L is :

1, 2. <*Expr*> ::= <*Constant*> | <*Var*> — constants and variables
3, 4. | <*Var*> (<*Expr*>⁺) | <*Var*> ‖ (<*Expr*>⁺) — function applications and annotated function applications
5. | <*Expr*> + <*Expr*> — arith. and log. expr.
6. | <*Expr*> : <*Expr*> — stream initialisations
7. | *tl* (<*Expr*>) — tail of a stream
8. | *let* <*Decl*>⁺ *in* <*Expr*> — expr. in an environment
9. | *if* <*Expr*> *then* <*Expr*> *else* <*Expr*>
10. <*Decl*> ::= *[rec]* <*Var*> (<*Var*⁺>) = <*Expr*> — declaration of functions
11. | *[rec]* <*Var*> = <*Expr*> — decl. of var. ■

Annotations to definition 2.1

1. The integer constants are drawn from the set *int* ..., -1, 0, 1, ... They denote constant streams.
2. The variables are identifiers that denote streams of integers or functions.
3. The functions, declared as in definition 10, can be applied to parameters that are any expressions. They are applied on a *n*-tuple of at least one actual argument.
4. If applying functions is annotated with the symbol ‖, then each expression is a process.
5. The usual arithmetic (for example +) and logic operators define a stream by pointwise applying.

They are strict in both arguments.

6. *x:y* denotes a stream whose first element is the *x*'s followed by the stream *y*.
7. *tl(x)* denotes the stream *x* without its first element.
8. The stream denoted by this expression is the one denoted by the expression in part *in*. It can use expressions associated with variables in part *let*.
9. The conditional is strict in its first argument.
10, 11. The functions and variables may be recursive. Variables are special cases of functions. They denote streams of values whereas functions denote mappings of *n*-tuple streams into a stream. Mutually recursive declarations of variables must be annotated with *rec*. For example *rec x=f(y) ; rec y=g(x)* is correct whereas *x=f(y) ; y=g(x)* would be not. ■

Several static checks can be performed on programs : for example we check that expressions are well-typed. But this static semantic aspects are not presented in this paper.

Definition 2.2 The syntax of AP²L contains three supplementary expressions (not allowed to programmers, but obtained by reduction) :

12. <*Expr*> ::= <*Expr*> ‖ (<*Expr*>⁺) — ‖ expr.
13. | λ <*Var*>. <*Expr*> — functions
14. | **Y** <*Var*>. <*Expr*> — recursive variable ■

Annotations to definition 2.2

12. They are established by unfolding the definition of <*Var*> in expressions <*Var*> ‖ (<*Expr*>⁺).
13. For reduction, the lambda expressions denote functions that are useful in the left part of expr. 12.
14. These expressions are recursive processes or networks. They are introduced in order to compute feedback in networks by communicating from output to input instead of unfolding. ■

2.3 Some examples

2.3.1. Processes

The network shown in fig. 1 is given by MISRA in its paper entitled *equational reasoning about nondeterministic processes* [20]. *Dfm* is a non deterministic process that computes a fair merge of two streams as in functional operating systems [7, 10]. One can not describe such a process with the deterministic syntax of AP²L presented in definition 2.1 (to do this see a nondeterministic definition in [14]). The process called *P* has the following behaviour : it sends *0* and then indefinitly receives an integer value *n* and sends the integer value *2n*. *Q* is a process that indefinitly receives an integer value *m* and sends the integer value *2m + 1*. They are expressed as :

- $P(d) = 0 : (2 \times d)$
- $Q(d) = 2 \times d + 1$

2.3.2. Networks

- Assume that the process *dfm* is predefined, the network in fig. 1 is described as :
  let $P(d) = 0 : (2 \times d) ; Q(d) = 2 \times d + 1 ;$ *rec* $d = dfm \parallel (P(d), Q(d))$ *in* d.
- *let rec* $nat = 0:(nat+1) ; s = Q \parallel (nat)$ *in* s computes the stream of natural odd integers.
- *let rec* $s = Q \parallel P(s)$ *in* s is a network with feedback that computes the stream $1 : ((4 \times s)+1)$.

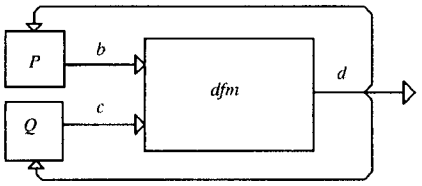


figure 1 : network with three processes

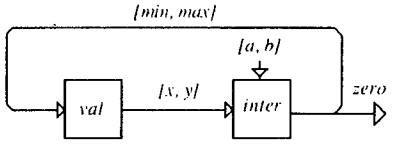


figure 2 : root searching

2.3.3. Function root

Let f be a real continuous function defined on the closed interval $[a, b]$ and assume that $f(a) \times f(b) \leq 0$, i.e. f has at least one root in $[a, b]$. The problem is to design an algorithm to find a root of f.

First, let us specify the terms of the problem :

data a, b : point — the interval bounds and the result

f : point → real ; ε : real — the given function and the required precision

result x : point — the point that represents the root

definitions

x such that $\exists$ *min, max* : point : $(a \leq min \leq x < max \leq b \wedge f(min) \times f(max) \leq 0 \wedge max\text{-}min < \varepsilon)$.

■

In [21] many solutions are proposed. The first is based on the idea to define a sequence of intervals $[min, max]$ whose size is decreasing. A sequence of points x is chosen partitionning each interval $]min, max[$. This solution is illustrated by fig. 2. The process called *inter* computes a new interval $[min, max]$ from the previous one, its middle x and the value y that is f applied to x. The process called *val* computes x and y. These two processes are connected in a cycle as shown in fig. 2. Assume that the notation $[a, b]$ describes a stream of pairs, this recursive network is expressed as :

```
let   val([min, max])=[(max+min)/2, f((max+min)/2)];   — value of f in the middle of the given interval
  inter([min, max], [x, y]) =
  if y×f(min) > 0 then [x, max] else [min, x] ;
  rec zero = [a, b] : inter ‖ (zero, val(zero))
in zero
```

Notice that the processes *val* and *inter* omit the termination aspects. Therefore they do not use the data ε. The result is any bound of the first value *zero* such that the interval is less than ε.

3. Lazy operational semantics

First, we define a '*call-by-name*' evaluation for the above language without the parallel annotation.

Second, to simplify our presentation, we assume that functions have only one argument. Moreover, we consider AP²L without conditional in order to restrict the semantic domain to integers. Care has been taken to make the language AP²L as simple as possible, while still interesting. Simple as it is, we feel justified in using AP²L as basis for our discussion, because it contains features normally found in data-flow languages, albeit only those necessary for the present treatment. Section 3.1 defines some notions needed in the following, and section 3.2 defines the lazy operational semantics.

3.1. Environment and values

Since we limit us to a first order language whose values are streams of integers, the environments and values are defined as :

- <*Val*>, the set of values contains every streams with a non-empty, totally evaluated prefix, i.e. prefix that is a non-empty list of constants,

  <*Val*> ::= <*Constant*> |<*ConstantList*> : <*WithoutVal*>

  <*ConstantList*> ::= <*Constant*> |<*Constant*> : <*ConstantList*>

The set <*Expr*> is decomposed as :

- <*Val*> is the set of values,
- <*WithoutVal*> is the complement of <*Val*> in <*Expr*>,
- <*Without:*> is the complement of expressions e_1:e_2 in <*Expr*>. ■

We define the environments of evaluation, called ENV, as a mapping that maps a variable into an expression (ENV: <*Var*> ↦ <*Expr*>). The semantic function $\mathfrak{C}$ defined below <*Decl*>∪<*Expr*>→-ENV→ENV adds variable-expression associations to initial environment, giving a new environment.

$\mathfrak{C}[\![rec]\ x(y) = e]\!]\varphi = [x \mapsto \lambda y.e] \cup \varphi$ — functions

$\mathfrak{C}[\![rec]\ x = e]\!]\varphi = [x \mapsto e] \cup \varphi$, — recursive or non-recursive variables

$\mathfrak{C}[\![d\ ;\ d^*]\!]\varphi = \mathfrak{C}[\![d^*]\!](\mathfrak{C}[\![d]\!]\varphi)$, $\mathfrak{C}[\![]\!]\varphi = \varphi$, — environment of lists of declarations

$\mathfrak{C}[\![let\ d^+\ in\ e]\!]\varphi = \mathfrak{C}[\![d^+]\!]\varphi$ — environment of expressions

$\mathfrak{C}[\![e]\!]\varphi = \varphi$ for some e excepted *let* d^+ *in* e' ■

Notations

— $v, v_i \in$ <*Val*> ; $c, c_i \in$ <*Constant*>,

— $x, y, z \in$ <*Var*> ; $e, e', e_i \in$ <*Expr*>, ; $d \in$ <*Decl*>, $d^* \in$ <*Decl*>*, $d^+ \in$ <*Decl*>$^+$,

— $f, f_i \in$ <*Without:*> ; $g, g_i \in$ <*WithoutVal*> ; $lc, lc_i \in$ <*ConstantList*>.

— $\varphi \in$ ENV , $\varphi[x] \in$ <*Expr*>∪{ω} ; It is such that : $\varphi[x] = e \Leftrightarrow \exists(y \mapsto e) \in \varphi :: y = x$

$\varphi[x] = \omega \Leftrightarrow \forall(y \mapsto e) \in \varphi :: y \neq x$ ■

3.2 Semantics

We define an immediate reduction relation →, between expressions in <*Expr*> defined in definition 2.1 except 4 and 9. The small step semantics is defined as the reflexive, transitive closure of this relation, written $\xrightarrow{*}$. We call this relation the '*call-by-name*' evaluation.

Definition 3.1

Let e be an expression in the environment φ (for example *let* d^+ *in* e is an expression e in the environment $\mathfrak{C}[\![d^+]\!]\{\}$), the semantics of e is the value v defined by the reduction $\varphi \vdash e \xrightarrow{*} v$. It means that there exists a finite sequence of reductions from e to v.

Var $\varphi \vdash x \rightarrow e$ if $\varphi[x] \neq \omega$ and $\varphi[x] = e$

Call $\varphi \vdash x(e) \rightarrow e'[e/y]$ if $\varphi[x] \neq \omega$ and $\varphi[x] = \lambda y.\ e'$

Plus $\varphi \vdash (c_1{:}e_1) + (c_2{:}e_2) \rightarrow c{:}(e_1 + e_2)$ where $c = c_1 + c_2$

Tl $\varphi \vdash tl(e_1{:}e_2) \rightarrow e_2$ *Hd* $\varphi \vdash (e_1{:}e_2){:}e_3 \rightarrow e_1{:}e_3$

$$S{:}_1 \quad \frac{\varphi \vdash f_1 \to e_1}{\varphi \vdash f_1{:}e_2 \to e_1{:}e_2} \qquad S{:}_2 \quad \frac{\varphi \vdash g \to e}{\varphi \vdash lc{:}g \to lc{:}e}$$

$$S_{tl} \quad \frac{\varphi \vdash f \to e}{\varphi \vdash tl(f) \to tl(e)}$$

$$S{+}_1 \quad \frac{\varphi \vdash g_1 \to e_1}{\varphi \vdash g_1{+}e_2 \to e_1{+}e_2} \qquad S{+}_2 \quad \frac{\varphi \vdash g_2 \to e_2}{\varphi \vdash e_1{+}g_2 \to e_1{+}e_2}$$

■

Annotations to definition 3.1

1. The semantics of a free variable is that of the associated expression in the environment.
2. The rule *Call* defines non-strict function application by β-reduction in the usual way. Note that such a semantics preserves the unfolding property.
3. The rule *Plus* defines addition pointwise. It is strict in both arguments.
4. The rules *Tl* and *Hd* satisfy the denotational semantics of *tl*, that is non-strict on the first value of its argument and that of ":" which is strict only on the first element of its first argument.
5. The recursive expressions compute the least solution to the recursive equations.
6. The rules S_i define a normal order and lazy evaluation. They lead to reduce subexpressions if and only if the whole expression is not a redex. For example, the rules $S{:}_1$ and $S{:}_2$ can apply only if the rule *Hd* cannot. Then the rule $S{:}_1$ reduces the first element of the stream, and next, we may evaluate the rest applying the rule $S{:}_2$. ■

Definition 3.2

We call '*call-by-slice-of-value*' evaluation of an expression $x(e)$ in an environment φ where $\varphi[x] = \lambda y.e'$ a reduction such that : $\varphi \vdash e \xrightarrow{*} lc{:}g$ followed by $\varphi \vdash \lambda y.e'[lc{:}y/y] \xrightarrow{*} v$. ■

Annotations to definition 3.2

Since a '*call-by-name*' evaluation converges if a '*call-by-value*' evaluation does and since a '*call-by-value*' is sound beside a '*call-by-name*' evaluation, we induce that the '*call-by-slice-of-value*' evaluation is sound in regards of the '*call-by-name*' one. ■

Some examples

In the sense of the '*call-by-name*' evaluation, the semantics of *let rec nat = 0:(nat+1) ; s = Q(nat) in s* is the infinite sequence of natural odd integers. That of *let rec s = Q(P(s)) in s* is the infinite sequence 1 5 21 84 336 .... The semantics of *let rec x=x:(0:(tl(x)+1)) ; f(y)=tl(y) in f(x)* is the infinite sequence of natural integers whereas that of *let rec x=x:(0:(tl(x)+1)) in x* is undefined since no reduction sequence converges. Notice that, in the sense of the '*call-by-slice-of-value*' evaluation, semantics values do not change except for *let rec x=x:(0:(tl(x)+1)) ; f(y)=tl(y) in f(x)* which isn't defined. ■

4. Parallel operational semantics

In this section we define an operational semantics that must reflect the parallel execution model of communicating processes. Section 4.1 defines a small step reduction semantics for AP2L provided with the ‖ annotation. In section 4.2 we discuss properties of the parallel annotation and prove its soundness beside the lazy operational semantics.

The operational semantics must reflect intuitions about behaviour of communicating processes. First, two branches of parallel expressions must be evaluated simultaneously or concurrently and the recursive networks (such as in fig. 2) must be computed in such a way that the output sequence is

evaluated once and only once. The rules $S\|$ and $S\|_i$ in def. 4.1 define parallel or concurrent computations. The rule *FB* means that a result stream which is also datum is communicated instead of computed again. Second, each branch of a parallel expression must be computed once and only once.

4.1 Small step semantics

The set <*Expr*> is decomposed as in section 3.1, moreover we distinguish expressions call <*WithoutRedex*> that are the set of expressions in <*Expr*> that do not contain a redex. ■

We define the environment of evaluation as in section 3.1, but we distinguish the recursive variables which are recursive networks (variables defined by a recursive unfolded expression with parallel annotations) as : $\mathfrak{C}[\![rec\ x = e]\!]\varphi = [x \rightarrow (\mathbf{Y}x.e[y(x)/(y\|(x))]] \cup \varphi$. In recursive networks, we replace parallel applications by usual ones in order to compute the resulting stream x once.

Notations : $h, h_i \in$ <*WithoutRedex*>, ■

We define an immediate reduction relation $\rightarrow$ called '*parallel-call*' evaluation between expressions in <*Expr*>. It is defined by union of rules in def. 3.1 and those given below in def. 4.1. The small step semantics is defined as the reflexive, transitive closure of this relation, written $\xrightarrow{*}$.

Definition 4.1

Let e be an expression in the environment φ, the semantics of e is the value v defined by the reduction $\varphi \vdash e \xrightarrow{*} v$. It means that there exists a finite sequence of reductions from e to v.

FB $\varphi \vdash (\mathbf{Y}y.c_1{:}e_1) \rightarrow c_1{:}(\mathbf{Y}y.e_1[c_1{:}y/y])$

$S_{\mathbf{Y}}$ $\dfrac{\varphi\backslash(y \rightarrow e') \vdash g \rightarrow e}{\varphi \vdash \mathbf{Y}y.g \rightarrow \mathbf{Y}y.e}$ $\qquad$ S_{λ} $\dfrac{\varphi\backslash(y \rightarrow e') \vdash g \rightarrow e}{\varphi \vdash \lambda y.g \rightarrow \lambda y.e}$

All$\|$ $\varphi \vdash e_1{}^0\|^0 e_2 \rightarrow e_1{}^m\|^n e_2$ where m, n > 0

All$P$ $\varphi \vdash h_1{}^m\|^0 g_2 \rightarrow h_1{}^m\|^n g_2$ where m, n > 0

All$C$ $\varphi \vdash g_1{}^0\|^n h_2 \rightarrow g_1{}^m\|^n g h_2$ where m, n > 0

Com $\varphi \vdash (\lambda y.g_1)^m\|^n(c_2{:}e_2) \rightarrow (\lambda y.g_1[c_2{:}y/y])^m\|^n e_2$

Out $\varphi \vdash (\lambda y.c_1{:}e_1)^{\,m}\|^n e_2 \rightarrow c_1{:}((\lambda y.e_1)^{\,m}\|^n e_2)$

$S\|$ $\dfrac{\varphi \vdash e_1 \rightarrow e_3,\ \varphi \vdash e_2 \rightarrow e_4}{\varphi \vdash e_1{}^{m+1}\|^{n+1} e_2 \rightarrow e_3{}^m\|^n e_4}$ where m, n≥0

$S\|_1$ $\dfrac{\varphi \vdash e_1 \rightarrow e_3}{\varphi \vdash e_1{}^{m+1}\|^n e_2 \rightarrow e_3{}^m\|^n e_2}$ $\qquad$ $S\|_2$ $\dfrac{\varphi \vdash e_2 \rightarrow e_4}{\varphi \vdash e_1{}^m\|^{n+1} e_2 \rightarrow e_1{}^m\|^n e_4}$ where m, n ≥ 0 ■

Annotations to definition 4.1

1. The rules S_i define a normal order evaluation except for parallel expressions. Apart from $S_{\mathbf{Y}}$, $S\|$ and $S\|_i$, the rules S_i lead to reduce subexpressions if and only if the whole is not a redex.
2. The rules $S_{\mathbf{Y}}$ and S_{λ} pull associations out of the environment to avoid a capture of the bound variable y by a free occurrence in the environment.
3. The rule $S_{\mathbf{Y}}$ allows us to compute a new value in the resulting stream of cyclic networks. After that, the rule *FB* (*FeedBack*) communicates this value as datum and put it in output channel. Such reduction carry out rendez-vous from output to input of networks. The rule *Com* realizes such an action between a producer and a consumer outside recursive edges in networks. The rules $S\|_i$ induce that every process is provided with unbounded *fifo* buffers as input and output. ■

Some examples

The parallel operational semantics of examples given in section 3.2 and annotated with the parallel operator as : *let rec nat = 0:(nat+1) ; s=Q ‖ nat in s, let rec x=x:(0:(tl(x)+1)) in x, let rec s=Q ‖ P(s) in s,* are identical to their lazy semantics. Unlike the others, *let rec x=x:(0:(tl(x)+1)) ; f(y)=tl(y) in f ‖ (x)* is undefined in the sense of after some steps the expression $f^m \| ^n(x)$ is reduced to $\lambda y.tl(y)\ ^{m-1}\|^{n-1}(x:(0:(tl(x)+1)))$ which is neither a redex nor a value. This drawback is due to the parallel annotation that introduces a '*call-by-value*' evaluation of *f(x)* which corresponds to a '*call-by-slice-of-value*' evaluation or a strict denotational semantics of *tl(x)*. ■

4.2 Properties of ‖ annotation

Since a parallel annotation is present, an expression may have many reduction sequences. It might seem that evaluating each branch of a parallel expression and communicating from producer to consumer would give an adequate operational semantics for ‖ :

$$\frac{e_1 \to e_3}{e_1 \| e_2 \to e_3 \| e_2} \qquad \frac{e_2 \to e_3}{e_1 \| e_2 \to e_1 \| e_3} \qquad \frac{e_1 \to e_3,\ e_2 \to e_4}{e_1 \| e_2 \to e_3 \| e_4}$$

$$(\lambda y.e_1) \| (c_2{:}e_2) \to (\lambda y.e_1[c_2{:}y/y]) \| e_2$$ ■

However, this strategy allows an infinite branch (for example a producer) to lead to the divergence of a parallel expression. In fact, these rules define erratic parallelism, in the sense of an *erratic* nondeterministic choice as defined in [25, 13]. To define an *angelic* parallelism, we need a mechanism that will ensure that convergent reductions are chosen if there exists at least one.

As HUGHES and MORAN in [13] about the nondeterministic choice operator, we do this by assigning to each branch a finite amount of *evaluation resources*, allocated by a fair scheduler. The expression $e_1{}^m \|^n e_2$ means that the left branch has *m* reductions allocated to it, while the right branch has *n*. A branch may reduce only if it has resources remaining. When both branches have used their allocation, the scheduler allocates non-zero positive resources to both with the rule *All‖*. When only one branch has used its allocation and when the other does not contain redex, the scheduler allocates positive resources to the first with the rule *AllC* or *AllP*. For example, it is the case if a producer has used its allocation and cannot communicate with a consumer that still have resources but cannot compute. Parallel expressions $e_1 \| e_2$ are represented in the operational semantics by $e_1{}^0 \|^0 e_2$.

The rules that express this scheduling are given in def. 4.1. To ensure that each branch gets non-zero positive reductions in each allocation phase, the rules prevent an infinite branch from causing the divergence of the whole parallel expression unless this one be undefined in the lazy semantics.

We claim our semantics is *fair* in the intuitive following sense : it is impossible to choose infinitely many times a branch if the other is reducible. We prove the two following properties :

- *soundness* : for all reduction according to '*parallel-call*', there exists a '*call-by-name*' reduction.
- *fairness* : for all reduction according to '*call-by-name*' and '*call-by-slice-of-value*' evaluation, any assignment for *m* and *n* leads to a '*parallel-call*' reduction.

5. Conclusions

We have presented a small applicative data-flow language supplied with a parallel annotation which corresponds to a fair weak non-determinism. These are concepts for a high level programming model for MIMD multicomputers. The proposed implementation, based on a scheduler of computing re-

sources, reflects intuitive parallel and asynchronous evaluation by a network of communicating interpreters. An evaluator may be associated to every branches of parallel expressions. Each of them reduces one branch and communicates its resulting stream to the interpreters that use it. Such an implementation induces three main advantages for efficiency : first it does not have to detect parallel tasks since they are explicit, second the size of parallelism grain is sufficient if users choose it correctly, and last, since interpreters do not share any subexpressions, implementations may satisfy the property of locality. Therefore our proposals can be easily implemented on MIMD architectures with distributed memory. Notice that without the parallel annotation, this is an usual data-flow programming model independent of some execution model as it is shown in section 3. Moreover, this model own important properties of functional paradigm such as unfolding and referential transparency in the sense given in [24].

6. References

[1] Ascroft E.A., Wadge W.W. - *LUCID : a non procedural language with iteration*. CACM 20, pp. 519, 1977.

[2] Backus J. - *Can programming be liberated from the VON NEUMANN style ?*, CACM 21, 8, 1978.

[3] Berthomieu G. - *Le langage L.C.S. et son interprète : une implantation expérimentale de CCS*. Colloque C^3, 85.

[4] Mc Carthy J. - *A basis for mathematical theory of ...*,Comp. progr. and formal syst..,North-Holland,p. 33-70, 63.

[5] Dannenberg R.B. - *Arctic : a functional language for real-time control*, Proc. ACM of Symp. on Lisp and functional programming, Austin, USA, pp. 96-103.

[6] Fairbairn J., ... - *Code generation techniques for functional languages*. Proc. ACM Conf. on LISP and functional programming, Cambridge, Mass., pp 94-104, 1986.

[7] Friedmann D., Wise D. - *An indeterminate constructor for applicative programming environment*. Conf. Record of the 7th Symp. on Principals of Programming Languages, 1980.

[8] Halbwachs N., ... - *Programming and verifying Real-Time systems by means of the synchronous data-flow language LUSTRE*. IEEE Trans. on Software Engineering, vol 18, n° 9, pp. 785-793, 92.

[9] Harrison D. - *RUTH : a functional language for real-time programming*, PARLE'87, LNCS n° 259 p. 297, 1987.

[10] Henderson P.-*Purely functional operating systems*,Funct. progr. and its appl.,Cambridge U. Press,p.177-192, 82.

[11] Hoare C.A.R. - *Communicating sequential processes*- CACM vol. 21, n° 8, 1978.

[12] Holmström S. - *PFL : a functional language for parallel programming*. Proc. of decl. languages for ll progr., 84.

[13] Hughes J., Moran A. - *Natural semantics for Nondeterminism*,Internal.Report. of Chalmers University, 1993.

[14] Julliand J. - *Une approche applicative de la programmation parallèle asynchrone*. Thesis Univ. of Nancy 1, 93.

[15] Kahn G. - *The semantics of a simple language for parallel programming*. Proc. of IFIP congress, 1974.

[16] Kelly P., ... - *An implementation of static functional process networks*, PARLE'92, LNCS n°605 p. 497-512.

[17] Le Metayer D. - *Approche fonctionnelle de la théorie à la pratique*, Thesis, Univ. of Rennes1, 90.

[18] Milner R. - *A proposal for standard ML*. Proc. of the ACM Symp. on LISP and functional progr., Austin, 1984.

[19] Milner R. - *Communication and concurrency*. Prentice-hall series in computer sciences, 1989.

[20] Misra J. - *Equational reasoning about nondeterministic processes*. Formal aspects of computing, p 167-195, 90.

[21] Perrin G.R., ... - *Communication relations:a paradigm for parallel program design*, S.C.P., vol 19, p. 25-59, 92.

[22] Plasmeijer M.J., ... - *The concurrent clean system*, .. Proc. 7nd Int. Conf. Apple, Paris, 1991.

[23] Peyton Jones S.L. - *Implementation of functional programming languages*, Masson, 1990.

[24] Sestoft P., ... - *Referential Transparency, definiteness and undfoldability*. Acta Informatica 27, p. 505-517, 90.

[25] Sestoft P., ... - *Non-determinism in functional languages*, Computer journal vol 35, n°5, pp. 514-523, 1992.

[26] Turner D.A. - *Miranda : a non strict functional language with polymorphic types*, L.N.C.S. n° 201, pp. 1-16.

RETARGETABLE COMPILER OF ANSI C SUPERSET FOR VECTOR AND SUPERSCALAR COMPUTERS

S. Katserov, A. Lastovetsky
(Computer Centre for Scientific Research, Moscow State University,
Vorob'ovy gory, 119899, Moscow, Russia)
S. Gaissaryan, D. Khaletsky, I. Ledovskih
(Institute for System Programming, Russian Academy of Sciences,
25, B.Kommunisticheskaya ul., 109004, Moscow, Russia
e-mail: ssg@ivann.delta.msk.su)

The paper describes an ANSI C language superset for vector and superscalar computers and its retargetable compiler prototypes. The superset, named C[], allows one to write portable efficient programs for SIMD (vector and superscalar) computer architectures. The article discribes the motivation of our approach, the vector superset of the C language, and the retargetable compiler system.

Key words: parallel programming, C language, vector computers, superscalar computers.

1 Introduction

The C language is commonly used by professional programmers because it allows one to develop highly efficient software portable within the class of UNIX systems. C reflects all main features of UNIX systems' architecture, which has an impact on the program efficiency [1].

As computer architectures have changed it has become necessary to reflect the changes in the compiler's internal languages, by adding constructs to express new computing facilities, such as vector calculations. But if we want to use these new facilities explicitly in programs, they should also be added to the C language.

We created a C language superset with the same vector capabilities as vector computer assembly languages, by adding several new notions to ANSI C. The resulting extended C languages, named C[], which allows one to write portable efficient programs for SIMD (vector and superscalar) computer archutectures [2]. Our motivation of the C[] language is given in sec. 2.

In sec. 3 we give a brief description of the C[] language as it is published in [2]. In sec. 4 two new statements added to C[] after publishing [2] are discussed. In sec. 5 we discuss principles of C[] implementation for superscalar computers and especially for Intel i860. In sec. 6 the retargetable compiler prototype system is described.

2 Motivation

As vector and superscalar architectures are an evolution of UNIX systems architecture, the language which plays the same part as C does for UNIX systems may be developed as a superset of the C language.

There have been many efforts to develop such C supersets ([3], [4], [5], etc.), but the supersets we know have following disadvantages:

- The conceptual models of these supersets are not sufficiently developed (for example, the concept of vector value is absent).
- The conceptual models reflect some peculiar features of particular architectures having no analogs in other vector and superscalar architectures (for example, the notion of descriptor in Vector C language [3] is natural for the Cyber 205 but not natural for supercomputers of Cray family because all their vector instructions are of register-to-register type; the notion of parallel objects in the C* language [4] is natural for the Connection Machine 2 but not natural for Cray, Cyber 205, and other shared memory supercomputers because it excludes explicit parallel processing of arrays).
- The supersets do not take into account requirements related to implementation of the compiler being portable and retargetable to particular architectures of considered class.

We considered the following requirements while developing the superset of C for vector and superscalar computers:

- The superset must adequately reflect all common features of the relevant architectures.
- The conceptual model of the superset must provide simple and efficient imlementation for all computers of the class.
- The superset must be suitable for implementation of the portable and retargetable compilers.

3 Brief description of the C[] language

The C[] language is a strict superset of ANSI C [6]. The following a brief description of its main features as it was described in [2].

The basic new notion of the C[] language is a notion of 'vector value' (or simply 'vector'). A vector is defined as an ordered sequence of values of any type (the elements of the vector). The types of all the elements of a vector must be the same. In contrast to arrays, a vector is not an object, it is a new sort of value.

An array is a container of vector values. The unary postfix [] operator is applied to a operand of array type and provides an access to the vector, being the value of the array object. Formally the [] operator cancels the conversion of the operand to a pointer.

The notion of array in the C[] language is extended by adding new attribute, namely, the step of allocation of array members in storage (in particular, it allows us to introduce the notion of subarray sensible enough). Correspondingly, the notion of pointer is extended as well as address arithmetic. Namely, a pointer has new attribute step, and address arithmetic takes into account this new attribute.

A formal parameter of a function may have an array type. The corresponding argument is an expression of the same vector type that is defined by the formal parameter. The function value also may have vector type.

The notion of vector causes the notion of lvector. Just as an lvalue is an expression designating some object, an lvector is a vector expression designating a set of objects.

The operand of unary *****, +, -, ~, ?, %, @, ! operators and scalar cast operators may have a vector type. One or both operands of binary *****, /, %, ?<, ?>, +, -, <<, >>, <, >, <=, >=, ==, !=, **&**, ^, |, **&&**, || operators may have vector type. An assignment operator may have as its left operand an lvector. In that case its right operand may have vector type. In any case the type of its right operand converts to the type of the left operand's value. The conditional operator may also have vector operands.

The linear (or reduce) operators are introduced. The unary linear **[*]**, **[/]**, **[%]**, **[?<]**, **[?>]**, **[+]**, **[-]**, **[<<]**, **[>>]**, **[&]**, **[^]**, **[|]** operators correspond to binary *****, /, %, ?<, ?>, +, -, <<, >>, **&**, ^, | operators. These operators are applicable only to vector operands.

The set of C[] operators together with facilities of packing integer vectors into bit-fields provide a general set of vector manipulations in vector computers.

4 New constructs of C[]

In this section we discuss two new statements — **par** and **pipe**, — added to C[] after publishing of [2] to describe parallel and pipelined calculations.

The **par** statement is another form of compound statement: a list of statements is enclosed in braces **{** and **}** preceded by the keyword **par**. It means that all these statements may be executed in arbitrary order, in particular, in parallel. Each result of the execution is considered be correct even if it depends on the order the statements are executed.

The **par** statement allowes the programmer to express data dependencies of his program in terms of the source language. If the programmer is sure that there are no data dependencies between some statements, he may include them in **par** statement. In general it will lead to more profound program optimization.

The **pipe** statement is a special loop statement, which allowes to express "skewed" loops. The idea of **pipe** is borrowed from [7]. It has the form

pipe (<expression>$_{opt}$;<expression>$_{opt}$;<expression>$_{opt}$) <statement>

The <expression>s have the same semantics as those in **for** statement. If body of **pipe** is a compound statement, it may contain a special label **p+:**. It means that the part of the current iteration beginning with the statement marked by this label may be executed in

parallel with the next iteration of the loop. If the label p+: is omitted it is implied that it marks the first statement and therefore all iterations of the loop can be executed in parallel. It is supposed that if the compound statement, being a pipe body, contains definitions of automatic variables, these variables are different for each iteration, that is if n iterations run in parallel, there are n different instances for each such variable.

5 C[] implementation for superscalars

It is obvious that C[] is suitable for vector pipelined computers But C[] is also suitable for superscalar computers because the vector expressions and constructs **par** and **p+:**, presented in sec. 4, allow to point parts of the program which can efficiently use their parallel facilities (pipelines).

It may be pointed out dicussing the mapping if C[] to Intel i860 microprocessor.

All scalar operators of C[] are translated to corresponding scalar operations of the i860 processor.

The vector operators of C[] can be mapped on pipelined i860 operations. For example, vector expresion

`c[] = a[] + b[]` may be mapped in the following sequence of instructions:

```
//*************************************************************************
//      c[]=a[]+b[]
//                          input:      r16 - address of vector a[]
//                                      r17 - address of vector b[]
//                                      r20 - vector size
//                          output:     r15 - address of vector c[]
//*************************************************************************
        fld.q       r0(r16),f8          //Load first 4 elements of A[]
        fld.q       r0(r17),f12         //Load first 4 elements of B[]
        mov         -4,r14              //r14 is used to decrement counter
        adds        r14,r20,r20         //dectement counter
        .dual                           //Enter dual mode
        bla         r14,r20,L0          //Set LCC flag
        adds        -16,r15,r15         //prepare C address
L0::    pfadd       f8,f12,f0           //Put first elements in pipe
        fld.q       16(r16)++,f20       //Load next 4 elements of A[]
        pfadd       f9,f13,f0           //Put next elements in pipe
        fld.q       16(r17)++,f24       //Load next 4 elements of B[]
        pfadd       f10,f14,f0          //Put next elements in pipe
        bla         r14,r20,L2          //Check if there are more elements
                                        //and set LCC
        pfadd       f11,f15,f16         //Put next elements in pipe
        nop
        .enddual                        //Exit dual mode
        pfadd       f0,f0,f17           //Get result from pipe
```

```
        nop                         //Still in dual mode
        pfadd       f0,f0,f18       //Get result from pipe
        nop                         //Still in dual mode
        pfadd       f0,f0,f19       //Get result from pipe
        br          END             //Goto END
        fst.q       r16,16(r15)     //Store results
L1::    pfadd       f8,f12,f29      //Put next elements in pipe
        fld.q       16(r16)++,f20   //Load next 4 elements of A[]
        pfadd       f9,f13,f30      //Put next elements in pipe
        fld.q       16(r17)++,f24   //Load next 4 elements of B[]
        pfadd       f10,f14,f31     //Put next elements in pipe
        bla         r14,r20,L2      //Check if there are more elements
                                    //and set LCC
        pfadd       f11,f15,f16     //Put next elements in pipe
        fst.q       r28,16(r15)++   //Store results
        .enddual                    //Exit dual mode
        pfadd       f0,f0,f18       //Get result from pipe
        nop                         //Still in dual mode
        pfadd       f0,f0,f19       //Get result from pipe
        br          END             //Goto END
        fst.q       r16,16(r15)     //Store results
L2::    pfadd       f20,f24,f17     //Put next elements in pipe
        fld.q       16(r16)++,f8    //Load next 4 elements of A[]
        pfadd       f21,f25,f18     //Put next elements in pipe
        fld.q       16(r17)++,f12   //Load next 4 elements of B[]
        pfadd       f22,f26,f19     //Put next elements in pipe
        bla         r14,r20,L1      //Check if there are more elements
                                    //and set LCC
        pfadd       f23,f27,f28     //Put next elements in pipe
        fst.q       r16,16(r15)++   //Store results
        .enddual                    //Exit dual mode
        pfadd       f0,f0,f29       //Get result from pipe
        nop                         //Still in dual mode
        pfadd       f0,f0,f30       //Get result from pipe
        nop                         //Still in dual mode
        pfadd       f0,f0,f31       //Get result from pipe
        fst.q       r28,16(r15)     //Store results
END::   nop
//*****************************************************************************
```

The expression:

```
c[] = k * a[] + b[]
```

where k is a scalar variable, may be treated in similar way; this example allowes to achieve pipeline chaining.

The linear (or reduce) operators can be mapped to sequence of instructions containing pipelined operations. For example, the expression

```
s = [+] a[]
```

may be mapped to the following sequence of instructions:

```
//*****************************************************************************
// s = [+] a[]
//        input:              r16 - &a[]
//                            r17 - size of the vector a[] (must be >5)
//        output:             f16 - s
//*****************************************************************************
//
       fld.d       r0(r16),    f20              //Load first 2 elements
       mov         -2,         r21              //Loop decrement for bla
       .dual                                    //Enter dual mode
       pfadd.ss    f0,         f0,        f0    //Clear adder pipe (1)
       adds        -6,         r17,       r17   //Decrement size by 6
       pfadd.ss    f0,         f0,        f0    //Clear adder pipe (2)
       bla         r21,        r17,       L1    //Initialize LCC
       pfadd.ss    f0,         f0,        f0    //Clear adder pipe (3)
       fld.d       8(r16)++,   f22              //Ld 3th & 4th elements
//*****************************************************************************
L1::   pfadd.ss    f20,        f30,       f30   //Add f20 to pipe
       bla         r21,        r17,       L2    //If more go to L2 after
       pfadd.ss    f21,        f31,       f31   //adding f21 to pipe and
       fld.d       8(r16)++,   f20              //loading next f20:f21
       // If we reach this point, at least one element remains
       // to be loaded. r17 is either -4 or -3.
       // f20, f21, f22, and f23 still contain vector elements.
       // Add f20 and f22 to the pipeline, too
       pfadd.ss    f20,        f30,       f30
       br          sumup                        //Exit loop after adding
       pfadd.ss    f21,        f31,       f31   //f21 to pipe
       nop
L2::   pfadd.ss    f22,        f30,       f30   //Add f22 to pipe
       bla         r21,        r17,       L1    //If more go to L1 after
       pfadd.ss    f23,        f31,       f31   //adding f23 to pipe and
       fld.d       8(r16)++,   f22              //loading next f22:f23
       // If we reach this point, at least one element remains
       // to be loaded. r17 is either -4 or -3.
       // f20, f21, f22, and f23 still contain vector elements.
       // Add f20 and f22 to the pipeline, too
       pfadd.ss    f20,        f30,       f30
```

```
        nop
        pfadd.ss    f21,        f31,        f31
        nop
sumup::.enddual                                     //Exit dual mode
        pfadd.ss    f22,        f30,        f30     //Still in dual mode
        mov          -4,        r21
        pfadd.ss    f23,        f31,        f31     //Last dual mode pair
        bte         r21,        r17,        done    //If there is one more
        fld.l       8(r16)++,   f20                 //element, load it and
        pfadd.ss    f20,        f30,        f30     //add to pipeline
        // Intermediate results are still in the adder pipeline.
        // Let A1:A2:A3 represent the current pipeline contents
//*************************************************************************
done:: pfadd.ss     f0,         f0,         f30     // 0:A1:A2    f30=A3
        pfadd.ss    f30,        f31,        f31     // A2+A3:0:A1 f31=A2
        pfadd.ss    f0,         f0,         f30     // 0:A2+A3:0  f30=A1
        pfadd.ss    f0,         f0,         f0      // 0:0:A2+A3
        pfadd.ss    f0,         f0,         f31     // 0:0:0      f31=A2+A3
        pfadd.ss    f30,        f31,        f16     // f16=A1+A2+A3
//*************************************************************************
```

But C[] program may contain portions of sequential code, and this code should also be optimized, especially the loops. **par** and **pipe** statements allow compiler to reduce delays of the instruction pipeline of superscalar processor and therefore to optimize scalar code portions.

In the case of loops **par** statement provides the information necessary for loop body optimization. **pipe** statement provides the information for mutual optimization of loop iterations.

6 The compiler prototype system

The portable and retargetable compiler prototype system was implemented on Sun SPARC Workstation in UNIX. To implement the compiler prototype system the Karlsruhe toolbox for compiler construction [8] was used.

The compiler consists of four stages. The first stage analyses the source program file and builds its internal representation (the abstract syntax tree).

The second stage translates the internal representation into an intermediate language being an extension of the RTL language used in GCC [1].

In the third stage, the intermediate program is tuned to the target computer.

The fourth stage is a retargetable code generator. Currently, it generates code for the Russian Cray-like supercomputer [9] and for the Intel i860 [10].

References:

1. R.M.Stallman. Using and Porting GNU CC // Free Software Foundation, 675 Mass Ave, Cambridge, MA, May. 1992

2. S.Gaissaryan, and A.Lastovetsky. An ANSI C Superset for Vector and Superscalar Computers and its Retargetable Compiler //J. C Lang. Transl., v.5, No.3, March, 1994, p.p. 183 — 198

3. Kuo-Cheng Li, and H. Schwetman. Vector C: A Vector Processing Language // J. Parall. Vect. Comput. v.2, No.2, 1985, p.p. 132-169

4. Connectiom Machine Model CM-2. Technical Summary (Version 6.0) /Thinking Machines Corporation, Cambridge, Massachusets. Nov.,1990.

5. R.Gisselquist. An experimental C Compiler for the Cray-2 Computer // ACM SIGPLAN Notices, 21(9), 1986, p.p. 32-36

6. ANSI. Programming Language C. X3.159-1989. Americn National Standard Institute, 1989

7. T.G.Lewis. Foundations of Parallel Programming: A Machine-Independent Approach. / IEEE Comput. Soc. Press, 1994

8. J.Grosch. Toolbox Introduction. / Compiler Generation Report No 25, GMD Forchungsstelle an der Universitaet Karlsruhe, 1992

9. V.A.Melnikov, Ju.I.Mitropolsky, V.Z.Shnitman, V.P.Ivannikov, A.N.Tomilin, and S.S.Gaissaryan. High Performance Computer System "Electronica SSBIS" / Proc. Internat. Conf. Parallel Computing Technologies, Novosibirsk, Sept. 7 — 11, 1991, p.p. 47-55

10. **i860** Microprocessor Architecture. / Intel, Osborne, McGraw-Hill, 1990

DATAFLOW ASSEMBLY LANGUAGE PROGRAMMING SYSTEM

Dr Larisa G.Tarasenko
Dept. of Software Systems,
Scientific Computer Center of the Russian Academy of Sciences,
32 A, Leninsky pr., Moscow, 117334, RUSSIA
email: user@comcp.msk.su fax: 938 58 84 tel: 938 17 86

Summary.

This paper discusses some problems of dataflow programming. The aim of this paper is to identify the concepts of dataflow program design, debug and analysis and the dataflow program execution environment and performance specification.

The article includes some qualities of the dataflow Assembly language, discusses the way of the dataflow program debug, suggests the dataflow program execution environment and the dataflow program performance specifications. The basis dataflow machine is based on the tagged token model of dataflow computation.

Key Words: Dataflow, Assembler, debugger, interpreter, execution, performance.

Introduction.

Parallel computation is a rapidly expanding area of computer science. Previous attempts at multiprocessor systems have many problems with partitioning a single program among multiple processors and all partitioning effort is pushed onto the programmer. In addition, experience has shown that such systems have inefficient processor utilization and the hand partitioning process is very error prone. In order to overcome these fundamental difficulties, it was necessary to develop some fundamental new approaches to program representation. The basic idea of dataflow machine was offered by J. B. Dennis [1].

The main idea of dataflow was borrowed from optimizing compilers which represent programs as directed graphs. The dataflow machine executes the directed graph representation program. In dataflow computer the control of parallelism is out of the hands of the programmer.

In dataflow computers program parallelism is expressed in the form of directed graphs in which nodes describe operations and arcs describe paths along which data is routed from operation to operation.

The study of dataflow computation commenced at Stanford University [2] and MIT [3] around 1970, pioneer projects at the University of Utah [4], Texas Instruments [5] and CERT-ONERA Toulouse [6] in 1977, 1978 and 1979, respectively. Since 1980, active dataflow research has been concentrated at MIT, University of Manchester [7] Electro-Technical Laboratory [8] and the Nippon Telephone and Telegraph company [9].

A dataflow machine exploits the parallelism inherent in problems, and executes it in a highly concurrent manner, but many problems remain to be solved before a dataflow machine can be of practical use in a real environment and before a dataflow machine can be of practical use for programmers. There are some problems of dataflow program design, checkout and development. There is need for a language that can be translated into dataflow machine code.

Some research of dataflow computer has been commenced at Scientific Computer Center of Russian Academy of Sciences in 1988 as a part of the Optical super-

computer Project of Russian Academy of Sciences. This article discusses the dataflow Assembly language programming system, consisting of Assembler, debugger, interpreter and program analyzer. This system was designed in Scientific Computer Center (SCC) of Russian Academy of Sciences and is the part of the Optical super-computer Project of Russian Academy of Sciences [10].

1. Dataflow computing models.

Dataflow systems implement the abstract graphical model of computation. Individual systems differ mainly in their approach as to how the computing should proceed. There are some basic dataflow computing models [11]. The main dataflow computing models are: static, recursive dynamic, tagged token dynamic, lazy-eager.

Static dataflow computing was proposed by Dennis for ultrahigh speed computing machines [1]. This computing model consists of operators, data and control arcs, and data and control tokens. There are allowed several tokens per link but there is a restriction of one token per time. An actor fires when there are no tokens on any of actor's output arcs. Concurrent reentrance is inhibited.

In dynamic dataflow computing [4], several instances of a node can be fired at a time and these nodes can be created at run time. Concurrent reentrance is permitted by code copying. In code copying, a new instance of a subgraph is created. This model enables recursive computations in the dataflow computing environment.

The tagget token dynamic computing model [7], proposed separately by Arvind and Gurd-Watson. A tag assigned to each token distinguishes its identity. Identically tagged tokens enable the execution of an operation. Tagging allows many data values per link at one time. Several instances of a node are fired at one time. Recursion and iteration are represented directly. Successive cycles of an iteration are allowed. The tagget token dynamic computing model is most efficient in exploiting the parallelism.

The eager-lazy dataflow computing model [9] was proposed by Amamiya. In eager evaluation, all possible computations are executed in parallel without optimizing. Conditional computations are executed parallel to the branches. CAR and CDR parts are evaluated in parallel to the CONS. CONS(x,y) is implemented using the getcell, writecar and writecdr operations. This is the lenient cons mechanism. In lazy evaluation, selected computations are executed to optimize the computation. The selected branch is executed after the execution of the conditional computation. In the lazy cons mechanism, the car or cdr part is evaluated only when its value is demanded. This model may be efficient for artificial intelligence applications.

2. Fundamental decisions.

The SCC Dataflow Machine is based on the tagged token model of dataflow computation. A machine-level program is a directed graph in which nodes represent machine instructions and arcs represent paths along which data travel from instruction to instruction at run-time. Carriers for data values are tokens. They are fired into the graph and each instruction determines individually whether or not to execute on the basis of the availability of its required input data. Since this decision is made locally at each instruction, the overall pattern of execution is entirely asynchronous.

Each token is divided into two fields, such as tag and data. Tag is divided into four fields, such as index, iteration, ganaration and address. The ganaration is used to distinguish concurrent function calls. The iteration is used to distinguish successive cycles of loops. The index is used to distinguish different array elements. The address is used to distinguish tokens to different instructions. Data field is divided into two fields, such as type and value. There are some restrictions. Each instruction has one or two input operands and at most two output arcs.

The instruction may be executed if all the input tokens have arrived. So there exist a token memory were arrives input tokens for two operands instructions. As each token arrives there, it is compared with resident tokens to see whether its partner token is already available and, if so, the relevant token is extracted from the token memory and sent off for further processing. Otherwise the incoming token is added to the set of the resident tokens.

Most of our research deal with design and analysis of dataflow programs and computations. So we design some program development system for design, debug, interpretation and analysis of dataflow programs.

3. The Assembly language.

The programming language is very important for program representation and low-level programming language is very important for efficient mapping the program onto the computing environment. Firstly, we design the Assembly language, which allow to describe dataflow computing efficiently.

The Assembly language program is the set of instructions. Each instruction can have one or two input operand and one or two result. The input operands of instructions name input tokens, and results of instructions name target tokens. Input and target tokens are named. If the input token name of one instruction coincides with the target token name of other instruction, it means, that during execution of the program the input operand of the first instruction is the result of second instruction.

Several instructions in one program can have as input operands equally named tokens. It will mean, that all these instructions use as the input operand and the result of the same instruction. In turn, several instructions in one program may have as results equally named tokens. It will mean, that the results of all these instructions shall be transmitted as operands to the same instruction.

The program may have the instruction, which input token and result have one name. It will mean, that the output of this instruction is its input. Thus, the Assembly language program describes oriented graph, which may contain the cycles and loops. Each node of this graph may have arbitrary number of input and output arches. The constant may be the input operand of instruction.

The Assembly language program contain four types of constants: integer, real, structural and address ones. The structural constant contain index, iteration ganaration and address of the array cell. The address constant contain the address of the instruction in the code memory.

The Assembly language program contain two kinds of constants: temporary and permanent. If the temporary constant is the input operand of any instruction, then it may be use once and this instruction may have another input tokens instead this constant. If the permanent constant is the input operand of any instruction, then it may be use multiple and this instruction have no another input tokens instead this constant.

The Assembly language program have several input operands. The input operand of dataflow program is the input token of it's instruction, which is not the target token of some another dataflow program instruction. The input operands of dataflow program may be defined before it's evaluation.

The result of Assembler is the machine-level dataflow program. The machine-level program consists of two files: codefile and datafile. Codefile contain program code and permanent constants and datafile contain the set of program temporary constants. There are a few kinds of dataflow program evaluations.

The primary dataflow program evaluation is the program evaluation without it's input operands. The result of the primary dataflow program evaluation is the machine-level dataflow program. The primary dataflow program evaluation calculates dataflow program constant expressions. The dataflow constant expression is the dataflow program subgraph without input arcs from another nodes except subgraph nodes.

The partial dataflow program evaluation is the program evaluation without entirely program input operands define [12]. The result of the partial dataflow program evaluation is the dataflow program. The partial dataflow program evaluation calculates only a few dataflow program operators.

The entirely dataflow program evaluation is the program evaluation with entirely program input operands define. The result of the entirely dataflow program evaluation is the set of dataflow program output data.

So, the dataflow program may be executed step by step by input operands arrive. As the result we have the sequence of partial dataflow program evaluations. The full set of program evaluations is equal to the sum of partial program evaluations. The step by step dataflow program evaluation is the foundation for dataflow programs generation.

The Assembly language contain a few kinds of operators: main and additional. The main Assembly language operator is equal to some machine-level instruction. The additional Assembly language operator defines some additional actions. For example, the additional operator may define the temporary constant, or internal and external input and output operands.

So, the dataflow program may be integrated from several number of dataflow programs. Each of this programs will be translated and primary evaluated independently.

The example of the Assembly language program is:

```
x2,y2   = УМ (X,X)
x4      = УМ (y2,x2)
x3      = УМ (y2,x)
5x      = УМ (x,цел 5)
x2x4    = СЛ (x2,x4)
x35x    = СЛ (x3,5x)
Y       = СЛ (x2x4,x35x)
        = ПВМ (Y,цел 0)
```

This dataflow program has one input operand X and computes the expression:

$Y=X^4+X^3+X^2+5*X$

The next example of the Assembly language program is:

```
mem: 0 0 0 0 1 1 0 1 0
mem: 0 0 0 0 2 1 0 0 0
c       = РЗМ (X,N)
```

a,z = УМ (c,z)
b,y = СЛ (a,b)
= ПВМ (Y,цел 0)

This dataflow program has two input operands X and N and computes expressions:

$Y=X$
$Y=X+X^2$
$Y=X+X^2+X^3$
.....
$Y=X+X^2+X^3+...+X^n$

Operator MEM defines the temporary constant. Operators УМ and СЛ define multiplication and addition instructions. Operator РЗМ defines token's multiply and operator ПВМ defines token's output.

The next example is.

It is necessary to compute:

$X_i=A_i*X_{i-1}+D_i$ for i=1,...,n.

We use for calculation the following method:

$X_i^{(0)}=D_i$ $X_j = 0$ if $j<1$ and $j>n$

$X_i^{(p)}=A_i^{(p-1)}*X_{i-2^{**(p-1)}}^{(p-1)}+X_i^{(p-1)}$ $p=1,2,...\log_2 n$

$A_i^{(p)}=A_i^{(p-1)}*A_{i-2^{**(p-1)}}^{(p-1)}$

The dataflow program have input operands n, A[], D[]. This program may be executed step by step. The partial step1 dataflow program evaluation is the dataflow program evaluation with one input operand n. The partial step2 dataflow program evaluation is the dataflow program evaluation with input operand D[], and the entirely dataflow program evaluation is the dataflow program evaluation with input operand A[]. The execution times of the primary, partial and entirely dataflow program evaluations are 3, 28, 28, 26 respectively, and the number of executed operands are 4, 515, 188, 159 respectively.

So, we generated two new dataflow programs. First new dataflow program has two input operands A[] and D[]. Second new dataflow program has one input operand A[].

4. The dataflow program interpretation.

The interpreter processes the machine-level program. The program consist of two files: codefile and datafile. Codefile contains the program code and permanent constants. Datafile contains. temporary constants. The interpreter loads the input data into the array for token memory and the program code into the array for code memory.

The interpreter has some parameters such as: number of processors, code memory capacity, token memory capacity and the interpretive mode and uses them for interpreter control.

We have a step by step interpretation. A step corresponds to the CPU cycle. In one step the processor can be occupied or free. Each instruction takes one or more steps. The interpreter processes ready instruction if it has one free processor for it. The processor will be occupied with this instruction the number of steps correspond it. The parallelism of the program of any step of the interpretation is equal to the number of occupied processors in this step. The program profile is the

XY graph, where X defines the step and Y defines the program parallelism inside the step of interpretation. The interpreter instantiates the program profile.

Each instruction produces one or two output tokens and sends them to the target instructions. If the target instruction must have two input tokens and the second token is not ready the first token will be written into the array for token memory. The memory profile is the XY graph, where X define the step and Y define the necessary token memory capacity inside the step of interpretation. The interpreter instantiate the memory profile.

5. The dataflow program debugger.

The conventional program consist of the ordered program code and data memory. The result of the consecutive execution of the program code is the data memory updating. So, the debugging of such program include the consecutive analysis of the data memory.

The dataflow program has no updating data memory. The dataflow program is the directed graph in which nodes represent machine instructions and arcs represent paths along which data travel from instruction to instruction at run-time. The execution of dataflow program is the step-by-step performance of arbitrary instruction set.

So, the dataflow program debug includes the analysis of data tokens travelling from instruction to instruction at run time. The dataflow program debugger must allow to analyze all input and output tokens of arbitrary subgraph of dataflow program graph, determining by means of its nodes. The dataflow program debugger must allow to analyze node of the tokens creation.

The debugger is used for debugging dataflow programs during their interpretation. The debugger uses breakpointing. There are several breakpoint types: addressing, selecting, inserting. The addressing breakpoints are defined with the address of breakpoint instruction. The selecting breakpoints are defined with the tag of the token selecting from the token memory. The inserting breakpoints are defined with the tag of the token writing into the token memory.

User may create several different breakpoints in the program. The debugger will stop the interpretation of the program at each breakpoint and let the user to watch and modify breakpoint data set and make breakpoint restart or program restart. The breakpoint data set consist of the input tokens and token memory contents.

A user may order to readout breakpoint data set into the file or display. User may order to have breakpoint at each program instruction.

6. The dataflow program execution environment and performance specification.

The dataflow computer has no memory, but it has a token memory. The token memory capacity is the critical resource of the dataflow system. The main dataflow program performance specification is the execution time, but the dataflow computer has a dynamic architecture. It may have a variable number of processors and token memory capacity. So, it is necessary to define the functional dependence of dataflow program performance and dataflow program execution environment from the processors number and token memory capacity. It is necessary to define the dataflow program execution environment and the dataflow program performance specifications.

6.1. Declarations.

A - The numerical algorithm,

P(A) - The set of programs for algorithm A,

p - The program from the set P(A),

T(p) - The execution time of the program p,

PR(t,p) - The program profile of the program p,

M(t,p) - The memory profile of the program p,

Define the functional form.

$$G(f(x,y),C) = \sum_{f(x,y)>0} \frac{|f(x,y)-C|+(f(x,y)-C)}{2}$$

Define functions.

PE(APE,p) = G(PR(t,p),APE) - The superfluous of the processors number requirement. The processors number is equal to APE.

ME(AMEM,p) = G (M(t,p),AMEM) - The superfluous of the token memory capacity requirement. The token memory capacity is equal to AMEM.

6.2. The dataflow program execution environment.

The expression:

PE(AMEM,p) = 0

is the execution environment of the dataflow program p.

The expression:

$\min_{p} PE(AMEM,p) = 0$

is the execution environment of the algorithm A dataflow computation.

6.3. The dataflow program performance specification.

The expression:

(T(p1) + PE(APE,p1)/APE) < (T(p2) + PE(APE,p2)/APE)

is the dataflow program performance specification. The performance of the dataflow program p2 is greater then the performance of the dataflow program p1.

The expression:

$\min_{p} (T(p) + PE(APE,p)/APE)$

is the condition of the most performance program for algorithm A.

The offered technique was used for determining the dataflow program performance. As a result it was possible to design more efficient machine-level dataflow programs and compiler algorithms.

Conclusion.

Previous attempts at multiprocessor systems have many problems with partitioning a single program among multiple processors and all partitioning effort is pushed onto the programmer. In addition, experience has shown that such systems have inefficient processor utilization and the hand partitioning process is very error prone. In order to overcome these fundamental difficulties, it was necessary to develop some fundamental new approaches to program representation.

The dataflow machine executes the directed graph representation program. In dataflow computer the control of parallelism is out of the hands of the programmer. In dataflow computers program parallelism is expressed in the form of directed graphs in which nodes describe operations and arcs describe paths along which data is routed from operation to operation.

This article discusses the dataflow Assembly language programming system, consisting of Assembler, debugger, interpreter and program analyzer. This system was designed in Scientific Computer Center (SCC) of Russian Academy of Sciences and is the part of the Optical super-computer Project of Russian Academy of Sciences. The SCC Dataflow Machine is based on the tagged token model of dataflow computation.

The programming language is very important for program representation and low-level programming language is very important for efficient mapping the program onto the computing environment. Firstly, we design the Assembly language, which allow to describe dataflow computing efficiently.

The Assembly language program describes oriented graph, which may contain the cycles and loops. Each node of this graph may have arbitrary number of input and output arches. The constant may be the input operand of instruction. The dataflow program may be integrated from several number of dataflow programs. Each of this programs will be translated and primary evaluated independently.

The dataflow program may be executed step by step by input operands arrive. As the result we have the sequence of partial dataflow program evaluations. The full set of program evaluations is equal to the sum of partial program evaluations. The step by step dataflow program evaluation is the foundation for dataflow programs generation.

The dataflow program debug includes the analysis of data tokens travelling from instruction to instruction at run time. The dataflow program debugger allows to analyze all input and output tokens of arbitrary subgraph of dataflow program graph, determining by means of its nodes. The dataflow program debugger allows to analyze node of the tokens creation.

We suggested the dataflow program execution environment and the dataflow program performance specifications, using dataflow program profile and dataflow memory profile.

The system is used for dataflow compilers debugging and dataflow program design. Some results of the program analysis were in use for computer architecture design.

It is important to note that this is only an initial attempt at dataflow program investigation. In particular, more work is required to determine the techniques for parallel algorithm design, dataflow program optimization, dataflow compiler design and dataflow program portability [13,14].

References.

1. Dennis J.B., "A Preliminary Architecture for a Basic Data Flow Processor", The Second Annual Symposium on Computer Architecture, Jan., 1975, p.126-132.

2. Adams D.A., A Computational Model with Data Flow Sequencing, Technical Report CS-117 (Stanford University, 1968).

3. Rodriguez J.D.., A Graph Model for Parallel Computations, Technical Report LCS-TR-64 (MIT,1069).

4. Davis A.L., The Architecture and System Methodology of DDM1: A Recursively Structured Data Driven Machine, Proceedings 5 th Annual Symposium on Computer Architecture (ACM,1978) pp. 210-215.

5. Johnson D. et al., Automatic Partitioning of Programs in Multiprocessor Systems, Proceedings IEEE Computer Conference (1980) pp. 175-178.

6. Plas A. et al., LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment, Proceedings International Conference on Parallel Processing (IEEE 1976) pp. 293-302.

7. Gurd J.R., et al., The Manchester Dataflow Computing System, in: Dongarre J.J., (ed.), Experimental Parallel Computing Architectures (North-Holland, 1987) pp. 177-216.

8. Yuba T. et al., SIGMA-1: A Dataflow Computer for Scientific Computations, Computer Physics Communications 37/1 (1985) pp. 141-148.

9. Amamiya M. et al., A List-Processing-Oriented Data Flow Machine Architecture, AFIPS Proceedings 51 (1982) pp. 143-151.

10. Burtsev V.S. et al., Fundamental Principles of RAS Super-computer Design, Technical Report SCC RAS, 1992.

11. Herath J., et al., Dataflow Computing Models, Languages, and Machines for Intelligence Computations, IEEE Transactions on Software Engineering, vol. 14, No. 12, 1988, p. 1805-1827.

12. Ershov A.P. Partial Evaluation: Potential Computation and Research Problems. // Software theory and practice: Proceedings Soviet-French Symposium. Part 1. Novosibirsk, 1978, pp. 5-40.

13. Tarasenko L.G. Some problems of the dataflow program portability, Pr. N 10, SCC RAS, 1991, p.12.

14. Tarasenko L.G. Some investigations of the functional language FP dataflow computer applications for numerical algorithms, Pr. N 20, SCC RAS, 1991, p.20.

OBJECT-ORIENTED LANGUAGE FOR DISTRIBUTED COMPUTATIONS

Dr. Valery A.Torgashev, Igor V.Tsaryov
Computing Structures Laboratory,
Saint Petersburg Computer Science and Automation
Institute of Russian Academy of Sciences
39, 14-ya liniya V.O., Saint Petersburg, 199178, RUSSIA
Email: vat@iias.spb.su

SUMMARY

The paper is devoted to problems of object-oriented programming in massively parallel systems. The original language for dynamic architecture computers (DAC) is offered. The main principles of such computers are described. The hierarchy of basic object classes of the system and some principles of program executing in DAC are considered. Some specific features of the language are discussed.

Keywords: *object-oriented programming languages, parallel processing.*

1. INTRODUCTION

Programming for parallel computers is a sophisticated problem, successful solving of which has a great influence upon efficiency and reliability of computations. The most wide-spread decision is including some means for creating, synchronizing, destructing parallel processes and for their distribution between processing units into traditional programming languages such as C, Pascal, Fortran and others. This method makes the programmer responsible for getting over the main part of difficulties. The programmer must carefully analyze an algorithm for its paralleling, distribute and synchronize parallel branches of the program, think of data transfer and of uniform charging processing units. The result is that the program comes out too complicated, overloaded with statements for organizing parallel processes, too incomprehensible and, as a rule, can't successfully use all hardware resources of computer. Only few classes of problems can be solved efficiently in this way.

A new approach to parallel computations, developed and implemented in Computing Structures Laboratory of The Saint Petersburg Institute for Computer Science and Automation of The Academy of Sciences of Russia, gives a possibility for efficient and convenient solving of this complicated problem by means of a complex of hardware and software decisions called Dynamic Architecture Computers (DAC) [1]. This approach is based on the theory of dynamic automata networks.

A programming language for distributed computations in DAC was offered by Dr. V.A.Torgashev ten years ago [2]. That language was completely adequate to the DAC principles and architecture but was hardly comprehensible and too complicated for implementation and programming. A sufficiently new approach to development of such a language was offered later by I.V.Tsaryov [3]. Works of the latest period have made some revisions of lead to more flexible and convenient programming language with full-scale object-oriented properties.

2. DYNAMIC AUTOMATA NETWORKS

Dynamic automata network (DAN) is a way for representation of the program which doesn't use an algorithmic or functional approach. The program represented as DAN includes a set of objects and links between them and it may be shown as a graph where the objects are nodes of the graph. The behaviour of any object depends on its own properties, its status and status of its environment (neighbour objects linked to this one). In spite of resemblance to data-flow model, this model of computation is quite different from the last one because DAN is not just an operator network, but a network in which operators, data, resources and other object of the program are nodes of the network with equal rights.

Objects may be created automatically while performing the program and any object may be deleted when its function is accomplished. So executing of the program comes to the series of continuous automatic transformations of the configuration and the volume of the network. The end of execution is fixed when the network looses an ability for changing (the number of nodes and their status don't change any more) and the rest of the network (data structure, as a rule) is the result of execution. The result may be also obtained as a full disappearance of the program network with some influence upon the peripheral environment (a side effect of the program).

3. DYNAMIC ARCHITECTURE COMPUTERS

Dynamic Architecture Computer is a virtual environment including massive parallel multiprocessor system and some specific software supporting the DAN model of computations. The principles of DAN can be implemented on any parallel multiprocessor system by means of software only but some hardware features can promote more efficient implementation of it. One of the most important features is availability of rather great number of communicational channels with high throughput between processing units (modules). Connections between processing units are usually organized as a recursive structure [4] in which few processing units in the cluster are connected with each other and the clusters of any level are connected in the same way. Another preference is that the processing unit contains two or more processors specialized for executive, control and communicational functions. Any processing unit has its own memory which may be common for all processors of the unit or partially distributed between processors.

The set of means for organizing parallel computations in DAC includes three main parts such as multiprocessor hardware, special operating system which supports principles of DAN and a special programming language "ЯРД" (pronounced like "yard") for writing programs in a principally new way.

4. TWO LEVELS AND TWO FORMS OF PROGRAMMING

The program in DAC must be represented on two levels of programming. A low level contains a set of programs or procedures written on Assembler language of a processing unit or on any high-level programming language like C, Pascal or other. These

procedures are the methods of objects. There are three types of methods for any object such as executive, communication and control procedures. An executive program realizes a corresponding computational function of any node like matrix or vector operations, spectral transforms such as FFT or any other processing of arrays or structures. Simple scalar operations can't be realized effectively as functions of nodes and must be hidden inside more complicated procedures. Communication methods are responsible for creating and destroying objects (nodes) and subnets and for transferring them to another processing units. Control methods of any node perform the function of interaction between nodes in the network, changing and analyzing status of node. Most of these methods are standard for corresponding type of node and are included in the operating system.

A high level of programming is represented by the object-oriented network programming language "ЯРД" (the Russian abbreviation for Dynamic Extendable Language). It provides the declaration of nodes, their types, structures and functions (methods), forms the network itself, permits binding the nodes with resources and, as a result, defines the parallel execution of the program. The current implementation of the language supports linking separately compiled methods as external routines but further implementations will provide the in-line compiling of Assembler and C procedures and functions.

The result of compiling is a load module which contains the description of the initial configuration of the program network, code of methods for all types of objects and optionally the data initializing sections.

The ЯРД language exists in two forms: a graphic and a text one. The graphic form of language is very convenient and natural for representation of the program. It can be implemented as a special graphic editor giving a possibility of drawing the program network on the screen of computer monitor. The text form of language is completely equivalent to the graphic form but the nodes of a program network and links between them are represented by a set of syntactic constructions which are similar to those ones of traditional languages such as Pascal or Algol-68. The main difference between ЯРД and traditional languages consists in semantics of the syntactic constructions.

5. HIERARCHY OF BASIC OBJECTS

There are seven basic classes of objects such as *operators*, *data*, *references*, *relations*, *resources*, *types* and *structures*. This classification is conditional because any node can contain as well data as programs or procedures (methods) as an object in an object-oriented programming languages. Objects can also change their classes during execution, for instance, operator represented in functional form changes its class to the class and type of result data when its execution is completed.

One of the main properties of objects being nodes of the program network is an ability for changing the network configuration. It can create new nodes and subnets, destroy them or change links between them and it can do it automatically, without any special programmer's actions.

Now let's consider main features of basic classes of nodes.

Operator objects mainly performs actions for computation (data processing). Operators are also nodes which mostly change the configuration and state of the network: create or destroy nodes or subnets or change links between them, change the status of neighbour nodes. Operator nodes may perform computations in the wide range of complexity from simple scalar fixed or floated point arithmetic operations, logical or string functions to big programs with complicated algorithms. In extreme case the whole sequential program may be represented as one operator node. But the level of complexity of an operator is very important for effectiveness of paralleling and executing the whole program. Using too simple operators causes ineffective work of the system because the overhead expenses are very high and using too complicated programs as methods of operators can't provide paralleling of the program and its distribution between processing

units. The optimal level of operators includes vector and matrix operations, data array conversion, fragments of digital image or signal processing and so on. Scalar operations may be represented as separate nodes, if they runs rather seldom and has a great significance and influence on execution control (for instance, calculating bounds and subscripts of arrays or slices).

Data objects can be represented as records or multidimensional arrays and their structure may cause the paralleling of processing only by means of their structural declaration. Readiness of data objects or their components is the main factor which causes activation or disactivation of objects linked to them and so they control process of computation (like in data flow model). Like operators data may have different levels of complexity, from scalar variables to multidimensional arrays of structures. Consideration of complexity of operator nodes may be also repeated for data nodes as well as for other nodes. Multiple using of scalar data nodes decreases performance of the system. But complicated structures or multidimensional arrays promote splitting the program into many fragments by generating nodes and subnets and distributing these computations between processing units.

References provide an access to components or any combination of components of structural nodes or of data structures included into one data node. The most important type of references are quantifiers 'all' or 'any' which causes the application of any action to each component of an object or to any component of an object correspondingly and provides creation of new objects or subnets which are being processed at the same time and on different processing units, if it's possible. References permit to obtain very complicated data structures, slices or selections, the triangle matrix may by the simplest of those. In common case reference may form arbitrary selection from data structure defined by not regular subscript expression but by routine with complicated algorithm.

Relations are the special form of operators which causes processing whenever any node linked with the relation changes its status or value. They have almost the same properties as operators but the different control methods. Relations provide fast reaction of the program network to all exchanges in the environment. For instance, using relations can provide effective implementation for simulating complicated systems such as electronic circuits or for real-time control, measuring or investigation systems.

Resource objects promote binding of an object being processed with hardware units (processors, modules, clusters, subsystems, memory storage units and others) and distributing the parallel processes among units. The only thing programmer must do is setting links between any kind of node or subnet of the program to the corresponding resource node, but he haven't to do it obligatory. Another kind of resource objects provide connection of data or other nodes to various peripheral devices in different ways. There is no any input/output operators in the language, so programmer must only link his data nodes with resource nodes needed and the data will be "pulled in" to the processing unit or "pushed out" to the peripheral device. The desirable data converting and formatting may be also provided by methods of the resource object. Non-standard resource objects may be developed and included in the program by the programmer.

Types are the special kind of objects which provide differentiation of objects of another classes. Type object is an owner of methods library for class and type it represents. A set of methods of any type forms the virtual methods table (VMT) which is an array of pointers to methods. Methods inherited from ancestors are also included in VMT. All methods are virtual. Node of any other class doesn't contain his own copy of methods library or VMT but it has a reference to the corresponding type object. It provides some special features of objects such as full independence on memory allocation of any node and ability for changing class and type of the node during execution of the program.

Structure objects. Any object may be a structural unit which contents a subnet. The structural object can contain objects of any class. If a structural object contains uniform set of objects (e.g. data array or structure) it must be referred to as data (or other) object with structural contents. Structure object must contain objects of different classes. The body of

structural is always an array of references to objects contained. The whole program is also the structure object.

Hierarchy of basic classes of objects is shown at figure 1.

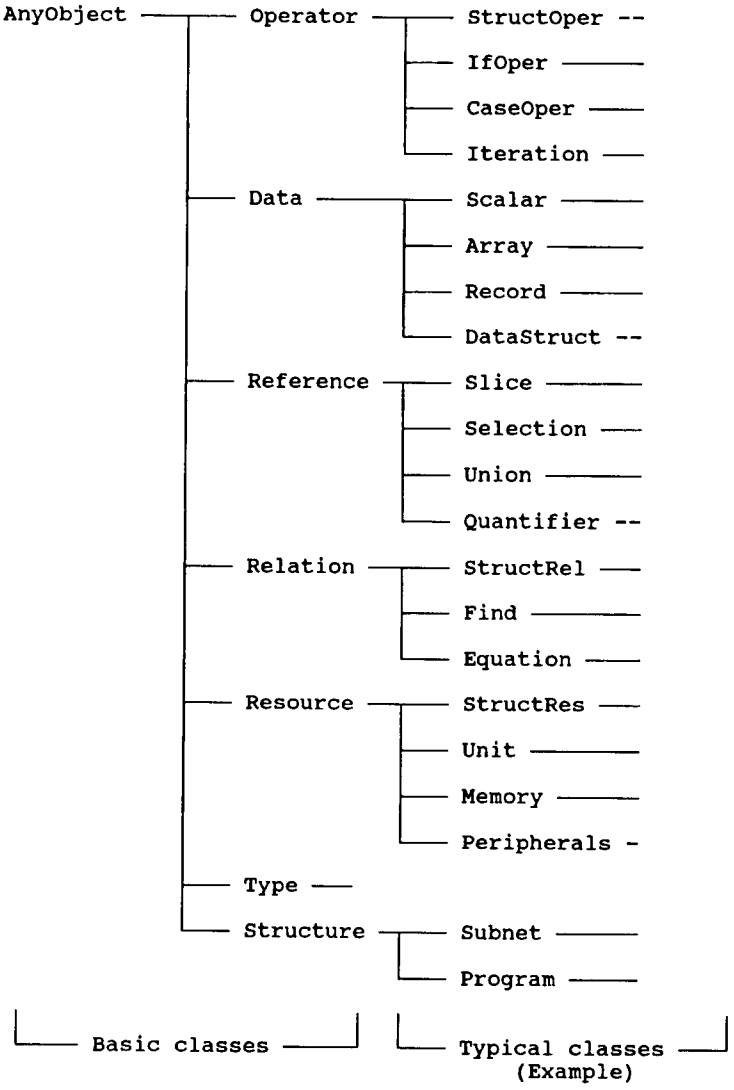


Fig.1. Hierarchy of basic classes of objects

AnyObject is a type object whose library contents the very common methods for all objects we have in users' programs and the operating system. These methods supports basic principles of behaviour of nodes in the network. The second column contents basic classes of objects listed above. All of them inherit properties (system data structures and methods) from AnyObject. These classes are also supported by the operating system. The third column contents an example set of object types, some of them may be supported by the operating system or language libraries.

6. PERFORMING THE PROGRAM

Any node has its own status which points the ability for execution or processing. Execution of a node depends on its own status and the status of nodes (arguments, results or dependents) linked with it. Carrying out the control of nodes processing is completely independent and asynchronous. Synchronizing of parallel processes is being accomplished by node status. The special inspection subsystem of operating system looks after correctness of transforming the program network and prevents from errors like deadlocks, clinches, cycling and infinite generation of nodes. It prevents from massive generating new nodes when there's not enough resources in the system. It also duplicates any information in different processing units that provides a very high reliability of DAC. In the case of malfunction of any processing unit or channel the objects of the program may be redistributed between the remainder of resources so program performance should be continued with minimal delay.

One of the most common construction of the language is the network expression that is syntactically equivalent to an expression or assignment statement. For instance, the statement (expression):

```
A := B * (C + D)
```

is translated to the subnet including four data nodes (A, B, C and D) and two operators (* and +) linked correspondingly to the right order of execution. It must be noticed that the plus ('+') operator in this expression will be converted to the data object representing the result of the operation during execution and then it comes to the '*' operator as data argument. And the result of the '*' operator will be redirected into the body of the data object A. If objects A, B, C and D in the following expression:

```
A [all i, all j] := B [, all j] * (C [all i] + D [all i])
```

are matrixes, represented as arrays of rows, operators '*' and '+' at a moment of execution don't do any computations but create a set of subnets which perform corresponding operation upon rows of matrixes. Matrix B will be transposed, because the first subscript position in the reference (slice) is empty. The named quantifiers 'all i' and 'all j' in references define components of the array which are used in one generated operator. The 'name' of quantifier (i, j in the expression above) may be represented not only by identifiers but also by complicated expressions including ranges, enumeration arrays and other constructions.

If the matrix is rather big and too many subnets are generated, they will be automatically distributed between free (not overloaded) resources of the system (other processing units) in a random way, the data (rows and columns of matrixes) will be automatically transferred to corresponding units following after their descriptors.

When and where will any object perform one of its functions (methods)? It depends not only on the status of an object and its environment, as told above, but also on the class or type of an object and its concrete properties. It also depends on degree of charging of processing units and channels and on the volume of information to be transferred. E.g. an

operator may be activated for execution only if all its arguments (input data) are ready for processing and its results are undefined. And relation is activated always if any change take place in its environment. The input resource is activated whenever data node linked to it is undefined and ready for getting information into its body and so on. Concrete types of objects may behave them differently from their ancestors (e.g. basic objects) if user defined control methods for his type of object. Overriding control and communication methods is a very fine work which requires special knowledge for high-qualified programmer.

And which method is to be performed in concrete situation? Few standard situations provided by operating system and hardware environment are supported by standard set of methods, included into basic set of object classes. They are as following: control and inspection functions, communicative functions, creating and deleting objects, linking and changing links between objects, data conversion and changing class or type of an object, slicing and combining arrays and many others. The operating system itself knows places in VMT, where are references to the methods, corresponding to current situation (different methods for different classes, of course). A set of executive methods for any type of object may be user-defined (except the case of library methods). The link between nodes in the program network may include appointment to the method used (in the form of qualified identifier) and when the object performs its executive function, the pointed method should be executed. If there's no such appointments, the main executive method should be executed (in many cases this is the only executive method of the node). Methods of one object may form chains and ending execution of the method causes the execution of the next method in the chain. E.g. finishing executive method of some operator may cause call methods for changing status of the result, for removing arguments or for changing class of this object.

7. SOME FEATURES OF THE LANGUAGE

The syntax of the language is similar to that one of such traditional languages as Pascal, Algol and partially C. It means that programmer can use most of habitual constructions such as identifiers, constants, symbols of operations and punctuation. Quite big part of reserved words are common too. There is more difference in complex constructions like 'statements', 'declaration' and others. Besides that, the language deliver a high level of flexibility by permitting to define synonyms in the program and by providing access to the lexical and syntactical level of the language (outside the program, by tuning the compiler). But using the last possibility requires high qualification of programmer.

Any expression or statement has unique reflection in the form of a network. So there's no principal difference between expression, statement or closed expression. Blocks or closed expressions (which are enclosed into parentheses or operator brackets **begin...end** or others) are translated into structural objects, if there is a name (identifier) linked with a block.

One of the most important concept in the language and DAN model of computations are links. Every expression, statement or declaration consists of some objects (may be simple objects such as constants) which are linked somehow in the network. Links may present in the program in many forms. Including objects in the expression is one of them, where links are defined by their disposition in the last one. The second form of operator, similar to procedure call, settles links between operator object given by its identifier and argument and result data or reference objects, which are given in the argument list or in the destination of an assignment.

Functional form of operator (like a common call of function), used in the expression, means changing the type (class) of an object after performing computations by an operator to the type and class (data, as a rule) of a result.

Some links may be given by special syntactic marks or reserved words. E.g. the simple declaration like

A, B, C : matrix;

settles links from data objects A, B and C to the **type** object **matrix**, and phrases

A at input; C at output; (A * B) at any unit;

provide links between objects and resources.

The language also includes *'if'* and *'case'* statements which are translated into subnet which contents a subnet for evaluation the Boolean expression, a set of operator nodes and a special operator node which performs a selection of one operator to be executed from the set. Alternative operators are not executed and come to indifferent state.

The language include no loops because an iteration loop such as 'while' or 'repeat' can be performed by recursive generating of nodes and the 'for' loop can be obtained by generating nodes accordingly to complicated references and quantifiers. But there is *'for all'* statement which spreads an action of quantifier to all components of a network inserted. The main programmer's control feature for paralleling computational processes are references which are sufficiently more complicated than that ones in traditional languages.

One powerful feature included into language specially for programming infinite processes such as signal processing, real-time control and other applications. It is a *'stream'* that means a potentially infinite array, the declared size of which defines a 'window' for a current part of stream being processed at the current time interval. A few windows may be opened for one stream simultaneously, if processing of previous window was not finished yet because of some malfunctions, and it happens automatically. Also some means for real-time applications are included in the language, such as support of hardware timers and binding processes (groups of objects) to the moments or intervals of time.

There is no necessity for programmer to think much about paralleling his program in most of applications. It is enough to use correct references (slices, selections) in the program and the system will separate, parallel and distribute corresponding computations dynamically between processing units and memories. Only in special cases programmer must use linking objects to resources or timers.

8. IMPLEMENTATIONS OF THE LANGUAGE

A compiler for text version of this language and an operating system supporting this method of parallel computations are developed in The Laboratory. A graphic version of compiler is being developed now. These programming means are implemented for DAC based on TMS320C30 processors (Texas Instruments) as a cross-compiler working on IBM PC. The compiler for the last revised version of the language is being developed now. Finally, this version will permit including procedure bodies written in C and Assembler into the text of the main program that will help to avoid some problems of linking external subroutines and building relocatable code. The last thing is very important for DAC because programs are distributed between many processing units and they are to be allocated to different places in different units.

References

1. V.A.Torgashev, B.M.Ponomarev, V.U.Plyusnin. Distributed Computations and Dynamic Architecture Computers. Preprint of Leningrad Research Computing Center of Acad. of Sci. of the USSR (LNIVC), Leningrad, 1982. (In Russian).

2. V.A.Torgashev. РЯД - Programming Language for Distributed Computations. Preprint of Leningrad Research Computing Center of Acad. of Sci. of the USSR (LNIVC), Leningrad, 1984. (In Russian).

3. Valery A.Torgashev, Igor V.Tsaryov. A Parallel Programming Language for Dynamic Architecture computers. Proc. of intern. conf. PACT-93, Obninsk, Russia, 1993.

4. Torgashev V.A., Glushkov V.M. and others. Recursive Machines and Computing Technology. IFIP conf. proc., Stockholm, 1974. North-Holland Publ. Co., Amsterdam, 1974.

A data-parallel declarative language for the simulation of large dynamical systems and its compilation

Olivier Michel, Jean-Louis Giavitto, Jean-Paul Sansonnet.

LRI - U.R.A. 410 du CNRS.
Bât 490 - Université Paris Sud
F-91405 Orsay cedex - France
email: michel@lri.lri.fr

Abstract: 81/2 is a declarative data-parallel language designed for the simulation of large dynamical systems. Such simulations are of growing importance and they requires more and more computing power. In consequence, 81/2 introduces a new entity, the web, that combines features of collection-oriented and data-flow language to express data-, stream- and control-parallelism. In this paper, we present the language 81/2, some examples of dynamical systems programmed in 81/2 and we describe the compilation process of a 81/2 programme.

Key-words: data-parallelism, collection-oriented languages, declarative languages, synchronous data-flow languages, simulation of dynamical systems.

I. Introduction

Nowadays, simulation of large dynamical systems represents the majority of supercomputers applications. In this article, a dynamical system refers as any application that modelizes space-time phenomena. Three usual examples are:

- numerical resolution of *partial differential equations* [1] describing continuously evolving systems;
- *discrete event simulation* where the system transitions occur when discrete events happen [2];
- *hybrid systems* simulation which are dynamical systems whose phase space involves continuous and discrete components [3] [4].

A language for the modelisation and simulation of such systems requires the following features:

- being *high-level:* most of the simulation processes are developed using low-level languages leading the designer to spend a lot of time in debugging and tuning its programs [5]. To let the designer concentrate on the modelling aspects, we advocate the use of a high-level language, where the entities expressed are close to the concepts used in the application problem [6] and hiding the low-level implementation details.
- being *parallel:* simulation of large dynamical systems (global atmospheric circulation, molecular dynamics,... [7]) as described in the *Grand Challenge Project* [8] requires a huge amount of computations. Typical requirements can only be satisfied using *massively parallel computers.*

Consequently, we propose 81/2, an experimental high-level parallel language devoted to the simulation of large dynamical systems.

II. The parallel language 81/2

The simulation of a system consists in computing the trajectories of the variables describing the system. By *variable trajectory*, we mean the successive values in time of a variable. This notion fits well with the concept of *stream* [9]. The 81/2 language focuses on models described by a set of functional relationships between stream variables, i.e. an 81/2 program takes the following form:

$$(1) \quad \begin{cases} x = f_1(x, y, \ldots) \\ y = f_2(x, y, \ldots) \\ \ldots \end{cases}$$

where x, y, ... denotes streams. Because relations like (1) often apply between large homogeneous sets of variables, (e.g. for multi-dimensional valued variables or when continuous variables are discretized), the value of a variable at a given time is often an array. These sets of values must be managed as a whole. Moreover the elements of these arrays are distributed among the processors according to the data-parallel paradigm [10], therefore we speak of *collection*. A *stream* of *collections* is named a *web* in 81/2.

II.1. The concept of synchronous stream

[11] have considered, to simplify the formal treatment of a program, that a variable denotes an infinite sequence of values rather than a single value. This approach takes advantage of representing iterations in a « mathematically respectable way » [9] and to quote [12]: « series expressions are to loops as structured control constructs are to gotos ». Such infinite sequences are called *streams* and are manipulated as a whole, using filters, transductors, etc.

Consider the following nested C loops:

```
for (i = 0; i < 3; i++)                i ⇒   < 0;      1;      2 <
  for (j = 0; j < 2; j++)              j ⇒   < 0;  1;  0;  1;  0;  1 <
        k = i+j;                       k ⇒   < 0;  1;  1;  2;  2;  3 <
```

We observe the successive values of the variables i, j and k to build the corresponding streams These streams do not have the same number of elements, although they are issued from the same time interval of observation, because they are related to the change of a memory element (Cf. fig 1). If we consider the element-wise addition of stream i and j (as in Lucid for example), the result is not the stream k.

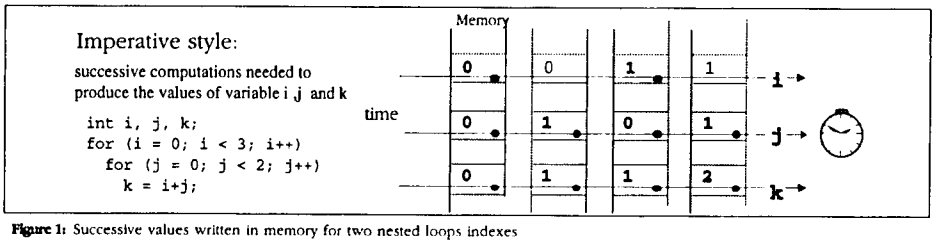


Figure 1: Successive values written in memory for two nested loops indexes

In consequence, we need a more sophisticated data-structure to take into account the relative "tempo" between streams of successive values. A *synchronous stream* introduces a global clock to handle the temporal succession. An instant of this clock is named a *tick*. The instants where the stream possibly changes its value are called the *tocks* of the stream. Synchronous streams are very different from a linear storage of data and the values of a synchronous stream vanish and leave room to the next values. At some tick that is not a tock, the stream value can be used and its value is the value of the previous tock, if it exists, else it is undefined. Here we consider 4 classes of operators to build a stream (Cf. the table 1):

- *Stream constant*. Scalar constants are promoted to stream constant with only one tock at tick 0. In addition, we use the constant `Clock n`, which always holds a true value with tocks randomly distributed with a density of `1/n` over the ticks.
- *Extension of a scalar operator*. Let ⊕ a scalar binary operator (i.e. an operator defined on the values of a stream), we define the stream x⊕y from the streams x and y. The expression x⊕y has a tock (i.e. is susceptible to change its value) if x has a tock, *or* y has a tock, *as soon as* both x and y are defined. The value of x⊕y at some tock is the result of the combination by ⊕ of the values of x and y at the corresponding tick (that is, ⊕ acts in an element-wise fashion on the stream elements).

- *Delay.* The operator $ is used to shift an entire stream giving access to the previous value. The tock of $x are the same of the tock of x, *except* for the first one. The value of $x at some tock is the value of x at the previous tock.
- *Sampling operator.* We present only one sampling operator: x when y. The values of x when y are those of x sampled at the tock where x takes a true value, as soon as x and y are defined.

Table 1: Examples of stream expressions. A column in this table represents a tick. A non empty column is a tock.

input	i	0		1		2	
streams	j	0	1	0	1	0	1
	i+j	0	1	1	2	2	3
output	$i			0		1	
streams	1	1					
	i==1	false		true		false	
	j when (i==1)			1			

A declarative synchronous stream programme is a set of equations: « *variable = stream expression* ». In addition, stream equation may be quantified by some temporal predicates stating its validity domain. For example,

```
{ cpt@0 = 33, cpt = $cpt + 1 when Clock 1 }
```

is a set of two equations defining the stream cpt. The equation quantified by « @0 » defines the first *tock* of cpt (tocks are numbered from 0). The other equation, without quantification, specifies the default definition of cpt. Cpt is a counter starting at 33, and increasing by 1 at the rate of Clock 1.

II.2. The concept of collection

A *collection* is a data structure that represents a set of elements *as a whole* [13]. Several kinds of aggregation structure exist: *set* in the SETL [14], *list* in LISP, *tuple* in SQL, *pvar* in *LISP [15] or even *finite discrete space* in Cellular Automata [16]. Data-parallelism is naturally expressed in terms of collections [17], [10]. From the point of view of the parallel implementation, the elements of a collection are distributed over the processing elements (PEs).

We consider here collections that are *ordered* sets of elements. Four kinds of function application can be defined on such data:

```
apply   : f(c1, …, cp)   : (collection^p → X) × collection^p → X
alpha ^: f^(c1, …, cp)  : (scalar^p → scalar) × collection^p →  collection
beta  \: f\c             : (scalar^2 → scalar) × collection →  scalar
scan \\: f\\c            : (scalar^2 → scalar) × collection →  collection
```

X means both scalar or collection; p is the arity of the functional parameter f

The first function application mechanism is the standard one: collections are considered as a whole and function application acts as usual. The second type of function application produces a collection whose elements are the "pointwise" application of the function to the elements of the arguments. Then, using a scalar addition, we obtain an addition between collections. The process of promoting a scalar to a collection function is known as an *alpha-denotation* in APL. The third type of function application is the *beta-reduction.* Beta-reduction of a collection using the binary scalar addition, results in the summation of all the elements of the collection. Any associative binary operation can be used e.g. a beta-reduction with the *min* function gives the minimal element of a collection. The scan application mode is similar to the beta-reduction but returns the collection of all partial results. See [18] for a programming style based on scan. Beta-reduction and scan can be performed in $O(\log_2(n))$ steps on SIMD architecture, where n is the number of elements in the collection, if there is enough PEs.

Some additional operators are defined. An element of a collection, also called a *point* in 8½, is accessed through an index. The expression T.n where n is an integer, is a collection with one point; the value of this

point is the value of the n^{th} point of T (point numbering begins with 0). The *if* operator extends the conditional construct to collection:

```
Q = if B then T else F fi
```

must be taken pointwise in space, that is, for each point n: `Q.n = if B.n then T.n else F.n fi`.

Geometric operators change the *geometry* of a collection, i.e. its structure. The geometry of a scalar collection is reduced to its *cardinal* (the number of its points). A collection can also be *nested*: the value of a point is a collection. Collection nesting allows multiple levels of parallelism and exists for example in ParalationLisp [19] and NESL [20]. The geometry of the collection is the hierarchical structure of point values. The first geometric operation consists in *packing* some webs together. The result is a *system*:

```
T = (a, b)
```

In the previous definition, a and b are collections resulting in a nested collection T. Elements of a system may also be named, achieving the idea of environment (a binding between names and values). Assuming

```
car = { velocity = 5, consumption = 10 }
```

The points of this collection can be reached through the dot construct using uniformly their label, e.g. `car.velocity`, or their index: `car.0`. The *composition* operator concatenates the values and merges environment. The following example concatenates the two collections A and B:

```
A = (a, b);  B = (c, d);  A # B ⇒ (a, b, c, d)
ferarri = car # (color = red) ⇒ (velocity = 5, consumption = 10, color = red)
```

The last geometric operator we will present here is the *selection*: it allows to select some point values to build another collection. For example:

```
Source = (a, b, c, d, e)
target = (1, 3, (0, 4))
Source(target) ⇒ (b, c, (a, e))
```

The notation `Source(target)` must be understood in the following way: a collection can be viewed as a function from [0..n] to some co-domain. Therefore, the dot operation corresponds to function application. If the co-domain is set of natural numbers, collections can be composed and the following property holds: `Source(target).i = Source(target.i)`, mimicking the function composition definition. From the parallel implementation point of view, selection corresponds to a gather operation and is implemented using communication primitives on a distributed memory architecture.

III. Examples of 81/2 programmes

III.a. Numerical resolution of a parabolic partial differential equation

We want to simulate the diffusion of heat in a thin uniform rod. Both extremities of the rod are held to 0 °C. The equation of the diffusion takes the following form:

$$\frac{\partial u}{\partial t} = k \cdot \frac{\partial^2 u}{\partial x^2}$$

which is discretized as [1]:

$$\frac{U_{i,j+1} - U_{i,j}}{k} = \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h^2}$$

The corresponding 81/2 programme is:

```
start = some initial temperature distribution;
begin = 0;
end   = 0;

U@0 = start;
U   = begin # inside # end;

float inside = 0.4*left(pU) + 0.2*middle(pU) + 0.4*right(pU);
pU           = $U when Clock;
```

The $ (resp. `left`, `middle` and `right`) operator represents a shift in the past (resp. in space). The # operator represents the concatenation; when is a sampling operator and `clock` the time discretization (Cf. Figure 2).

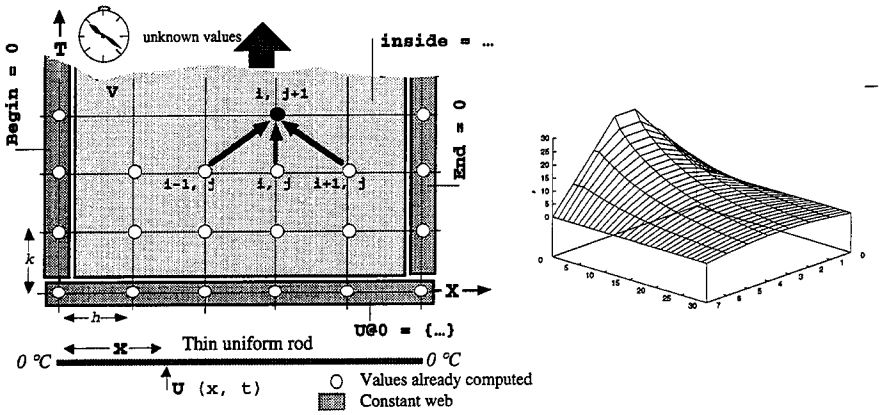


Figure 2: diffusion of heat in a thin uniform rod

III.b. The simulation of a reactive system

Here is an example of an hybrid dynamical system, a "wlumf" which is a "creature" whose behaviour (eating) is triggered by the level of some continuous internal state (Cf. [21] for such model in ethological simulation):

```
System environment = { boolean FOOD = Random };;
System wlumf = { i_am_hungry@0 = false;
                 i_am_hungry   = (glucose < 3);

                 glycaemia@0 = 6;
                 glycaemia   = if i_am_eating
                                  then 10
                                  else max(0, $glucose - 1) when Clock
                               fi;

                 i_am_eating@0 = false;
                 i_am_eating   = $i_am_hungry && environment.FOOD };;`
```

The result of an execution is given in Figure 3.

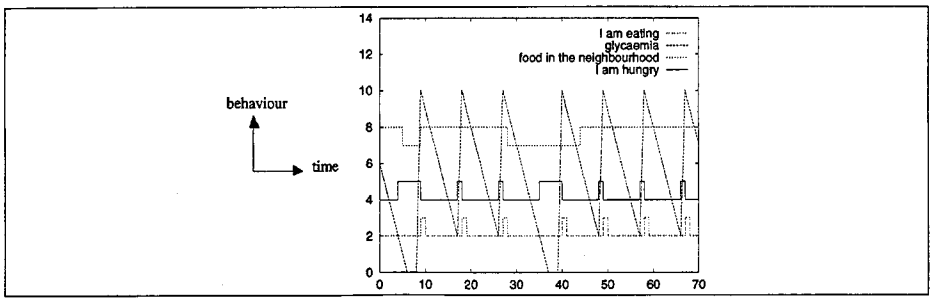


Figure 3: behaviour of an hybrid dynamical system.

III.c. Iterations of the logistic map

The logistic equation is: `f (x) = k*x*(1-x)`. The following 81/2 programme computes a map composed by the concatenation of iterations of the logistic map for parameters `k` starting from 1.2 and step 0.1, and for a discretization `x` of the initial interval [0, 1]:

```
start = 0.05 * '20;
k@0   = 1.2;
k     = $k + 0.1 when Clock;
separate[5] = 0;
map = ((start # separate) # k*map*(1-map)):[125];
```

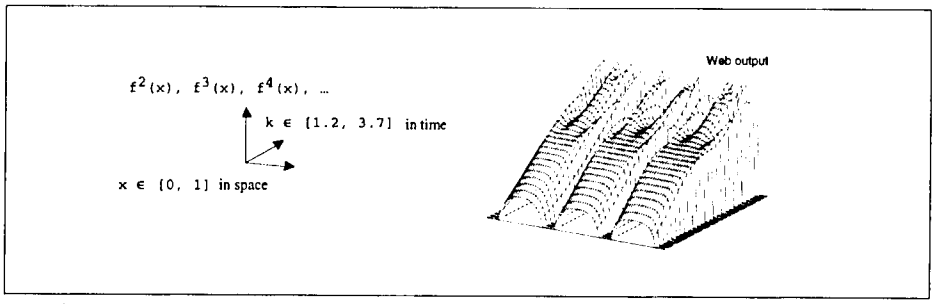


Figure 4: iteration of the logistic equation

IV. Compilation

The compilation process is composed of four phases :

- The *type inference* and *analysis* computes the shape of the collections involved in a programme [22].
- The *dependency analysis* builds a partial evaluation order between the expressions. This step is trivial for scalar data-flow programmes but not obvious when dealing with array. For example, the programme `a = 0 # (a + 1): [10]` is a recursive definition of an array whose ith element has the value i. The expressions involved by this definition cannot be evaluated in parallel and must be sequentially processed.
- The *clock computation:* 81/2 advocates a static execution model. That is, the triggering of parallel activities is inferred at compile-time rather than at run-time. This is done by synthesising for each expression, through an abstract interpretation, a predicate *Clock(t)* which indicates at time t, whether the evaluation has to be done or not. The static evaluation model answers the drawbacks sustained by Gajsky [23] against the standard dynamic data- or demand- driven execution models.
- The *mapping* and *scheduling:* the mapping and scheduling phase of the compilation consists in distributing the evaluation task over the parallel processors. The clock predicates ensure an iterative scheduling of the tasks [24].

IV.1. The clock calculus

The computation of a web value in a 81/2 programme corresponds to the computation of every successive collections that are solution. The value of a stream at a given time is a collection (of scalar values) named *instantaneous value* of the web.

The *clock* of a web `X` is a boolean stream holding the value `true` if tick is a tock for the web X, else holding a `false` value. Each tick is a tock for a clock. The clock calculus of a web is needed to decide whether the computation of an instantaneous value has to take place at some tick or not. Let `x` denotes the

instantaneous value of X, and `clock(X)` the instantaneous value of the clock associated to X. Every definition

```
X = f(Y)
```

in the initial programme, is translated in:

```
x = if clock(X) then f(y)                                    (1)
```

This expression is synthetised by induction on the structure of the definition of X. For example:

```
clock (A when B) = b ∧ clock(B)
```

This transformation produces a normal form from the original web definition. Roughly, the compiler will generate for any expression of the programme, a task executing the process shown in (1). It is still necessary to compute the dependency between the tasks to determine their relative order of activation.

IV.2. The code generation

Three different kinds of code generation have been thought of and are to be generated:

- code for a virtual SIMD machine written in C very close to CVL [25] and adaptated to the execution on a SIMD or vectorial architecture;
- a sequential standalone C code using no dynamical memory allocation nor function-call stack;
- a full MIMD code.

At the moment, the compiler written in C [26] and in an ML dialect [27], generates a code for a virtual SIMD machine implemented on a UNIX workstation and the C code. However, all the compiler phases assume a full MIMD target architecture and we are working on the MIMD code generation.

V. Related works and conclusions

V.1. Related works

The transformation of stream expressions into loops is extensively studied in [12]. The expressions considered do not allow recursive definitions of streams. Our proposition handles this important extension as well as "off-line cycle" expressions. We share however the preorder restriction, i.e. the succession of stream elements must be processed in time ascending order (this is not the case in a language like LUCID). We focus also on unbounded stream and therefore we do not consider operations like *catenating* (Cf. [12]).

[24] consider the static scheduling of a class of data-flow graphs used in digital signal processing. The translation of a (recursive) stream definition into a (cyclic) data-flow graph is straightforward. Their propositions apply but are limited to the subset of "on-line" program [28]. This restriction excludes the sampling operator presented above (an operator similar to the T-gate of [29]) and requires the presence of at least one delay on each cycle of the data-flow graph.

Compiled execution model for data-flow are also investigated in the domain of real-time programming LUSTRE [30], SIGNAL [31] and in the domain of signal processing [32]. The reactive languages LUSTRE and SIGNAL include the full set of stream operators we use and allow recursive definitions. However in real-time programming, the interests are focussed on the synchronization constraints: the handling of time is quite different and data-parallelism is out of considerations. They consider only the compilation of "strongly synchronous" programs [33] which correspond to the combination of streams "progressing at the same rate" or "with the same clock": a property called "isochrony". In accordance with the specificity of their applications domain, real-time programming, LUSTRE and SIGNAL do not allow "heterochronous" expressions and sophisticated techniques [34] have been developed to check and reject heterochronous programs. Similar approaches exists in the field of systolic programming: Crystal [35], ALPHA [36] or for asynchronous cellular array [37]. But this approaches lack the concept of collection (element accesses are explicit).

V.2. Conclusions

The 81/2 language is an experimental high-level parallel language combining the advantage of the SIMD and MIMD execution model through the embedding of collections in a synchronous and static data-flow scheme resulting in an hybrid SPMD or MSIMD execution model. (see a companion paper [38] in this conference for the mapping and scheduling of data-parallel dataflow tasks on MIMD architectures) Using data- and implicit control- parallelism enables:

- the maximal expression of the parallelism inherent to an application;
- the effective use of the parallelism which implies the less implementation overhead (with respect to the target architecture);
- the hiding of communication costs by the computation of (parallel) independent activities.

All the examples stated in this paper have been processed by the existing compiler and the data visualisation done through Gnuplot [39]. We are currently working on the generation of MIMD code and the extension of the language to handle dynamically shaped webs.

VI. References

[1] G. D. Smith, Numerical solution of partial differential equations: finite difference methods, third edition, 1985, Oxford Applied Mathematics and Computing science series, Oxford University Press.

[2] R.Righter, J.Walrand, *Distributed simulation of discrete event systems*, Proc. of the IEEE, vol 77, n°1, Jan.1989.

[3] A. Benveniste, M. Le Borgne, P. Le Guernic, *Hybrid Systems: the SIGNAL approach*, LNCS 736, Springer-Verlag, 1993.

[4] A. Back, J. Gluckenheimer, M. Myers, *A Dynamical Simulation Facility for Hybrid Systems*, LNCS 736, Springer-Verlag, 1993.

[5] P. Fritzson, N. Andersson, *Generating Parallel Code from the Equations in the ObjectMath Programming Environments*, LNCS 734, Springer-Verlag, 1993.

[6] E. V. Zima, *Recurrent Relations and Speed-up of Computations using Computer Algebra Systems*, LNCS 721, Springer-Verlag 1992.

[7] G. S. Almasi, A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, 1989.

[8] *Grand Challenges - HPCC*, U.S. Off. of Sci. and Tech. Policy's Committee on Phy., Math., and Eng. Sci., 1991.

[9] W. W. Wadge and E.A. Ashcroft, *Lucid, the Data flow Programming Language*, Academic Press U.K., 1985.

[10] J. M. Sipelstein, G. E. Blelloch, *Collection-oriented languages*, proc. of the IEEE, Vol. 79, N° 4, april 1991.

[11] L.G. Tesler, H.J. Enea, *A language design for concurrent processes*, AFIPS Conference Proceedings, vol. 32, pp 403-408, 1968

[12] Richard C. Waters, *Automatic Transformation of Series Expressions into Loops*, ACM Trans. on programming Languages and Systems, Vol. 13, N°1, Jannuary 1991, pp 52-98.

[13] G. E. Blelloch and G. W. Sabot, *Compiling Collection-oriented Languages onto Massively Parallel Computers*, Journal of Parallel and Distributed Computing, 8, 119-134 (1990)

[14] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, Programming with sets: An introduction to SETL, Springer-Verlag 1986.

[15] Thinking Machine Corporation, *The Essential *Lisp Manual*, Cambridge MA, 1986

[16] Tommaso Tofooli and Norman Margolus, *Cellular Automata Machine*, The MIT Press, Cambridge MA, 1987

[17] W. Daniel Hillis and Guy L. Steele JR, *Data Parallel Algorithms*, Communication of the ACM, vol. 29 N°12, December 1986.

[18] G. E. Blelloch, *Scans primitive Parallel Operations*, IEEE Transactions on Computers, Vol. 38, N° 11, November 1989.

[19] G. W. Sabot, *The Paralation Model: Architecture*, Independent Parallel Programming, MIT Press, Cambridge MA, 1988

[20] G. E. Blelloch, *NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129*, School of Computer Science, Carnegie Mellon University, April 1993.

[21] P. Maes, *A bottom-up mechanism for behavior selection in an artificial creature*, proc. of the first int. conf. on simulation of adaptative behavior, Bradford Book, MIT Press, 1991.

[22] J.-L. Giavitto, *Typing geometry of Homogeneous Collection*, Proc. of the 2nd Int. Workshop on array, ATABLE-92, Montréal, Jun 1992.

[23] D.D. Gajski, D.A. Padua and D.J. Kuck, R. H. Kuhn, *A second opinion on data flow machines and languages*, IEEE Computer, feb 1982.

[24] K. K. Parhi, D. G. Messerschmitt, *Static rate-optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding*, IEEE Trans. on Comp., Vol 40, N°2, February 1991.

[25] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, M. Zagha, *CVL: A C Vector library. Technical Report CMU-CS-93-114*, School of Computer Science, Carnegie Mellon University, 1993.

[26] B. W. Kernighan, D. M. Ritchie, *The C Programming language*, Englewood Cliffs, NJ, Prentice Hall 1978.

[27] X. Leroy, *The Caml Light system release 0.6 - Documentation and users manual*, INRIA, Sep 1993.

[28] A. Aho, J. Hopcroft, J. Ullman, The design and analysis of computer algorithms, Addison-Wesley, 1974.

[29] Jack B. Dennis, *First Version of a Data Flow Procedure Language*, proc. of the Programming Symposium, Paris, April 9-11, 1974, LNCS 19, Springer.

[30] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, *Lustre: a declarative language for programming synchronous systems*, in 14th ACM Symposium on Principles of Programming Languages, january 1987.

[31] P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier, *SIGNAL, a dataflow oriented language for signal processing*, IEEE-ASSP, 34(2):362-374, 1986.

[32] Edward A. Lee, David G. Messerschmitt, *Static Scheduling of Synchronous Dataflow programs for Digital Signal Processing*, IEEE Trans. on Comp., Vol C-36, N°1, January 1987.

[33] G. Berry, A. Benveniste, *The synchronous approach to reactive and real-time systems*, Proc. of the IEEE, vol. 79, n° 9, September 1991.

[34] A. Benveniste, P. Le Guernic, *Hybrid dynamical systems theory and the SIGNAL language*, IEEE transactions on Automatic Control, vol. 35, n° 5, May 1990.

[35] M. C. Chen, *A Parallel Language and its compilation to multiprocessor machines or VLSI*, POPL'86, Florida, pp 131-139.

[36] C. Mauras, *Definition of Alpha: a language for Systolic Programmation*, INRIA research report N° 482, Jun 1989.

[37] R. Cornu-Emieux, G. Mazaré, P. Objois, *A VLSI asynchronous cellular array to accelerate logical simulations*, proc. of the 30th. Midwest Internationnal Symposium on Circuit and Systems, 1987.

[38] A. Mahiout, J.L. Giavitto, J.P. Sansonnet, *Distribution and scheduling data-parallel dataflow programs on massively parallel architectures*, Software for Multiprocessors and Supercomputers, September 21-23, 1994, Moscow

[39] T. Williams, C. Kelley, (C) GNUPLOT An Interactive Plotting Program, reference manual, Free Software Foundation, 1993

Compiling a producer-consumer LEQ system in a network of communicating processes.

Marie-Christine Eglin-Leclerc — *IUT - Roanne*
Jacques Julliand — *Université de Franche-Comté*
Laboratoire d'Informatique, Université de Franche-Comté
25030 Besançon cedex France
Tél : +33 81.66.64.51 Fax : +33 81.66.64.50
E-mail : julliand@univ-fcomte.fr

Abstract - To avoid considerations linked to the implementation the first solution of a problem is written in a functional way, like a system of recurrent equations. With the aim to do an asynchronous interpretation this equational system is transformed into a set of systems which contain three sub-systems : a producer one and a consumer one linked by a communication medium. This new system is next traduced into a network of sequential communicating OCCAM processes. This paper presents in details this last step.

Keywords - Compilation, Asynchronous Interpretation, Language of recurrent equations, Sequential processes, Parallelism.

1- INTRODUCTION

Lots of new research activities in the field of parallel architectures are realized. We can talk about the techniques of parallel program constructions where we can distinguish the two following sorts of works :

- the derivation techniques which are independent of target architectures ([1], [2], [3], etc),
- the compilation techniques which are automatic or assisted transformations([4], [5], [6], etc).

Although the target architectures are different these last named works have the same interpretation domain. In fact all these ones have a synchronous interpretation. In the asynchronous interpretation domain it seems that the design tools, the parallelization tools and the evaluation or debugging tools are still too few [7]. Our study tries then to contribute to the enrichment of the knowledge in this domain. We define :

- a functional language which allows to describe problems like equational systems,
- an asynchronous interpretation for MIMD machines.

Equational languages are classical (cf Lucid[8] and Lustre[9]), it is the same thing for their synchronous interpretation : it interprets the recurrence indice like the time, relatively to a global clock. The k^{th} occurrence of a variable x is calculated at the instant k. Giving an asynchronous interpretation means that only the order which is defined by the dependences is conserved. To reach an execution DMPC (Distributed Memory Parallel Computers), an operational point of view consists to associate a network of producers and consumers to each equational system. Each producer (called calculation) is linked by a system of messages communication (called communication) to a consumer (also called calculation). To define a such network the data dependence graph associated to the equational system is progressively and systematically transformed [10].

Then the transformed system can be interpreted in an asynchronous way. It can be considered like a generalized "network of functions" [11] and it can be implemented (traduced) like a network of communicating processes written in a concrete language such as OCCAM ([12]).

This paper presents in details the proposed language of recurrent equations, its denotational semantics and the techniques which traduce the equational system (after transformation), into an asynchronous network of communicating processes.

2- PRESENTATION OF THE LANGUAGE

This language called LEQ (Language of **EQ**uations) is purely functional [13]. A program (called "system") is a set of recurrent equations which define result variables and intermediate variables from data.

2-1- Syntactical presentation

The syntax of LEQ may be seen as a subset of the LUSTRE syntax :

- A variable is denoted by an identifier, as x, y, z,...
- To simplify the presentation we limit ourself to integer and boolean variables.
- An equation «x = e» means that the expression e and the variable x are identical, i.e. can be substituted anywhere one for the other. Example : The system : $\begin{cases} x = y + z \\ t = 2 * x \end{cases}$

  where y and z are given, is equivalent to the equation t = 2 * (y + z).
- Expressions can be built from variables or constants, classical arithmetical or boolean operators (+, *, ..., **or**, **and**, ...) and operators which are specific to the data flows.

These last operators are the following :

- ° if e is an expression, «**pre** e» is an expression,
- ° if e and e' are two expressions, «e' → e» is an expression,
- ° if p is a boolean expression and e is an expression, «**when**(p,e)» is an expression,

Example : p = false → **not** (**pre** p)
y = **when**(p,x)

The operator **when** is named extraction operator; → and **pre** are named recurrence operators. The semantics of these specific operators is rather delicate to understand, they will be explicited in the next paragraph.

Closing this paragraph we can give a summary of the concrete syntax of LEQ as :

PROG	::=	{SYST}	*-- a program is a set of sub-systems*
SYST	::=	{EQU}	*-- a sub-system is a set of equations*
EQU	::=	IDS = EXP	*-- IDS is an identifier or a pair of identifiers*
EXP	::=	**when**(EXP, EXP) | **if** EXP **then** EXP **else** EXP **endif** | EXP_AB	
EXP_AB	::=	EXP_AB + EXP_AB | EXP_AB **and** EXP_AB | ... | **pre** EXP_AB |	
		EXP_AB → EXP_AB | ... *-- EXP_AB is an arithmetic or boolean expression*	
IDS	::=	ID , IDS | ID	
...			

2-2- Denotational semantics

To address asynchronism we define the semantics of our language LEQ, by introducing temporal sequences and temporal domains. A discrete modelization of the time is given by $\mathbb{N}$: at each occurrence of a sequence its value and its instant in $\mathbb{N}$ are associated. The semantics of a object LEQ is then a mapping from $\mathbb{N}$ (the indices) to $\mathbb{N} \times T$ ($\mathbb{N}$ represents the time and T the type of values), i.e by curryfing, a mapping of the form : $\mathbb{N}$ -> $\mathbb{N}$ -> T. Thus the semantics is given in terms of temporal domains and value sequences.

Definition 1 : We call temporal domain any increasing mapping d from $\mathbb{N}$ to $\mathbb{N}$.

Notations : $\tilde{d} = \{ t \in \mathbb{N} / \exists k \in \mathbb{N}, t = d[k] \}$

2-2-1- Semantics of a variable

Definition 2 : The semantics of a variable x of type T in LEQ is a mapping from $\mathbb{N}$ to T defined by $[\![x]\!] = \chi$ o d where d is a temporal domain and χ is a sequence of values in T.

So, for any index k in $\mathbb{N}$, $[\![x]\!][k] = \chi[d[k]]$ or $[\![x]\!][k] = \chi[t]$ with $t = d[k]$.

Example : Let d : (0,1,2,...) -> (0,2,4,...) and χ : (0,1,2,...) -> (a,b,c,e,f,...)
x a variable whose temporal domain is d and whose sequence of values is χ.
x is defined as $[\![x]\!][0] = \chi[d[0]] = \chi[0] = a$; $[\![x]\!][1] = \chi[d[1]] = \chi[2] = c$
$[\![x]\!][2] = \chi[d[2]] = \chi[4] = f$ ...
The figure 1 represents this variable as a temporal axis.

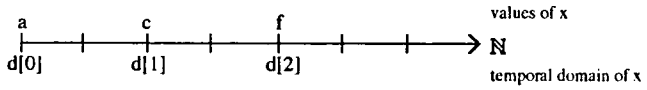


Figure 1 : Variable as a temporal axis.

2-2-4- Semantics of expressions

We classify operators of the language LEQ in two classes : the operators which do not change temporal domains, and the operator "when" which modifies the temporal domain.

2-2-4-1- The first class of operators may be divided in two sub-classes (cf Alpha [14]) :
- immovable operators which compose variables component by component,
- spatial operators which allow definitions of variables by recurrence.

Immovable operators - They are arithmetical or boolean operators extended to sequences. They operate occurrence by occurrence on variables which must have identical temporal domains.

Definition 3 : The semantics of an expression «e **bop** e'» built from two arguments e and e' of type T, with identical temporal domains d, is a mapping from $\mathbb{N}$ to T defined as : $[\![e \textbf{ bop } e']\!] = \xi$ o d
where - χ is the sequence of values of e, ψ is the sequence of values of e'; and
- ξ is a sequence of values in T defined as : $\xi[t] = \chi[t]$ bop $\psi[t]$, $\forall t \in d$

Example : Let d : $\mathbb{N}$ -> (0,2,4,...), the temporal domain of e and e',
χ : $\mathbb{N}$ -> (true,false,true,false,false,...), the sequence of values of e,
ψ : $\mathbb{N}$ -> (false,true,false,true,false...), the sequence of values of e',
then $[\![e \textbf{ or } e']\!][0] = \chi[0]$ or $\psi[0]$ = true; $[\![e \textbf{ or } e']\!][1] = \chi[2]$ or $\psi[2]$ = true
$[\![e \textbf{ or } e']\!][2] = \chi[4]$ or $\psi[4]$ = false; ...

Spatial operators - We define two spatial operators : the dart operator and the preceding one.

• The dart operator, noted →, allows the initialization of variables. «e → e'» is identical to the argument e' except for its first occurrence which takes the value of the first occurrence of e.

Definition 4 : The semantics of an expression «e → e'» built from two arguments e and e' of type T, with identical domains d, is a mapping from $\mathbb{N}$ to T defined as : $[\![e \rightarrow e']\!] = \xi \circ d$

such that $[\![e \rightarrow e']\!][0] = [\![e]\!][0] = \chi[d[0]] = \xi[d[0]]$

$[\![e \rightarrow e']\!][k] = [\![e']\!][k] = \psi[d[k]] = \xi[d[k]] \quad \forall k \in \mathbb{N} / k > 0$

where χ is the sequence of values of e, and ψ is the sequence of values of e'.

• The preceding operator, noted **pre**, allows to express recurrent definitions. For each occurrence of its parameter, it returns the value of the previous one.

Definition 5 : The semantics of an expression «**pre** e» built from an argument e of type T, is a mapping from $\mathbb{N}$ to T defined as : $[\![\textbf{pre } e]\!] = \xi \circ d$

such that $[\![\textbf{pre } e]\!][0] = \xi[d[0]] = nil$ and $[\![\textbf{pre } e]\!][k] = \xi[d[k]] = [\![e]\!][k-1] = \psi[d[k-1]] \quad \forall k \in \mathbb{N} / k > 0$

where nil is an undefined value, d is the temporal domain of e, and ψ is the sequence of values of e.

Example : Let d : $\mathbb{N}$ -> (0, 2, 4, ...) the temporal domain of e and e',

χ : $\mathbb{N}$ -> (0, 1, 2, ...) and ψ : $\mathbb{N}$ ->(1, 3, 5, ...) the sequences of value of e and e',

then the semantics of «**pre** e'» is : $[\![\textbf{pre } e']\!][0] = nil$

$[\![\textbf{pre } e']\!][1] = [\![e']\!][0] = \psi od[0] = \psi[0] = 1$; $[\![\textbf{pre } e']\!][2] = [\![e']\!][1] = \psi od[1] = \psi[2] = 5$; ...

and the semantics of «e → (**pre** e')» is :

$[\![e \rightarrow (\textbf{pre } e')]\!][0] = \chi od[0] = \chi[0] = 0$; $[\![e \rightarrow (\textbf{pre } e')]\!][1] = [\![\textbf{pre } e']\!][1] = 1$

$[\![e \rightarrow (\textbf{pre } e')]\!][2] = [\![\textbf{pre } e']\!][2] = 5$; ...

2-2-4-2- The operator when

It modifies temporal domains : the temporal domain d' of an expression «**when**(p,e)» is a "sub-domain" of the domain d of its operands, i.e. $\tilde{d'} \subset \tilde{d}$. It allows the extraction of a sub-sequence from a sequence having the same type.

Definition 6 : The semantics of an expression «**when**(p,e)» built from p of type boolean and e of type T (p and e having the same temporal domain d) is a mapping from $\mathbb{N}$ to T defined as : $[\![\textbf{when}(p,e)]\!] = \chi \circ d'$ where $\tilde{d'} = \{ t \in \mathbb{N} / \exists k \in \mathbb{N}, t = d[k] \wedge \pi[t] = true \}$ with π the sequence of values of p, and χ is the sequence of values of e.

Example : Let d : $\mathbb{N}$ -> (0, 2, 4, ...) the temporal domain of e and p,

π : $\mathbb{N}$ -> (true, true, false, false, true, ...) the sequence of values of p,

χ : $\mathbb{N}$ -> (a, b, c, d, f, ...) the sequence of values of e,

then $\tilde{d'} = \{ 0,4,... \}$ and, d' : $\mathbb{N}$ -> (0, 4, ...)

because $\pi od[0] = \pi[0] = true$, $\pi od[1] = \pi[2] = false$, $\pi od[2] = \pi[4] = true$, ...

and the semantics of **when**(p,e) is: $[\![\textbf{when}(p,e)]\!][0] = \chi od'[0] = \chi[0] = a$

$[\![\textbf{when}(p,e)]\!][1] = \chi od'[1] = \chi[4] = f$ ...

2-3- Example

We consider the example of an axle-counter [9]. A part of a railway regulation system, schematized in figure 2, must perform the following function : a track is divided into two districts. Two pedals Pl and Pr are set at the boarder between two districts, overlapping each other as following :

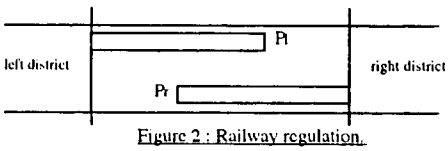


Figure 2 : Railway regulation.

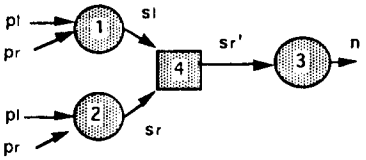


Figure 3 : Data dependence graph.

The device knows the states of the pedals, from two boolean variables pl and pr (pi is true whenever there is a wheel on the pedal Pi). It must compute two boolean variables sl and sr (si is true when and only when an axle enters in the district i) and the result n (number of axles which have gone from the left district to the right one, reduced by the number of axles that have crossed the boundary in the reverse direction).

After transformation the initial equational LEQ system is such that the associated data dependence graph [15] is defined by particular schemes : these schemes have a type producer-consumer via a communiation. Then the axle-counter problem is written as follows :

{1} sl = **false** → (**not** pl) **and** (**pre** pl) **and** (**not** pr). -- an axle enters in the left district

{2} sr = **false** → (**not** pr) **and** (**pre** pr) **and** (**not** pl). -- an axle enters in the right district

{3} n = 0 → **if** sr' **then** (**pre** n) + 1 **else** (**pre** n) - 1 **endif**.

{4} sr' = **when**(sl **or** sr,sr); -- = *valuated if an axle come in a district*

The data dependence graph associated with this system is presented in figure 3 : nodes represent equations and oriented edges represent data transmissions. Note that the nodes which are considered as a calculation are drawn as a circle. The ones considered as a communication are drawn as a rectangle.

3- TO TRANSFORM A LEQ SYSTEM IN AN OCCAM NETWORK

We choose to express a network in the OCCAM syntax as the following feature [16] :

OCCAM program :
{declaration of one channel for each edge of the graph
and declaration of associated protocoles}
{declaration of one procedure for each node}
{instruction to put into parallel all the procedures}

All the informations which are useful to create a such program are given by the equational system, its data dependence graph and its type environment.

3-1- OCCAM channels definition

We distinguish two types of channels :
- The "external" ones allow to read the data and to write the results.
- The "internal" channels allow the transmissions of values between OCCAM procedures.

3-1-1- To create an "external" channel

By convention, we call "keyboard*j*" an external channel which comes in a procedure *j* . We call "screen *j*" an external channel which comes out a procedure *j*. The type of channel is defined from the type of the input external variables (resp. output external variables) in the LEQ sub-system which is represented by the node *j*. According to the type of the language LEQ the channels will have a type **INT**, **BOOL** or a n-uple of these ones. In the first case the type can be directly noted in the channel

declaration (for ex. **CHAN OF INT** keyboard1 :). In the second case, the definition of a protocole is necessary. Thus, in the axle-counter problem we cannot write

CHAN OF BOOL;BOOL keyboard1, keyboard2:

but **PROTOCOLE** input_type **IS BOOL;BOOL :**

CHAN OF input_type keyboard1, keyboard2: -- allow to read pl and pr

3-1-2- To create an "internal" channel

We know that a data dependence graph is built from basic graphs whose the type is producer-consumer via a communication. Then if we create a channel for each input and output edge of all communication, we create all internal channels (cf figure 4).

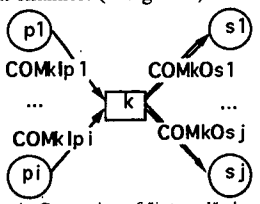


Figure 4 : Generation of "internal" channels.

Example of the axle-counter problem : **CHAN OF BOOL COM4I1, COM4O3, COM4I2 :**

3-2- To create the procedures

The calculation nodes give birth to calculation OCCAM procedures. The communication ones give birth to communication OCCAM procedures. In the two cases a computation of variables is defined by studying the recurrence order in the LEQ equational system associated to the considered node.

3-2-1- Calculation procedures

Each calculation procedure (node) represent only one equation LEQ. Its general form is :

```
PROC CALj ()
    declarations of variables
    SEQ
        computation of first terms of sequences
        OUTPUT_P        --- transmission of values on output channels
        WHILE TRUE
            SEQ
                computation of general terms of sequences
                OUTPUT_P
:
```

with *OUTPUT_P*::= **COMnosIj ! u | PAR** {**COMnosIj ! u**}$^{\forall nos \in lnos}$

where lnos = list of communications which receive the values of *u* computed in this procedure.

Each variable is the representation of a sequence of values. The set of variables which are useful in this procedure contains the input variables (read on channels coming in this procedure), the output variable (defined by the equation and diffused on all channels coming out this procedure), and the auxiliary variables introduced to transform the recurrence equation into an iterative program.

3-2-2- Communication processes

A communication *no* whose the extraction equation is the following : $cid_1 = \mathbf{when}(exp_pred, cid_2)$ is written in OCCAM by instanciating the parameters of a canonic procedure ([13]). The values of these parameters are defined from :

- the name and the type of output channels;
- the name of the input channels, the name of transmitted variables and their types;
- the equational system which defines the predicate (all equations excepted extraction one), and the equation ***pred*** = *exp_pred*;
- the second term (*cid2*) of the extraction equation. This n-uple of identifiers will allow to define from the input sequences the ones which give birth to the output sub-sequences.

As far as their use, all the values of the same variable transmitted to a communication by an input channel are saved in a array (fifo). The reception of values on an input channel, the computation of the emission condition and the emissions on an output channel are realized in a non-deterministic way.

Example: If the considered communication is written in LEQ as :

$$\{5\}\ \begin{cases} h = \textbf{true} \rightarrow \textbf{pre}(\textbf{false} \rightarrow (\textbf{not}\ h\ \textbf{or}\ x)) \\ h' = a\ \textbf{or}\ c \rightarrow \textbf{pre}(b \rightarrow \textbf{pre}\ a\ \textbf{and}\ (h'\ \textbf{or}\ b)) \\ cid_1 = \textbf{when}(h\ \textbf{and}\ h', cid_2) \end{cases}$$

the set of main equations whose the variables computations are deduced is defined by :

$$\{6\}\ \begin{cases} (1)\ h = \textbf{true} \rightarrow \textbf{pre}(\textbf{false} \rightarrow (\textbf{not}\ h\ \textbf{or}\ x)) \\ (2)\ h' = a\ \textbf{or}\ c \rightarrow \textbf{pre}(b \rightarrow \textbf{pre}\ a\ \textbf{and}\ (h'\ \textbf{or}\ b)) \\ (3)\ \textbf{pred} = h\ \textbf{and}\ h' \end{cases}$$

Thus, we will beforehand traduce, not only one equation (as in the case of calculation procedures) but every recurrent equation of the set.

3-3- Computation of variables defined by recurrent equations

A recurrent equation can not be directly transformed in an iterative program without analyzing in detail the dependancies in sequences (data dependancies and recurrence dependancies) ([17]).
To show it, we can consider the LEQ equation {6}(2). The computed sequence of values must be (a^0 **or** c^0,b^0,a^0 **and** (b^0 **or** b^1),...,a^{n-2} **and** (h'^{n-1} **or** b^{n-1}),...), then the iterative program can not use only four variables *h'*, *a*, *b*, and *c*. The following program is wrong :

```
... ? a  ... ? b  ... ? c  h' := a or c              --- first initial term (h^0 = a^0 or c^0)
... ? a  ... ? b  ... ? c  h' := b                   --- second initial term (h'^1 = b^1)
... ? a  ... ? b  ... ? c  h' := a and (h' or b)     --- general term (here it means h^n = a^n and (h'^n-1 or b^n)
```

Auxiliary variables must be introduced to memorize the values a^{n-2} of the parameter *a*. Then, computations and inputs must be scheduled to use the truth occurrences of data.

The following steps allow to realize these two points :

- to transform recurrent equations having k-dependancies (k>1) in a set of recurrent equations having only 1-dependancies,
- to delete cycles in the precedence graph associated to the created set of equations,
- to write the OCCAM program : declaration of variables, definition of computations of first and general terms.

3-3-1- High-order recurrence eliminations

The objective of this treatment is to give the valid occurrence of a sequence for a computation. In this aim we introduce new auxiliary equations (variables) to allow that the previous occurrences are available at the time of their uses. Thus the transformation called T_r applied on an equation in form

x = f(**pre**(g(**pre** e)))

yields the following set of equations : $\begin{cases} x = f(\textbf{pre}\ y) \\ T_r[y = g(\textbf{pre}\ e)] \end{cases}$ where y is a new identifier

Remark that, if e is not a **pre**-expression, $T_I[y = g(\textbf{pre}\ e)] \triangleq y = g(\textbf{pre}\ e)$.

Then the transformation of an equation having a recurrence order n ($n>1$) in an equation having a recurrence order 1 introduces $n-1+m$ auxiliary equations (variables) having an order 0. The number m of auxiliary equations is introduced by suppressing the k-dependancies in the external sequences.

Example : After transformation, the LEQ expression {6} is put in the form :

$$\{7\}\ \begin{cases} (1)\ h = \textbf{true} \rightarrow \textbf{pre}(\textbf{false} \rightarrow (\textbf{not}\ h\ \textbf{or}\ x)) \\ (2)\ h' = a\ \textbf{or}\ c \rightarrow \textbf{pre}\ z \\ (3)\ \textbf{pred} = h\ \textbf{and}\ h' \\ (4)\ z = b \rightarrow \textbf{pre}\ a\ \textbf{and}\ (h'\ \textbf{or}\ b) \end{cases}$$

Therefore the computation of variables can not be yet put in the form of an imperative program : two occurrences a^n and a^{n-1} of a same sequence a can not be associated to the same variable. A new transformation T_C is necessary. It is realized by analyzing the precedence graph of the equational system and suppressing the detected cycles.

3-3-2- Cycle eliminations

Recall that it exists a precedence link between x and y (y depends on x noted $x \xrightarrow{(n)} y$) if y is defined from x, or if x is defined from ***pre*** y (n is the reference of equation which links these two variables in the equational system). The both cases presented in figure 5 imply that the computation of x must be realized before y.
Therefore some equational system can generate precedence graphs having the form presented in figure 6 (containing a cycle). It means that x must be computed before y and y before x.
To find a computation of the system we transform one more time this system by introducing a new auxiliary variable z. The new equational system (transformed) is associated to the no-cycle precedence graph presented in figure 7.

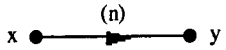


Figure 5 : y depends on x.

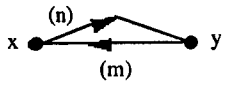


Figure 6 : Graph with cycle

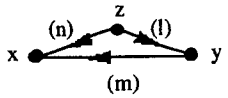


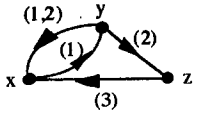
Figure 7 : Graph without cycle

This operation which deletes the cycles in the precedence graph must be applied until the precedence graph contains no cycles.

With intend to introduce significant auxiliary variables and to minimize their number, we suggest the following strategy.

- While at least a cycle exists in this graph :
  - A minimal cycle is extracted. We use a research algorithm of the shortest path (the path which "uses" as few equations as possible).

example : Figure 8 :



The path *(1,1)* is of course smaller than *(1,2,3)*, but it is also smaller than *(1,2)*.

  - The following transformation T_C is applied to the minimal sub-system.

$T_C[x=f(\textbf{pre}\ y),\ldots] \triangleq \{x=f(t);\ t=\textbf{pre}\ y;\ \ldots\}$

where "..." is a set of equations which can be empty, and f can be a mapping of y.

- The initial sub-system is replaced by its transformation in the global system whose the precedence graph is built.

The precedence graph is a graphic representation of the **pre** operations in the equational system. Each system may be then considerated as its graph and a set of equations without **pre.**

Example : The precedence graph (figure 9a) associated to the system {7} is analyzed. The strategy gives the new equational system {8} associated to the graph which is presented in figure 9b.

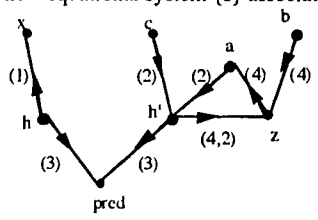


Figure 9a : Graph with cycle

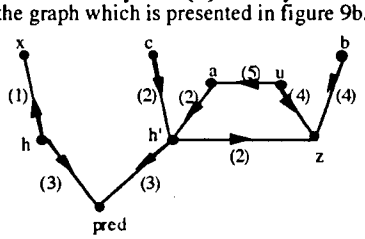


Figure 9b : Graph without cycle

The system {8} is then considerated as its graph (figure 9b) and the without-**pre** system {9}.

{8}
(1) h = **true** →**pre**(**false**→(**nothor**x))
(2) h' = a **or** c → **pre** z
(3) **pred** = h **and** h'
(4) z = b → u **and** (h' **or** b)
(5) u = **pre** a

{9}
(1) h = **true** →**false**→(**nothor**x)
(2) h' = a **or** c → z
(3) **pred** = h **and** h'
(4) z = b → u **and** (h' **or** b)
(5) u = a

3-3-3- Code generation

To write the OCCAM program we must declare variables and define the computations of first terms and of general terms. This second point needs to find the scheduling of computations then to define the computation of a term : affectation of variables, transformation of equations in mind to define the computations of the next terms.

3-3-3-1- Declaration of variables -
Each sequence defined by an equation is implemented by a variable. If the variable is auxiliary, its type is the same as the one of its expression, else it is given by type environment of the LEQ system .

3-3-3-2- Computation of first terms -
The sequence of instructions which is associated to each initial term of the equational system is defined by systematical traduction of equations after scheduling.

Scheduling : It is deduced from the analysis of the precedence graph :

- we take some entry point (without predecessor) in the precedence graph. We write the OCCAM affectation allowing the definition of its first terms.
- we delete this top in the graph then we start again the previous operation until the graph is empty.

Computations of the first initial terms : An entry point is implemented as following :

- the entry point is a variable coming in the procedure via a channel *A*. If any other variable given by *A* does not still acquire, the instruction is *A ? list_of_var_which_are_given_by_this_channel.*
- the entry point represents a sequence y defined by $y = e$.

An equation contains as much initial terms as opearators "→" in its expression LEQ (the number of initial terms for an equational set is the maximum of the number of initial terms of each equation).
In the computation of an initial term of an equation two cases can be exist :

- the considerated equation still contains at least an initial term ($x = e \rightarrow e'$). The OCCAM affectation is $x_{occam} := e_{occam}$ and the expression of the equation becomes $x = e'$.
- the considerated equation does not contain (any more) initial terms. The OCCAM affectation computes its general term. The equation is not modified.

Computations of others initial terms : It is realized in the same way that to the first term. Only one difference exits : the expression of equations does not contain reference to the alway treated initial term. The precedence graph does not take care to the variables used in the computation of first terms.

The definition of computations of the initial terms is finished when it does not exist operators "→" in set of equations.

Example : Computation of {9} : Two initial terms must be defined.

Definition of the first term : One of the possible scheduling of computations is the following : *h, u, x, a, b, c, h', pred* and *z*. The following instructions are deduced :

```
h := TRUE                                          --- the first term of h
u := a (any value)                                 --- auxiliary variable
... ? x     ... ? a     ... ? b     ... ? c        --- external variables are read
h' := a OR c                                       --- the first term of h'
pred := h' AND h     --- the first term of pred (defined according to the general term)
z := b               --- an other auxiliary variable
```

When the first term is defined, the equational system is {10}.

Definition of the second initial term : After graph analysis, the following instructions are deduced :

```
h := FALSE                                         --- second initial term of h
u := a                                             --- auxiliary variable
... ? x     ... ? a     ... ? b                    --- external variables are acquired
h' := z              --- the second term of h' is defined according to the general term
pred := h' AND h     --- second term of pred (defined according to the general term)
z := b               --- an other auxiliary variable
```

Now all initial terms are defined then the equational system is {11}.

{10}
(1) h = **false** → (**not** h **or** x)
(2) h' = z
(3) **pred** = h **and** h'
(4) z = u **and** (h' **or** b)
(5) u = a

{11}
(1) h = **not** h **or** x
(2) h' = z
(3) **pred** = h **and** h'
(4) z = u **and** (h' **or** b)
(5) u = a

3-3-3-3- Computation of general terms -
The determination of instructions which computes the general term is realized in the same way. Only one difference exists : in the case of the communication procedure the external variables are not acquired on the input channels. They have already been acquired and they are in buffers.

3-4- Parallel computing

Instruction to parallel computing :

```
PAR
    {COMno ()}^∀ no ∈ Ncom
    {CALno ()}^∀ no ∈ Ncal
:
```

with Ncom = list of numbers of communication nodes, Ncal = list of numbers of calculation nodes.

This OCCAM's instruction allows the simultaneous activation of all procedures.

Remark : You can find the OCCAM program of the axle-counter problem in annex.

5- CONCLUSION

The aim of this work is to contribute to define a systematic method which leads from recurrence equations to asynchronous parallel programs. If processes of a synchronous network are delayed by a static control and if occurrences dependencies are dynamically established, then such an interpretation must give a better efficiency that a synchronous one.

Then the first step of our work consists to give an explicit expression of the parallel decomposition whose we deduce progressively and systematically a parallel program, i.e. communications and synchronisations. To this aim we introduce efficient tools of expression and sound rules of transformation adapted to a parallel asynchronous computation model. The resulting program is an equational system written in the language LEQ. It is associated to a data dependence graph which is created from schemes containing a producer and a consumer linked by a communication medium. Then this system can be interpreted as a Kahn deterministic network [11].

The second step realizes the implementation of the equational LEQ system in an OCCAM network of communicating processes. Note that our method does not give an optimal code but our convention (all the j^{th} values of the input parameters are read to the j^{th} step) is satisfied.Thus, our compiling method is relatively simple and it gives a sound code in all the cases.

We implement this derivation method in the system CENTAUR [18]. We use METAL to implement the languages LEQ and OCCAM. We program the rules and the transformation strategies in TYPOL by using the natural semantics formalism [19]. The final tool allows to obtain OCCAM implementations of LEQ programs.

REFERENCES

[1] E.W. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, 1976.

[2] K.M. Chandy, J. Misra. *Parallel Program Design : A Foundation*. Addison-Wesley Publishing Compagny, 1988.

[3] Z. Manna, P.L. Wolper. *Synthesis of communicating systems*. LNCS 92, Springer-Verlag, 1984.

[4] P. Quinton. *Automatic Syntheses of Systolic Arrays from Uniform Recurrence Equations*. Proc. IEEE 11th Int. Symp. on Computer Architecture, Ann Arbor, MI, USA, pp 208-214, 1984.

[5] P. Clauss, C. Mongenet, G.R. Perrin, *Calculus of space-optimal mappings of systolic algorithms on processor arrays*, Int. Conf. ASAP'90, Princeton Univ., 09 / 1990.

[6] P. Feautrier. *Semantical analysis and mathematical programming; application to parallelization and vectorisation*. Workshop on Parallel and Distributed Algorithms, Bonas, Octobre 1988.

[7] F. Bieler. *Partitioning Programs into Processes*. CONPAR90. VAPPIV Zurich. Septembre 1990.

[8] E.A. Ascroft, W.W. Wadge, *LUCID : a non procedural language with iteration*, CACM vol 20 n°7,1977.

[9] N. Halbwachs, P.Caspi, D. Pilaud, J.A.Plaice, *LUSTRE: A declarative language for programming synchronous systems*, LNCS n° 215, pages 178-188

[10] M-C. Eglin, J. Julliand. *Translating Equational Systems into Communicating Processes*. Internatinal Journal of Mini&Mirocomputers, ACTA Press, Volume 15, Number 3, 1993.

[11] G. Kahn. *The semantics of a simple language for parallel programming*. Information Processing 74. North-Holland Publishing Compagny. 1974.

[12] D. May, *The OCCAM language*, SIGPLAN Notices, vol. 18, n° 4, 1983.

[13] M-C. Eglin-Leclerc. *Compilation de 'Systèmes d'Equations Récurrentes en Réseaux de Processus Communicants*. Thesis of the Univ. of Franche-Comté, France, 12/1991.

[14] C. Mauras, *Définition de ALPHA : un langage pour la programmation systolique*, INRIA Research Report. 09/1989.

[15] A.V. Aho, R. Sethi, J.D. Ullman, Compilers : *Principles, Techniques, and Tools*,Addison Wesley, 1986.

[16] M-C. Eglin, J. Julliand, G-R. Perrin. *Generating parallel processes from recurrence equations*. Parallel Computing and Transputer Applications, Barcelone, Espagne, september 1992.

[17] GREGOIRE, *Informatique-Programmation, Tome 1*, édition Masson, 1986.

[18] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, *CENTAUR Version 0.5*, INRIA Sophia-Antipolis, 02 / 1988.

[19] G. Kahn, *Natural semantics*, INRIA Research Report n° 601, 1989.

ANNEX : OCCAM PROGRAM implementing the railway regulation

```
CHAN OF INT screen3 :
CHAN OF BOOL COM4I1, COM4I2, COM4O3 :
PROTOCOLE input_type BOOL ; BOOL :
CHAN OF input_type keyboard1,keyboard2:
```

```
PROC COM4 ()        -- extraction communication
  [ ...] BOOL Tabep , Tabes , TabO :
  INT beg ,beg3, end1 , end2 , endO :
  BOOL ep, es, pred :
  SEQ
    beg := 0
    end1 := 0
    end2 := 0
    endO := 0
    beg3 := 0
    WHILE  TRUE
      ALT
        COM4I1   ? Tabep[ end1 ]
            end1 := end1 + 1
        COM4I2   ? Tabes[ end2 ]
            end2 := end2 + 1
        beg < min( end1 ,end2 )
            SEQ
                ep := TABep[beg]
                es := TABes[beg]
                pred := ep OR sb
                beg := beg + 1
                IF pred
                  SEQ
                        TABO[endO] := es
                        endO := endO + 1
                TRUE
                  SKIP
        endO > beg3
            IF COM4O3 ! TABO[beg3]
                beg3 := beg3 + 1
            TRUE
                SKIP
:
```

```
PAR      -- instruction of parallel computing
  CAL1 ()
  CAL2 ()
  CAL3 ()
  COM4 ()
```

```
PROC   CAL1 ()  -- computation of the sequence ep
  BOOL pp, ps, pp1 :   --- pp1 represents pre pp
  SEQ
    pp1 := pp
    keyboard1 ? pp, ps
    ep := FALSE
    COM4I1 ! ep
    WHILE   TRUE
      SEQ
        pp1 := pp
        keyboard1 ? pp, ps
        ep:=(NOTpp) AND pp1 AND (NOTps)
        COM4I1 ! ep
:
```

```
PROC CAL2 ()   -- computation of the sequence es
  BOOL pp, ps, ps1 :   --- ps1 represents pre ps
  SEQ
    ps1 := ps
    keyboard2 ? pp, ps
    es := FALSE
    COM4I2 ! es
    WHILE   TRUE
      SEQ
        ps1 := ps
        keyboard2 ? pp, ps
        es:=(NOTps) AND ps1 AND (NOTpp)
        COM4I2 ! es
:
```

```
PROC CAL3 ()   -- computation of n
  BOOL es' :
  INT n :
  SEQ
    COM4O3 ? es'
    n := 0
    screen3 ! n
    WHILE   TRUE
      SEQ
        COM4OP3 ? es'
        IF es' = TRUE
          n := n + 1
        TRUE
          n := n - 1
        screen3 ! n
:
```

Fortran DVM - Language for Portable Parallel Programs development.

N.A.Konovalov, V.A.Krukov, S.N.Michailov, A.A.Pogrebtsov
Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya Sq, Moscow 125047, Russia
E-mail: krukov@d23.keldysh.msk.su (Internet & Bitnet)

Abstract

This report introduces the new language model which allows to express functional and data parallelism in scientific and engineering applications on massively parallel computers. The model combines the major advantages of PCF Fortran and HPF models and is designed for development of portable applications which can be efficiently executed on distributed and shared memory computers. The model supports development of library of standard parallel programs and construction of parallel applications from parallel modules. The model also provides the means for dynamic load balancing.

Key Words: Portable Parallel Program, Massively Parallel Computers, Distributed Memory, Shared Memory, Load Balancing.

1 Introduction

The development of the programming language Fortran DVM (Distributed Virtual Memory) and related tools is hold in the Keldysh Institute of Applied Mathematics as a part of RAMPA-project [1]. The main goal of this project is to automate the development of the portable scientific and engineering applications for massively parallel computers with distributed memory.

The development of the applicational programs for distributed systems encounters a few obstacles. The portability problem is the major one. We would like to emphasize two aspects of this problem :

- The concurrent programs, developed with conventional language tools for the distributed systems, require serious efforts for their porting on computers with different architectures (sequential, SIMD processor arrays, MIMD shared memory machines), or different configurations.

- The other two problems that should be solved are linking the concurrent program using simple modules and accumulating the bank of common concurrent programs.

While there is no solution for the portability problem distributed systems are not likely to be widely used.

Since that the main goal of the development of Fortran DVM language (FDVM) is to provide the portability of parallel programs.

Besides the FDVM language must provide the development of efficient programs for massively parallel computers. In this case the performance is determined mainly by the load balancing and overheads for the interprocessor communication and synchronization.

2 The Main Ideas of the FDVM Language.

The concept of FDVM language is based on the following main ideas:

- The execution of a parallel program must correspond with its potential parallelism, explicitly declared by a user by means of the special directives or parallel constructs. Even enhanced with specification of the data distribution the method of the automatic parallelizing has a principal defect: there is no explicit and clear model of parallel execution. This makes impossible to provide a user with convenient tools for the analysis of the execution and improving performance of his program.

  We believe that distributed shared memory computers are going to become the most popular among massive parallel systems. Since that to describe the potential parallelism of a program it is natural to use the means suggested in the PCF Fortran standard [2]. These means represent the extensive experience of utilizing multiprocessing computers with shared memory. Certainly, it is necessary to adopt these means to distributed memory computers.

- In case of non-uniform memory computers (among them are distributed systems, with logical-shared but physically distributed memory) it is necessary to provide a user with the means to determine the data mapping and to distribute the computations among the processors. The suggested means of the HPF language [3] are well suited.

- In case of computers with non-uniform memory it is imperative to provide a user with the means of a virtual memory access control.

These means were suggested and successfully used for the control of virtual memory in the Fortran programs for BESM-6 computer [4].

- Finally, it is necessary to give a user means for the dynamical load balancing.

3 The Description of FDVM Model.

For the purpose of simplification suppose that programming in Fortran DVM is done in several stages as follows.

3.1 The Abstract Parallel Machine (APM) Design.

The description of the parallel machine is necessary for a user to control mapping of computations on processors. In case of distributed system APM is also needed to control the data location. Since PCF Fortran is geared to the uniform multiprocessors computers with shared memory, a user have only capabilities to determine the number of processors for execution of the specific parts of a program. In the case of distributed systems the mapping of computations on processors demands more accurate planning and could not be performed as dynamically as it is possible on the shared memory computers.

Abstract parallel machine is introduced to provide a user with a opportunity to completely specify internal potential parallelism of a program. Since that the APM is the hierarchy of abstract parallel subsystems, all of which are either set of named subsystems of the next level of hierarchy or multidimensional array of such subsystems. The representation as a set is convenient for parallel sections mapping. The representation as an array is used for loop iterations mapping. Subsystems of the lowest level of the hierarchy are elementary abstract processors. The simultaneous existence of different variants of a description of each subsystem is allowed.

If APM or some its subsystems are to be used for data location management, they must have at least one representation as an multidimensional array of elementary processors.

The APM may be described statically or constructed dynamically during an execution of a program. The dynamic construction is necessary for work with dynamic arrays, as well as for the development of a library of standard

parallel programs, which are stored as object modules and do not require special preliminary setting.

3.2 Development of a Program for APM.

To develop a program a user uses the following model of its execution.

At the beginning of an execution of a program there is only one branch (the control flow) which is started on the APM from the first program statement. After the entering the parallel construct (parallel loop or group of parallel sections) the branch is divided into several parallel branches, which are all executed on the specified APM subsystems. The branching is carrying out according to a variant of representation of current subsystem as an array or set of subsystems of the next level of hierarchy.

On exit from the parallel construct all branches join the same branch, which existed prior to entering the parallel construct. At this moment all shared variables, updated by the parallel branches, become available to all processors executing this branch.

If during the execution of one of the parallel branches it is necessary to use the values of shared variables updated by the other branch, a user must synchronize the branches. The synchronization is available in several forms: events, arithmetic sequence synchronizers, and critical sections (PCF Fortran).

To avoid formal data dependence of parallel branches, which is an obstacle to independent execution, some variables (and specially described common-blocks) may be declared as private. In this case a copy of these variables is created in each parallel branch, and that copy is used independently from the others. Private data definition also reduces the scope of availability of data, and eliminates the needless work to provide data consistency.

Now let us determine the meaning of the phrase "branch is executed on the subsystem". These words could be interpreted in two ways, depending upon what architecture of APM is needed by a user. In case of shared memory machine the sequential parts of program (before the enter to parallel construct and after the exit from it) are executed on the base processor of the subsystem. In case of distributed memory machine each processor of the subsystem executes the same sequence of statements to have its own copy of variables.

3.3 Declaration of Data and Computations Mapping.

In case of distributed system a user must determine data location in the local memory of the APM processors. It is implemented through the aligning of the arrays and their mapping to those APM subsystems, which have a structure as a multidimensional array of processors (ALIGN in the HPF). Collapsed mapping (when many elements of an array may be mapped to the same processor) and replicated mapping (when one element of an array may be mapped to many processors) are allowed.

A user specifies data replication when its calculation can be performed more efficiently repeatedly on different processors than transferring the data between the processors. By default all data which are not mapped by user, are duplicated on all processors of the subsystem where the branch owning these data (the data are private for this branch), is executed.

To specify the mapping of parallel loop iterations on the processors a user can define the correspondence between the loop iterations and preliminary mapped variables.

3.4 Managing Access To Non-Local Data.

A user can indicate that when accessing some arrays the access to virtual memory should be performed in bulk before-hand transfers (after the computation of required values by a corresponding processors) rather than element-wise transfers (according to requests from individual processors). In case of pipeline computation, a programmer must find the balance between the number of transfers and synchronization events on one hand and the productivity of the pipeline on the other.

A user can also specify the buffering mode of non-local variables in the processors memory, which permits to reduce the interprocessor data exchange during repeated access.

3.5 The Specification of Mapping of the APM onto Virtual Parallel Machine (VPM).

Virtual Parallel Machine (VPM) is the machine which is provided to the task by architecture and underlying system software. The structure and number of processors of this machine should be close to real parallel computer or part of it. For distributed computer MPI-machine [5] could be such an example.

VPM as well as APM may be represented as an hierarchy of subsystems. In this case the subsystem should be precisely defined (numbers of its all

virtual processors). A subsystem must include only those processors, which belong to the mother subsystem.

Mapping is correspondence between APM and VPM resulting in each processor of APM will be mapped onto a processor of VPM. We suggest to provide three ways for such mapping.

- The mapping may be specified by the language constructs, processed during the compilation of the program (DISTRIBUTE in the HPF). This way is the least flexible, but the most effective in terms of overheads.
- The mapping may be specified through the programming environment (for example, as a UNIX command-line argument).
- The mapping may be made dynamically during program execution by the special statements (REDISTRIBUTE in the HPF).

  This way of mapping as well as means for use of allocatable assumed-shape arrays and means of dynamic creation of APM and VPM are necessary for a user to ensure dynamic load balancing.

4 Comparison of FDVM Model with Major Existing Models.

Let us consider the differences of the FDVM model from the other models, which can be used in distributed systems.

4.1 Message Passing Models.

Currently these are the most widely used models. In those models a user specifies the precise data and computation mapping on the processors and provides the processors cooperation by message passing. Although this approach seems to grant the maximum efficiency of a program execution it is not always true due to the following reasons.

- The reduction functions (SUM,MAX,MIN) significantly affect efficiency of a program. In general their optimal programming on the applicational level is impossible. There are no internal reduction functions in most of the models, including PVM [6], though such functions exist in MPI and EXPRESS [7].

- It is difficult to organize the effective file access in application if the distributed system has many channels to discs.
- Execution efficiency of a program on the massively parallel computers depends on the load balancing which is extremely difficult to provide in these models.

Although the models are quite expedient in expressing of functional parallelism, in case of data parallelism there are serious problems due to the lack of the global address and name space. The main defect of these models is the strict tie of the program to hardware architecture, including the number of processors and their links. The development of the libraries of standard programs is practically impossible.

The unquestionable advantage of these models is no need to develop special compilers and possibility to use these models in conventional languages extended by library functions.

4.2 Shared Memory Models.

These models use global address space. PCF Fortran and Fortran S [8] are the most characteristic among these models.

The PCF Fortran standard noted that this model is acceptable not only for the shared memory computers, but also for the distributed memory computers. But this is mostly true for systems such as CRAY T3D and CONVEX SPP1000 (distributed shared memory computers) which provide the access to non-local data nearly as fast as to local data. The use of this model on the majority of distributed systems is practically impossible due to the lack of means to control data and computations mapping.

Fortran S model provides a user with means of mapping of computations onto processors. Unfortunately the use of program-simulated virtual memory in the model without means to control it unenviably reduces the efficiency of serious applications.

The advantage of these model, as well as FDVM model, is relatively simple compilation.

4.3 High Performance Fortran Model.

This model uses global name space and user control over data mapping onto local memories of processors. The model supports the data parallelism and does not support functional parallelism.

There are also two principal differences in the concepts of Fortran DVM and High Performance Fortran languages.

First, the user of FDVM has the possibility of total control over the distribution of computations among processors, while the user of HPF is not allowed even to know how a particular compiler will perform such distribution.

Second, the user of FDVM can fully control the transfer of data between processors and its buffering. The user of HPF has to rely on the "smartness" of the compiler in these matters.

The main effect of the differences in approaches mentioned above is that at the cost of certain additional effort the user of FDVM is always able to make his program as efficient as it could be when conventional means for parallelization and message passing were used. Yet this does not involve any decrease in portability.

The following means of tuning performance of FDVM programs are provided, which are absent in HPF:

- Means of processor load balancing:
  - possibility to define groups of processors, varying in shape and composition;
  - possibility to cancel computation of new values of a variable on the same processor thus spreading certain loop iterations and parallel section over several specified processors;
- Means of optimizing interprocessor communication:
  - definition of data portion sizes;
  - definition of buffer sizes and buffering modes;

Differences in approaches between FDVM and HPF have lead to different views on the role of compile-time optimizations. The lack of an optimizing compiler from FDVM does not stop the user from achieving required efficiency, although the use of such compiler might in certain cases free him from providing the above annotations.

5 Conclusion.

We believe that the success of FDVM language depends on the successful development of FDVM model. This model on one hand must meet the porta-

bility and efficiency requirements and on the other hand must be clear for an applicational programmer as well as relatively easy to implement. Since that developers were focused on designing of the model, which was three times significantly modified during 1993-1994.

Presently the design of the run-time system (LIB-DVM) for this model is close to completion. This run-time system is a backbone of the implementation of FDVM language.

The next stage of FDVM language development is going to be manual programming of several applications (probably using the simple experimental compiler-preprocessor) on the Fortran 77 or C languages extended with LIB-DVM means.

Only after the completion of the above stage the formal FDVM language will be described. If by that time the means of functional-parallelism and dynamic load balancing are introduced to HPF language , it would be reflected in FDVM language.

This work is supported in part by the grant of Russian Fundamental Researches Foundation, No. 93-012-628.

References

[1] V.A.Krukov, L.A.Pozdnjakov, I.B.Zadykhailo. *RAMPA - CASE for portable parallel programs development.* Proceedings of the International Conference "Parallel Computing Technologies", Obninck, Russia, Aug 30-Sept 4, 1993.

[2] *PCF Fortran.* Version 3.1, Aug.1, 1990.

[3] *High Performance Fortran.* Language Specification. Version 1.0, Jan.25, 1993.

[4] N. A. Konovalov, V. A. Krukov, E. Z. Ljubimskij. *Controlled virtual memory.* Programming 1, 1977 (In Russian).

[5] *MPI: A Message Passing Interface.* The MPI Forum, August 1993.

[6] A.Geist, A.Beguelin, J.Dongarra, W.Jiang, R.Manchek, V.Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory ORNL/TM-12187, May 1993.

[7] *EXPRESS: A Communication Environment for Parallel Computers.* ParaSoft Corporation, Release 2.4, 1988.

[8] F.Bodin, L.Kervella, T.Priol. *Fortran S: a Fortran Interface for Shared Virtual Memory Architectures.* Int. Conf. on Supercomputing, 274-283, July 1993.

Section II :
Neural Networks

NEUTRAM -- A Transputer Based Neural Network Simulator

Dmitry O. Gorodnichy*, Alexander M. Reznik
Institute of Problems of Mathematical Machines and Systems
Cybernatics Center of Ukrainian Academy of Sciences
* - Email: dmitry@dmipt.icyb.kiev.ua
Kiev, Ukraine

1. Inroduction

In recent years Artificial Neural Networks (NN) became increasingly popular and of great importance as an alternative approach for solving problems where no precise rules are known, such as error correction, pattern recognition, classification, associative memory etc. Contrary to classical programming, neural networks are trained by some learning rules with examples of a particular problem in order to obtain the final solution on the base of acquired knowledge.

Researches in this field, such as finding an optimal learning rule, investigating the behavior of a neural network (i.e. its ability to solve a particular problem), is extremely difficult sometime without simulation of the model under development. Conventional computer systems often do not yield the required performance for such Parallel Distributed Processing even with 100 units. The problems are that available memory is not sufficient and processing iteration rate is far from desired; it is not uncommon for such simulations to take many hours to process simple network. The introduction of parallel computers offers new opportunities in neural network simulation technology.

Here in Kiev in order to investigate practically particular learning rules of connectionist models was designed the neural net simulator NEUTRAM, which simulates NN of various sizes and configurations on a net of transputers (for this purpose four INMOS T800 transputers were used). It processes in almost real-time mode with a net with up to 600 neurons, i.e. up to 36000 connections. For comparing, the same program running on an Intel 486DX-50 processor processes nearly an order of magnitude slowly, and with memory restriction to 200 neurons. Now NEUTRAM is being intensively used in IPMMS of Ukrainian Ac.Sc, and

with its aid a great quantity of useful statistics has been being obtained, which itself helps to solve many theoretical problems.

This paper is dedicated to the strategy of flood-fill neuro-processing, which allows the NEUTRAM to achieve high performance iteration rate, and its implementation on distributed processors. The questions of organization of interaction of distributed units-neurons are considered.

2. Neural Net as a System of Independent Units-Neurons

The ANN is typically modeled a system of N simple units - neurons. At any time t each neuron i is characterized by a couple of figures ($Yi(t)$, $Si(t)$), where $Yi(t)$ is two-state { -1, +1 } potential (or state, or output) of the neuron at time t, and $Si(t)$ is its local field (also called post-synaptic potential, or membrane potential, or magnetization) is computed by performing the sum of its inputs weighted by the weight (or synaptic, or coupling) coefficients **Cji** (where Cji means connection from neuron j to neuron i), sometime minus threshold value Bi:

$$Si = \sum_{j=1}^{Ni} Cji\, Yj, \qquad (1)$$

where **Ni**, called the size of synaptic tree of the neuron, is the number of neuron inputs (correspondingly, the number of neuron outputs is called the size of its axon tree). And synaptic coefficients **Cji** are constant values, calculated previously in so called learning stage on the basis of properties we want our NN to exhibit. The behavior of the network (i.e. its evolution in time) is determined by the decision rule:

$$Yi(t+1) = F(Si) = \begin{cases} 1, & Si > 0 \\ Yi(t), & Si = 0 \\ -1, & Si < 0 \end{cases} \qquad (2)$$

Since each neuron is processed independently from others, NN has great potential for parallelism, and that's why it is preferable to simulate them on parallel processors.

3. Flood-fill Neuro-processing.

The concept "neuro-processing" refers to all operations under the neural net data, such as (1) and (2), resulting in calculating neuron potentials. And as can be seen, the most time-consuming operation there is the calculating of postsynaptic potential in (1), which is executed for i=**1..N** in every single iteration. The time necessary for calculating **Si** by (1) is

$$T = Ni*(Tc+Ta), \qquad (3)$$

where Tc is time of "compare" operation , Ta - time of "add" operation (Ta = Tc).

The main idea of flood-fill processing strategy consists in the fact, that only a few quantity of neurons really alter their potentials during an iteration. Moreover, the number of such neurons diminishes increasingly with each iteration, until eventually it becomes equal to zero, when we say, that iteration process has terminated. This brings us to the natural procedure of calculating postsynaptic potential:

$$S[i] = S[i] + 2 \sum_{k=1}^{Ki} C[k][i]\, Y[k], \qquad (4)$$

where Y[k] are potentials of those, called "excited", neurons, that have altered their potentials since last iteration. (Furthermore, even from the very beginning we can often say (looking to initial input pattern), that some neurons are "excited", comparing with others, prevailing in the pattern (e.g. as in the case of white letter, drawn on the black background). Here the time for calculating S[i] is

$$T = 1 + Ki*(Tc+2Ta), \qquad (5)$$

From here it become obvious that, when Ki << 2/3Ni, we have a considerable increase in calculation rate.

4. The Implementation of Flood-fill Neuro-processing.

Now consider NN with arbitrary configuration (i.e. NN with size **Nj** of axon tree different for each neuron). The configuration of a NN in processor memory is presented by the set of axon tree, according to the following data structure (see FIG. 1):

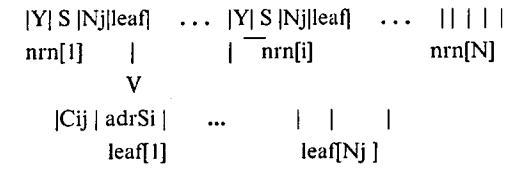


FIG.1a *Neural net configuration data structure used for neuro-processing*

```
NEURON
  { int Y;
    float S;
    int N;
   TREE *leaf;
  } ;

TREE
{
 float Cij;
 float *adrSi;
} ;

NEURON *nrn;
```

FIG.1b *Data structure used in program*

And described by step the flood-fill procedure will be following:

Step a: Potential that dominates in the majority of the prototype neurons is designated as "background" (e.g. in the program this state is -1, which corresponds to the black colour of the pattern).

Step b: initial postsynaptic potential S for this background is calculated: $S[i] = -\Sigma C[k][i]$.

Step 1: After an input vector is given, the buffer of "excited" neuron ebuf is filled, that is: the numbers of such neurons, which potential differs from background is put down into the buffer (e.g. *ebuf* = {2,3,4,21,37,38 }).

Step 2: Dissolving the buffer: once neuron i is excited (i.e. *ebuf* contain number i), it bring about alterations of S[k] of all neurons connected with it. Hence, we should run through all axon tree of each excited neuron and update all S[k], depended on it (this Step can be processed under all neuron in parallel):

if (Y[i] == +1) S[k] += 2 C[i][k]

if (Y[i] == -1) S[k] -= 2 C[i][k]. k=1..N .

Step 3: After all consequences of exciting of some neurons are resolved (i.e. all S[k] are updated), we have exactly $S[i] = \Sigma\ C[j][i]\ Y[j]$. Here we can apply the decision rule Y[i] = F(S[i]), which gives again some excited neurons (The number of them will be probably less than that in previous iteration). This Step also can be processed under all neuron in parallel .

Step 4: And again we go to Step 1, putting into ebuf the sign of neuron potential in addition to its number (e.g. *ebuf* = { 1,5,-37,-38 })

Iteration occurs until ebsize, the size of *ebuf*, equals zero (which corresponds to empty buffer), so there is nothing to dissolve. And the body of the program looks like:

1. fill buffer *ebuf*;
2. dissolve buffer *ebuf*
3. apply the decision rule;

5. The Transputer Implementation of Flood-fill procedure.

INMOS transputer itself is a very powerful processor, RISC architecture of which allows very fast operations. Moreover, it can process memory operation (such as, storing and picking) concurrently with arithmetical operations. But the main feature of transputer is that it can be linked with other transputers via links, link interface working concurrently with CPU, which allows user to distribute program code among processors to run it simultaneously. This facility gives user not only the opportunity to acquire new amounts of memory, which itself is quite worthy, but also the chance to increase the calculation process rate in almost as much as the number of transputer presented times (provided that the program code is suited for this). So the task to prepare the program to run in parallel is one of the most complex and important for those who work with transputers. And the passage below is dedicated to how this task is resolved in NEUTRAM.

When simulating NN on the net of processors, we allocate neurons uniformly among processors available, each axon tree being allocated as shown in FiG.2.

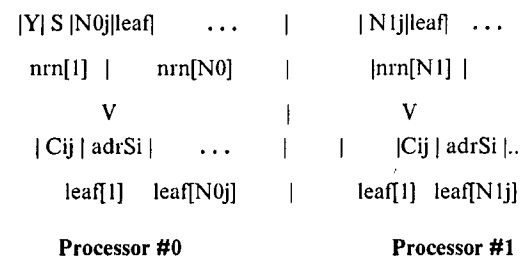


FIG.2 *Structure of distributed axon tree*

(as in the case of simulation neuron net on a array of processors)

Thus each transputer has in memory its own part of state vector and own part of synapse weights, and process only these parts, filling its own *ebuf* of its own excited neurons; to update its postsynaptic potential transputer receives *ebuf*[i] from other transputers. And the task of each processor therefore consists of six main processes:

1. fill my buffer *ebuf*;
2. send my buffer *ebuf*;
3. receive other buffers *ebuf*[i] from other transputers;
4. dissolve my buffer *ebuf*;
5. dissolve other buffers *ebuf*[i];
6. apply the decision rule;

until (*ebuf* and all *ebuf*[i] are empty)

And since it is desirable for CPU and link interface to work simultaneously, we allocate designated processes in such a way that delivering information and its processing work in parallel. Below is the structure of the program NEUTRAM, running on a net of four transputers, the flood-fill neuro-processing within one processor being presented.

```
WHILE ( all ebuf are not empty )
 SEQ
(1)   filling_own_buffer( );                    /* filling ebuf   */
      PAR
(4)     process_own_data( );                   /* dissolving ebuf */
(2)     sent_buf_to_others( ); /* send ebuf to other processors */
(3&5)  receive&process_data_from_otherTRAM( );
       /* receive ebuf[i] from other processors and dissolve them */
( 6 )  apply_decision_rule();
```

As can be seen the task of one processor does not depend on the configuration of transputer net. In the case when processors are not connected directly, data should be delivered thru intermediary, which results in introduction one more process ROUTER(). And in order to resolve the problem of mapping software links onto hardware ones the process MultiPlexor() should also be added. But in this paper we don't discuss these questions, since we deviate from them by simulating NEUTRAM on four transputers.

6. Transputer Simulation Implementation Problems

The main problem arises in simulating NN on transputers is data exchange among the processors. It can be implemented either by public memory (heap), or by channels (here arises the problem of four real links available, which entails the introduction of multiplexor processes for array of more than four transputers). The next problem associates with particular implementation of the program. As programming language OCCAM or 3L C (or any other language extended from sequential one) can be used. The former specially designed for implementing parallel programs, being the entirely enclosed subsystem, lacks language manipulation tools (excepting the folding editor); the latter having such facilities as dynamic memory allocation and processing complex structure, has more difficulties in organizing really parallel processes.

Since in the case of simulating neural nets the data of parallel processes are strongly interwoven, and the size of network is assumed to be defined while running the program, 3L C seemed to be more suitable for writing the simulator. And we use it for our purpose, which results in

introduction of complex system of semaphores in order to organize mentioned above PAR-process.

5. Conclusion

The flood-fill neuro-processing algorithm was developed. This algorithm was shown to yield a considerable increase in neuro-processing calculation rate , when given a number of excited (informative) neurons less than 30% from total number of neurons in the network, which is , generally speaking, the case for more practical applications. The problems of its implementation onto multy-transputer system were considered, the ways for data structure allocation and data interaction organization being discussed. Also considered were the questions of particular flood-fill neuro-processing algorithm implementation on the net of four transputers in NEUTRAM, neuron network simulator designed for investigation of connectionist models.

References

1. Wasserman, P. D. (1989). Neural Computing: Theory & Practice. // Van Nostrand ,Reinhold, New York. (ISBN 0-442-20743-3)

2. The Transputer Databook //INMOS Limited, 1988

A Transputer Based Robot Vision System to Evaluate Neuro-Control

D. Kuhn J.-P. Urban S. Hagmann H. Kihl

Laboratoire TROP, Faculté des Sciences et Techniques
Université de Haute Alsace
F-68093 Mulhouse Cedex, France

e-mail: kuhnd@uhafst.univ-mulhouse.fr
Tel. 33 89 59 64 32 Fax 33 89 59 63 59

Abstract. The performance of different types of neural networks are to be evaluated when applied to real robotic applications like manipulator arm movements and arm positioning. An experimental robotic platform is designed consisting of a five degrees of freedom manipulator, and an image acquisition system to provide visual feedback. The architecture of the system is modular and entirely transputer-based, including image acquisition and processing, the Neural Network generating control signals and the robot interface. The approach is illustrated on a simple arm positioning task.

keywords: Transputer, Neural Networks, Control, Robot Vision

1. Introduction

A distinctive property of the primates sensorimotor systems is certainly their fast and efficient reaction to unpredictable changes in their environment. An other basic ability is to react in a parallel and coordinated manner with their different effectors. Research in Neuroscience these last two decades helped to better understand the mechanisms of certain parts of the brain. For example, it is known that the cerebellum is one of the regions of the brain involved in the adaptive control of movement and is organized as a network of massively interconnected neurons. Its structure has inspired quantities of works resulting to models derived from the perceptron theory and able to learn complex input output functions [1,2].

Artificial Neural Networks (ANN) offer a way to view certain complex problems from a new perspective. One of the intrinsic properties of ANN are their ability to adapt to changes in the parameters of the model. Their implementation is simpler than the use of mathematical models and the behavior is robust when faced with accidental changes of system parameters [3]. Growth of interest to apply ANN for control applications has being considerable in the last years and a few real experiments can be noted [4,5].

The objective we pursue is to examine the performance of different types of neural networks when applied to real applications and to examine their suitability for robot control. We therefore developed a robotic platform consisting of a 5 degrees of freedom manipulator with a jaw gripper similar to those most currently used in industry, and an image acquisition system to provide visual feedback through one or more cameras.

Next section details the associative neural network approach. Section 3 justifies the choice for a transputer based parallel architecture and section 4 describes the simulation environment developed for the design and test of the ANN before being runned under experimental conditions. Finally, a simple positioning task illustrates the global functioning of the whole system.

2. Neural Network Approach

Recently, there has been a considerable growth of interest in establishing a direct connection between vision and robot action through neural networks. In a conventional approach, sensor-based control systems require explicit knowledge of the kinematics and dynamics of the robot and a careful calibration of the sensor system. One attractive feature of neural networks is their learning capabilities. They can learn a complex nonlinear relationship, such as robot kinematics and dynamics through a training procedure and approximate the function with significantly less computations.

We are interested in a self-learning system where no explicit knowledge is present but where an implicit model is learned from the behavior of the robot. The robot system receives all the information needed for adaptation from its own cameras and, thus, learns without an external teacher. Kuperstein [6] developed a direct inverse control scheme in which a neural net learns the inverse dynamics of a robot system so that it can follow a desired trajectory. Kawato [7] applied neural networks as adaptive kinematic or dynamic controllers. Ritter, et al, formed an association between a pair of position vectors representing the images of a 3D target point on the camera retinas and a joint angle vector, where individual pairs of position vectors are organized into a 3D feature map based on Kohonen's self-organizing learning [8]. The use of control loops and Self-Organization Maps (SOM) ensures both fast learning of the algorithm and good working precision [9].

In a Kohonen type of network [10], each neuron represents a part of the input space and associates the corresponding output vector. This direct association, behaving like a learning 'look up table', characterizes all networks allowing fast learning for control applications. We use therefore the generic terminology Associative Neural Network (AsNN). At first, we test a modified SOM algorithm for control [8] and a mixed SOM + Backpropagation algorithm.

Figure 2.1. gives a functional diagram of the Robot Vision system.

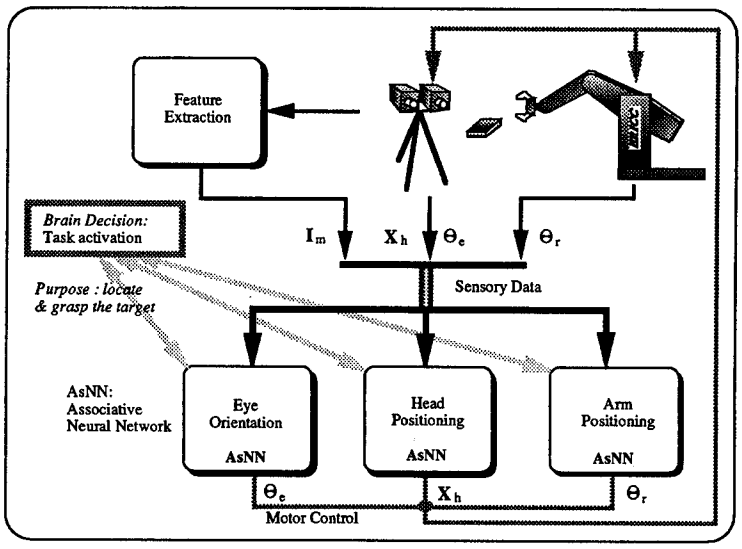


Fig. 2.1 : Functional diagram of the Robot-Vision System

The basic robotic task we address is the positioning of the end-effector of the arm relatively to a known object seen by a controllable stereoscopic vision system. This generic task includes control of ballistic movements, visual servoing for final approach, camera control for eye and head positioning. The problem is divided in a set of elementary tasks which will be addressed individually. The bottom of figure 2.1. shows three closed-loop functions implemented independently with Associative Neural Networks (AsNN). The sensory input data contains Image features *Im* and the joint positions of the arm and the head *Qr, Qe, Xh*. The different AsNN and the feature extraction can be implemented separately and concurrently. Each AsNN will be parallelized to meet the real-time implementation constraints [11].

3. Architecture Considerations

3.1. The Need

At first glance, the whole experimental setup can be viewed as three independent interconnected systems. An image processing system, where visual input from one or more camera is adequately processed to extract image features. A computing unit implementing the AsNN issuing the control signals for the robots arm movements. A robot manipulator including the mechanical structure, the electrical actuators and power control, and a user friendly control interface.

Additional constraints have to be taken into account :
The computing power dedicated to the implementation of the AsNN must be sizeable : not only different kinds of AsNN will be evaluated, but also different arm movement applications will be tested involving the synchronization of several networks. The computational power required being application and solution dependent, it can not be estimated precisely but grows fast as the number of neurons increases.
One important point is the satisfaction of the real time constraint : In order to ensure visual servoing, the visual feedback has to be realized at a reasonable rate. This means that the information from the camera(s) has to be processed, the resulting image features input to the AsNN and the resulting control signals fed back to the robot controller at close to video rate (40 ms). In practice this implies that a limited processing time is assigned to each module, and that the communication between them does not slow down the process.

These remarks argue for a more integrated system both for hardware considerations and homogeneity of software development. Therefore a parallel implementation, entirely transputer based, was selected. The main advantages are :
- *a modular structure* : Each function of the system constitutes an autonomous module.
- *an homogeneous structure* : The basic independent and parallelizable modules are all transputer based and communicate homogeneously through serial transputer links.
- *a sizeable computing power* : This need can be met by increasing the transputer network and parallelizing the algorithms.
- a portable code : The source code can be developed on different platforms (MS-DOS, UNIX...) and easily adapted to the number of processors required and is compatible with next generation transputers. The available compilers are C, C++, OCCAM.

3.2. System Architecture

Figure 3.1 gives a global view of the system architecture retained. The transputer system is hosted by a PC through two interconnected motherboards receiving up to 10 TRAnsputer Modules (TRAMs) each. General purpose TRAMs as well as the dedicated modules [12] (Image Acquisition, Image Restitution and Robot Interface) are plugged onto the motherboards. It is therefore easy to increase the transputer network by adding TRAMs and linking together more motherboards.

Image processing : Image acquisition and restitution are implemented with commercially available TRAMs (respectively IMS B429 and IMSB437). The restitution module visualizes the processed images on a screen, an important function for the programmer during the development phase. The image acquisition module can capture grey level images at video rate and has two signal processors for on the flow preprocessing (e.g. 2D convolution...). The onboard T805 can be used for feature extraction. A bottleneck may arise if one needs to transfer the whole image to other transputers for further processing. We experimented a throughput of 5.95 images/s via one serial 20Mbit/s link for image acquisition and 2D convolution with 6x6 matrix of 512x512 pixels 8 bit grey level images. This transfer rate can be optimized when transfering the image on two or three links. Our emphasis being on the control of arm movements, in the first experiments we voluntarily keep the image processing simple enough (object, lightning, resolution) to extract the image features in real time with the only acquisition module. Though a single module can multiplex the entries of several cameras, it's necessary to associate a module to each camera for real time purposes.

Robot Control : The core system is constituted of a sizeable transputer network built with general purpose TRAMs (right now up to 18 T80X). This architecture allows a parallel implementation of the ANNs. The size of the Transputer network is held variable and can be adjusted to the complexity of the AsNN by adding the appropriate number of transputers. The parallelization of the neural network is worked out up to the point of meeting the real time constraint.

Robot Interface : We implemented a dedicated transputer module realizing the robot's interface between the transputer world and the robot's control bus. Control signals, joints angles and velocities, are sent to the individual joints controllers and actual joint positions and velocities are returned.

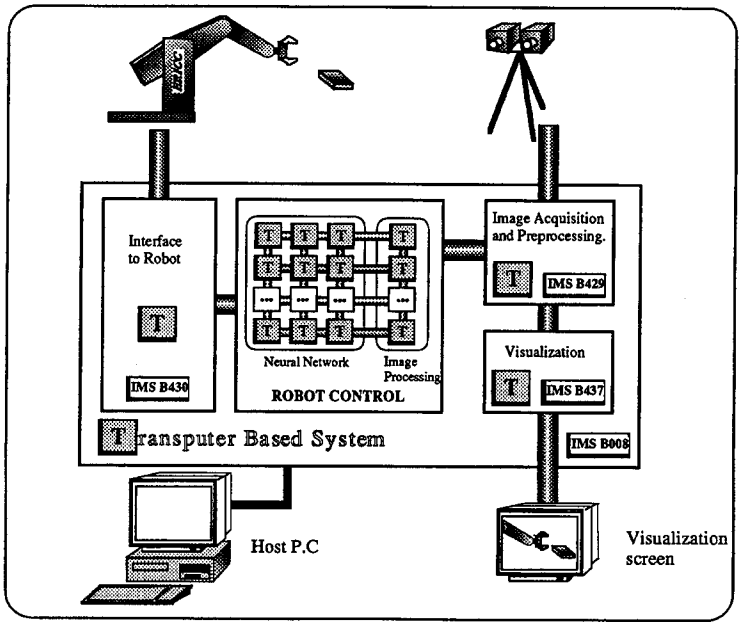


Fig 3.1 : System Architecture.

Future extension : The active positioning of the cameras is not yet implemented. A robotic head has been designed and is currently under development at the Ecole Nationale d'Ingénieurs de Metz (see figure 3.2.). The control architecture will be the one described above.

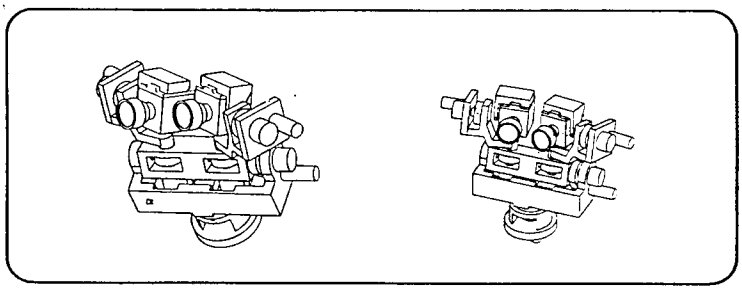


Fig 3.2 : Robotic Head.

4. Simulation Environment

To finalize a neural network application, usually a good quantity of trials have to be done. Different parameters must be adjusted like the number of neurons, their interconnections, the learning rate... To appreciate the effect of each modification in parameter, the network has to be trained, meaning that a set of inputs have to be repeatedly presented until the network has converged. Therefore, a simulation of the robot environment has been built to serve for development purposes of the AsNN and also to limit the strain put on the manipulator to keep it from premature ageing.

Based on the real environment, the robots kinematics and the cameras have been synthesized on a 3D Silicon Graphics workstation. Figure 4.1. shows the real robot and the simulated one. But the core system, the transputer network implementing the AsNN, continues to be the same for both simulation and real experiment (see figure 4.2). The transputer network and the graphic workstation have therefore to be linked together to exchange visual information and control signals. A serial link is sufficient to transfer the images features and control signals representing very little information. A near future extension is a fast SCSI2 link to transfer images.

When a satisfactory behavior of the simulated system is achieved after defining and training the AsNN, one can replace the workstation with the real robot environment and test the behavior of the AsNN with the experimental setup.

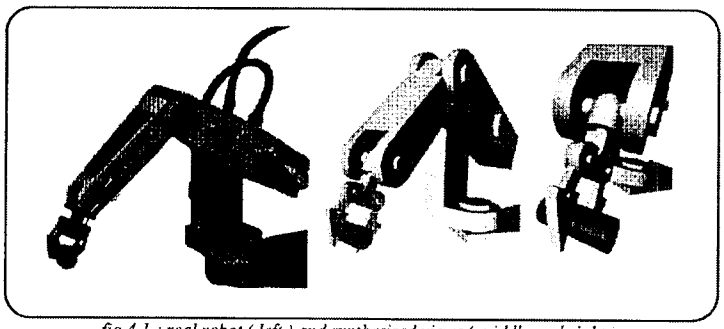


fig 4.1 : real robot (left) and synthesized views (middle and right) at identical angle adjustments and different camera positions.

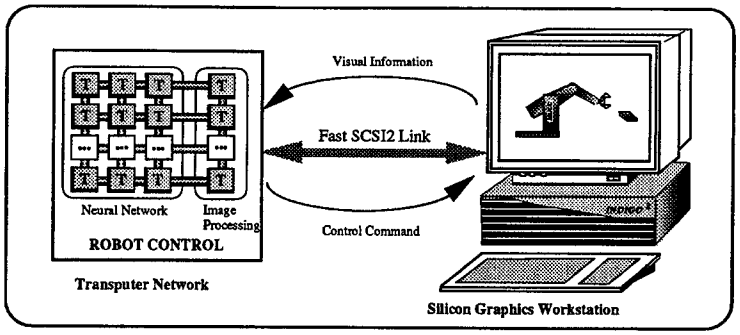


Fig 4.2 : Simulation Architecture : the robot and cameras are now simulated.

5. Results and Perspectives

We illustrate the feasibility in terms of processing time with results for a simple arm positioning task with a static camera using a modified Kohonen network. The robot moves only two joints in a plane with the purpose to superimpose the gripper on a sphere (see figure 5.1.a). The camera covers the workspace of the two joints. The image processing module extracts the coordinates of the center of gravity of the sphere. The Kohonen network, of size 10x20 neurons, is trained with 6000 epochs. Figure 5.1.b shows the discretization performed by the neural network within the limits of the two joint workspace. The response of the network (angular coordinates) implemented on a single T805 30Mhz 4Mb transputer module is obtained in less than 5 ms, which is acceptable for real time implementation.

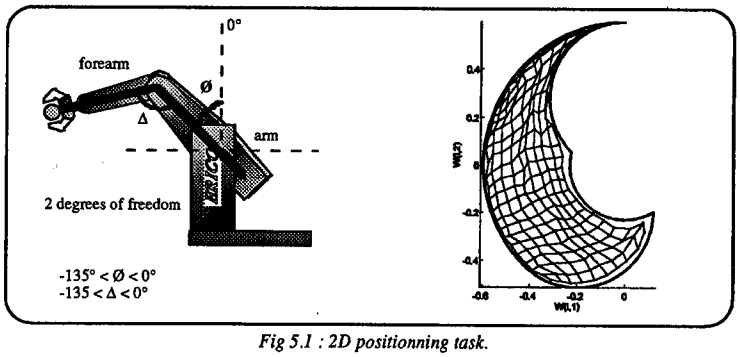


Fig 5.1 : 2D positionning task.

Training cycles take from 20 to 50 ms/cycle, depending on the size of the neighborhood considered (note that this figure is not prohibitive). Once trained the neural network acts as a look up table and the precision obtained depends directly on the number of neurons used. With 200 neurons and a 500x500 pixels camera at a distance of 2m the mean precision obtained is 8 pixels.
This experiment is too simple to be satisfactory, but it serves as a testbed to demonstrate the proper working of the whole system and to evaluate the computational needs of more complex applications.

Perspectives are to replace the T805 implemented core system with T9000 technology, which should offer substantial performance improvements [13]. Further development will concentrate on the control of the robotic head performing the positioning and adjustment of the cameras.

Acknowledgements

This work has been partially funded by the Région Alsace under agreement between Université de Haute-Alsace, Conseil Régional d'Alsace and Telmat Industries.

Bibliography

[1] Marr D. A theory of cerebellar cortex. *Journal of Physiology*, 202:437-470, 1969.

[2] Albus J.S. A theory of cerebellar function. *Mathematical Biosciences*, 10:25-61, 1971.

[3] Dufossé M., Julien J.-P., Para C., Sicard C., Commande neuromimétique pour un robot anthropomorphique. *Science Technique Technologie* , n°15, 1990.

[4] Miller W. T. III, Sutton R. S., Werbos P. J., Neural Networks for Control, *MIT Press*, 1990.

[5] Ritter H.J. , Martinetz T.M. , Schulten K.J. , Neural Computation and Self-Organizing Maps, *Addison-Wesley*, 1992.

[6] Kuperstein M. , Adaptive visual-motor coordination in multijoint robots using parallel architecture, *IEEE Conf. on Robotics and Automation*, pp 1595-1602, March, 1987.

[7] Miyamoto H., Kawato M., Setoyama T., Suzuki R., Feedback error learning neural network for trajectory control of a robotic manipulator, *Neural Networks*, n°1, p 251-265, 1988.

[8] Ritter H.J. , Martinetz T.M. , Schulten K.J. , Three-dimensional Neural Net forLearning Visuomotor Coordination of a robot arm, *IEEE Transactions on Neural Networks*, Vol 1, pp 131-136, 1990.

[9] Walter J.A. , Schulten K.J. , Implementation of Self-Organizing Neural Networks for Visuo-Motor Control of an Industrial Robot, *IEEE Transactions on Neural Networks*, Vol 4, pp 86-95, 1993.

[10] Kohonen T. , Self-Organization and associative Memory, *Springer-Verlag*, 1984.

[11] Auger J.M , Parallel Implementation on Transputers of Kohonen's Algorithms, *Euro Courses in Computing with Parallel Architectures*, September 10-14, 1990.

[12] Inmos, Transputer Development and iq systems databook, *SGS-THOMSON*, Second Edition 1991.

[13] Inmos - SGS-Thomson Microelectronics. The T9000 Transputer Product Overview, 1991.

MULTI-LAYER NEURAL NETWORKS ON A PIPELINE OF TRANSPUTERS APPLICATION TO AUTOMATIC ALL-NIGHT SLEEP STAGES QUOTING

PINTI Antonio ** & GROSSETIE Jean-Claude *

** UNIVERSITE DE MULHOUSE
FACULTE DES SCIENCES ET TECHNIQUES -TROP
68093 MULHOUSE - FRANCE

* COMMISSION OF THE EUROPEAN COMMUNITIES - TP 361
INSTITUTE FOR SYSTEMS ENGINEERING & INFORMATICS
21020 ISPRA (VA) - ITALY

Phone : + 39 332 78 98 78
Fax : + 39 332 78 53 78
Email : antonio.pinti@cen.jrc.it

Abstract

This paper describes the implementation of a multi-layer neural network based on the back-propagation algorithm using the mean gradient technique implemented on a T-NODE MIMD machine. This parallel software has been used to optimise the automatic human' sleep stage quoting.

Keywords : transputer, Occam, parallelism, neural networks, sleep stages classification.

1. INTRODUCTION

The study has been executed for a project in the field of automatic classification of stage quoting of human'sleep by using methods of connectionist pattern recognition to obtain a real-time exploitation of sleep signals [1].

The signals are generated by electrodes stuck on the patient's body, and are transmitted to an information system by a dedicated real-time acquisition system. Numerical processing of these signals is then used in order to give an output X vector of 17 time-frequency parameters each 30 seconds time interval. Each vector is subsequently classified into one of the 6 stages of sleep defined by an international standard [2]. A histogram which represents the stage of the sleep as a function of time is traced in order to visualise the real-time evolution of the stages of the patient's sleep. This tracing allows to determine the patient's pathology.

Artificial Neural Network (ANN) techniques applied on very large data-base implies long computation time with respect to the learning phase algorithm [3]. In order to reduce this computation time, the neuronal algorithm has to be parallelised taking into account the hardware and software constraints. Hardware constraints are mainly due to the access data transfer delay ; while, software constraints need to be split into elementary tasks, each of them being implemented in best way with respect to the present available hardware .

This paper consists of five sections. Section 2 presents the multi-layer neural network algorithm. Section 3 describes the parallel implementation of this connectionist algorithm on a pipeline of N-transputers T800. Section 4 presents the analysis of human'sleep stage quoting and gives the

experimental ratio results of classification with neural network and K nearest neighbours. Finally, section 5 concludes this paper.

2. DESCRIPTION OF A FULLY CONNECTED MULTI-LAYER NEURAL NETWORK

Connectionist methods are already used in many applications for pattern-recognition [4, 5, 6]. They require a considerable amount of computing power during the training phase. Once this phase has been finished, the parameters of the connectionist model are fixed and the model to able to perform the tasks for which it has been trained for.

To use a fully multi-layer neural network as a classifier ; 2 data-bases systems are needed, each of them being classified and must be representative of the specified type of problem. When non-linearities are met, the data-base, which has been set-up, need to include an important number of patterns associated with the model.

On one hand, the initial data flow has to be used is used to learn a network. On the other hand, the second data flow is used to define the so-called ratio of generalisation with respect to the computed network. The following figure 1 presents the back-propagation algorithm, illustrating the forth and back flow of data through a neural network :

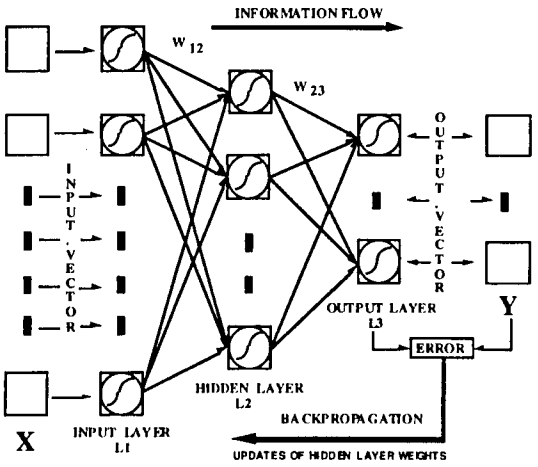


Figure 1 : Architecture of multi-layer neural network with 3 layers.

Different methods for adjusting the connection-weights between the neuron layers are available and are presented in different scientific books [7, 8]. Back-propagation methods based modified gradient algorithm has been used.

The modified back-propagation algorithm is using each vector (pattern) data as an input for the network at stage L1, in order that the network has to be trained by this data flow. Using the Euclidean norm, the mean error between the computed output and the desired output is evaluated. This latter mean error is used for updating the internal connection weights. Afterward, the previous back-propagation algorithm using a variable step μ is used in order to speed-up the convergence process. This variable has to fulfil the following condition : $0 < \mu < \mu_{max}$.

Phase training description : a learning algorithm is implemented using the Fletcher-Powel gradient optimisation technique [9] :

If K is the total pattern number, M the total number of class, then the optimisation problem may be stated as :

find the connection coefficients in order to minimise the cost function e_{mean} define by :

$$e_{mean} = \frac{1}{K} \sum_{k=1}^{K} (\frac{1}{2} \sum_{m=1}^{M} (l_{2km} - y_{km})^2)$$

- Stage 0 : initialisation of the weights using random value and with $\mu = 0.1$ and $e_{precedent} = \infty$.

Remark : For simplify the k-index has been omitted for all the mathematical quantities.

- Stage 1 : at a given pattern k, the L1 input unit layer has to received a net input.

$$\left[L_1\right]_p^1 = [X]_p^1 \qquad \text{with } X \in \Re^p \text{ (p = number of parameter)}$$

-Stage 2 : compute of the L2 hidden layer.

2-a $$\left[L_2^{input}\right]_n^1 = \left[W_{12}\right]_n^p \left[L_1\right]_p^1 \qquad \text{with L2} \in \Re^n \text{ (n = number of neuron in hidden layer)}$$

2-b $$\left[L_2^{output}\right]_n^1 = sig(.)\left\{[I]_n^n \left[L_2^{input}\right]_n^1\right\}$$

with the thresholds function sig(.) being defined by :

$$\text{sig(h)} = \text{th(h)} = \frac{e^{\alpha h} - e^{-\alpha h}}{e^{\alpha h} + e^{-\alpha h}} \text{ is a sigmoïd function with } \alpha = 1.$$

with $W_{12} \in \Re^{p*n}$.

- Stage 3 : compute of the L3 output layer.

3-a $$\left[L_3^{input}\right]_m^1 = \left[W_{23}\right]_m^n \left[L_2^{output}\right]_n^1$$ with L3 $\in \Re^m$(m=number of neuron in output layer)

3-b $$\left[L_3^{output}\right]_m^1 = sig(.)\left\{[I]_m^m \left[L_3^{input}\right]_m^1\right\}$$

with $W_{23} \in \Re^{n*m}$.

-Stage 4 : compute of the usual error using Euclidean distance, with $Y \in \Re^m$.

4-a $$e = \frac{1}{2} \sum_{m=1}^{M} \left[L_3^{output}{}_m - Y_m\right]^2$$

4-b $$[E]_m^1 = \left[L_3^{output}\right]_m^1 - [Y]_m^1$$ the $[Y]_m^1$ components are y_j, j=1..m, which are

defined by $y_j = \begin{cases} +1 \ \ if \ \ j = C; \\ -1 \ \ otherwise. \end{cases}$ C ($C_k \in \{1,2,...M\}$) is a class of X_k vector.

- Stage 5 : compute the gradient of the current error e with respect of all the elements of the W_{12} and W_{23} matrix.

$$\Delta W_{32}{}^{i}_{j} = \frac{\partial e}{\partial W_{32}{}^{i}_{j}} = {}^{t}\left[L_3^{output} - Y\right]_1^m \frac{dsig}{dh}(.) I_m^m [a]_m^n \left[L_2^{output}\right]_n^1$$

with $[a]_m^n$ having the generic element defined by $a_\alpha^\beta = \delta^{\alpha i}\delta_{\beta j}$

and i ∈ (1, .., n) , j ∈ (1, .., m)

$$\Delta W_{12}{}^{i}_{j} = \frac{\partial e}{\partial W_{12}{}^{i}_{j}} = {}^{t}\left[L_3^{output} - Y\right]_1^m \frac{dsig}{dh}(.) I_m^m W_{32}{}^{n}_{m} \frac{dsig}{dh}(.) I_n^n [a]_n^p \left[L_1^{output}\right]_p^1$$

with $[a]_n^p$ having the generic element defined by $a_\alpha^\beta = \delta^{\alpha i}\delta_{\beta j}$

and i ∈ (1, .., n), j ∈ (1, .., p)

- Stage 6 : loop on stage 1 until all the training vectors k have been presented to the input L1 of the neural network.

-Stage 7 : compute the mean error and mean gradient after the previous loop.

$$e_{mean} = \frac{1}{K} \sum_{k=1}^{K} e_k$$

$$\Delta W_{12}{}^{i}_{j_{mean}} = \frac{1}{K} \sum_{k=1}^{K} (\Delta W_{12}{}^{i}_{j})_k$$

$$\Delta W_{23}{}^{i}_{j_{mean}} = \frac{1}{K} \sum_{k=1}^{K} (\Delta W_{23}{}^{i}_{j})_k$$

- Stage 8 : compute the updated of new weight : W_{12} and W_{23}.

If $e_{mean} \leq e_{precedent}$

Then

- $\mu = 1.15 * \mu$ with the constrain being : $\mu \in \Re$ and $0 < \mu < \mu_{max}$.
- For hidden-to-output connections the mean gradient descent rule gives for all components of W_{12} and W_{23}matrix:

$$W_{23old} = W_{23}$$
$$W_{23} = W_{23} - \mu \Delta W_{23mean}$$

- For input-to-hidden connections, the mean gradient descent gives :

$$W_{12old} = W_{12}$$
$$W_{12} = W_{12} - \mu \Delta W_{12mean}$$

Else

- $\mu = \frac{\mu}{2}$
- $W_{12} = W_{12old}$
- $W_{23} = W_{23old}$

-Stage 9 : return on Stage 1 while ($e_{mean} > e_{desired}$).

Phase test or classification description :

Once the algorithm has converged, the connection weights are fixed. The input vector X of p parameters is multiplied with matrices consisting of elements which represent the weight of the connections computed in the previous phase of training. The associated L3 output vector is thus available. The class of vectors X can be determined by analysing the values of L3 vector. Each neuron of output layer is associated with only one class that has to be identified. The index of component of the L3 vector which have the maximum value, will give the index of the associated class of vector X : class (X) =m with m = max value ($L3_1$, .., $L3_M$).

3. IMPLEMENTATION OF A NEURAL NETWORK ON A PIPELINE OF TRANSPUTERS

Our work consisted to parallelise the neural network on a MIMD distributed-memory T-NODE of 64 transputers T800-20 which are connected to a Unix host system by a ITFTP-32 card which contains also a T800-20. Each transputer of this network has a local memory of 4 mega bytes not shared.

The main objective was to reduce the time of convergence of the multi-layer network during the training phase [10]. This allows a fast simulation of different architectures of connections from a large database (more than 10000 vectors in each database : training, test). Thus the automatic classification of sleep can be optimised.

The program has been developed in the parallel language OCCAM to exploit the whole computing power of the transputers [11]. The entire calculations are made in simple floating point precision (32 bits), which facilitates the processing of any data type.

To have a easy implementation of the algorithm on N transputers, we have used a architecture of pipeline of transputers. With this architecture, it is easy to add a transputer on the network without modification of parallel algorithm and communication between transputers. The following figure shows the synoptic of hardware system and a communication process:

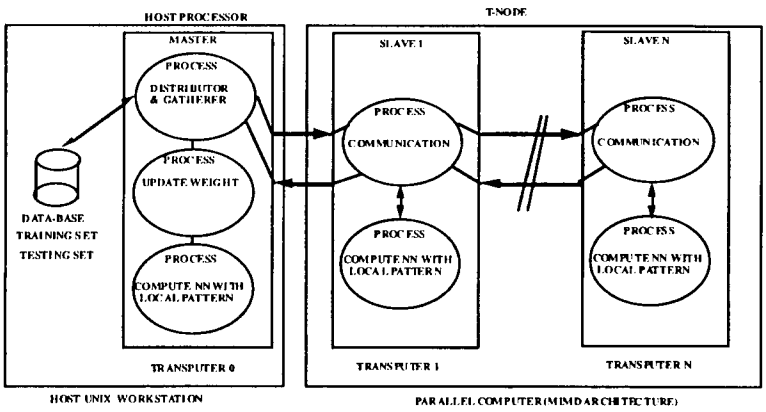


Figure 2 : architecture of system and mapping processes onto network of transputers

The back-propagation algorithm used for the training of the neural network requires that all vectors X of the set of training vectors are treated identically. For each vector X presented at the input layer L0 of the network, the propagation to the subsequent layers L1, L2, ..., LN has to be calculated. The error between the desired and the actual output vector of the final layer LN has to be determined.

In order to parallelise the neural network we used the parallelization mode of Single Program Multiple Data (SPMD) appropriate for this type of problem. The implementation of the program on an arbitrary number of transputers has been facilitated by a process which distributes the computation load according to the number of available processors. During the initial phase the process numbers all processors thus allowing them to identify themselves in the protocol of data-transmission. All transferred data packages are identified by a header of the message. The connections of the neural network are identical on each processor.

The host processor of the ITFTP-32 interface board performs the following tasks: it reads the entire training database, it splits this database into N sub-databases and numbers each package from 1 to N (N represents the number of available processors on T-NODE), it distributes these sub-databases on the different processors. Finally it sends the weights of the connections between the neurons to each processor in order to have the same configuration of the multi-layer neural network on each processor. Each processor of the T-NODE calculates the square error of the neural network for each vector X of the corresponding sub-database. This mean error is added one by one and is finally read by the host processor. This one corrects the connection weights using mean gradient back-propagation and transmits the new weights to each slave-processor. This procedure is repeated until a determined number of iterations has been reached or an initially defined minimum error has been obtained.

SPEED-UP AND EFFICIENCY

The most interesting performance measure of multiprocessor implementation is the speed-up factor. $S(N) = T(1)/T(N)$ where S(N) is the speedup factor depending upon the number of processors N and T(N) is the time taken by a computation on an N processor system. The parallel algorithm implemented on the 65 available transputers shows a speed-up S(65) of 63 in comparison with the same algorithm implemented on a single transputer T(1). The program efficiency is equal to 96%. 4% is this efficiency is losing in communication time (send a weight and back-propagation of error) and especially in time for updating the connection weight by a host transputer.

4. APPLICATION TO AUTOMATIC HUMAN'SLEEP STAGES QUOTING

A first study on automatic human'sleep analysis was made by LENGELLE R. with a first connectionist method [12]. The most appropriate pattern algorithm for this problem has been well identified. However, the best performance for each tested method was not obtained, due to the lack of computer power.

To carry out this automatic sleep analysis with multi-layer neuronal network parallelised, two large pattern data-bases are created. This data-bases are composed of vectors of 17 spectral-parameter labelled with stage of sleep. They have a manual quotation of 23 nights of sleep of 23 patients.

The training set are constituted of 12 entire nights of sleep which have a total of 12455 vectors. The testing set are constituted of 11 entire nights of sleep which have 11745 vectors. This testing set is used to evaluate the performance of generalisation.

Each of this nights have been quoted manually by 10 experts. The comparison of different quotation have shown a good percentage inter-expert equal to 88%. This percentage will be the limit value of good classification in testing phase.

The following table indicates the number of vectors in each stage of sleep for the 2 whole of data.

class	awake	stage 1	stage 2	stage 3	stage 4	stage 5	total
learning set	1374	257	5179	1140	1599	2906	12455
testing set	1300	227	5178	672	1548	2820	11745

Tableau 1 : number of vector in each class.

The number of sample in each class is representative of phenomena to study. The figure 3 shows the percentage of good classification on the two data-bases (training set and testing set) tested in function of the number of neuron in hidden layer:

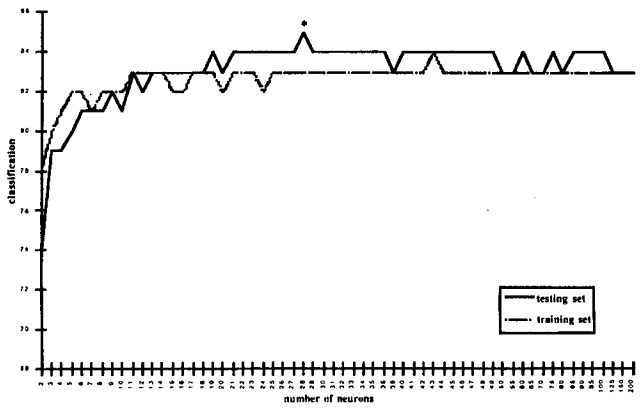


Figure 3 : Determination of the optimum number of neurons in hidden layer (correct percent).

The network had 17 neurons in a input layer and 6 neurons in a output layer (one for each stage of sleep). The number of neurons in a hidden layer was varied from 2 to 200. The graphic of figure 3 needed more than 400 training epochs. Each network was tested with different random values for synaptic weight. 1000 iterations are necessary for each convergence set. We have taken for each network, the synaptic-weight which gave the best result of classification on testing set.

The analysis of the curves shows up to 11 neurons in a hidden layer, the network gave the best performance on a training set. This result is reversed beyond this value.

The optimum capacity of neuronal network is obtained for a number of neurons equal to 28 in a hidden layer. The ratio of good classification on a testing set is equal to 85%. The optimum number of neurons for learning set is equal to 42. The ratio of correct classification on training set is equal to 84%.

This experimental result shows that the phenomena is complex and difficult to model. The ratio of good classification approaches the maximum ratio of 88% achieved by inter-expert analysis on the wholes-data available.

For the present automatic sleep analysis, we used a neural network of three layers (one input layer, one hidden layer, one output layer) which require 17, 28 and 6 computing elements respectively.

To show the advantage of multilayer neural network, we compared other pattern recognition algorithm for the sleep analysis problem: With the neural network we obtained 85% of successful classification while the method of the K nearest neighbours (KNN) optimal succeeds in 79% for eighteen neighbours when using 12000 training vectors in the data base (corresponding to eleven nights of sleep).

Moreover, neural network method allows to classify a vector within 25 ms which is 130 times faster than with the KNN method.

5. CONCLUSION AND PROSPECTS

As a conclusion, an important reduction of global computing time has been achieved using the described parallel algorithm. This algorithm, already used in the laboratory for automatic classification of human'sleep stages, has been optimised by exploiting the computing power of the T-NODE parallel super-computer. For example, the C.P.U. time required for the training phase has been divided by a factor of 63 when compared with a sequential transputer system.

Due to the implementation of the parallel software on the T-NODE computer, a considerable number of about 12000 training vectors has been used for the training phase corresponding to the present neural network system. As a consequence, for a given number of classes, and using the plane mapping, a very good estimation of the partition zones has been achieved with 85% of successful classifications during the working phase.

Furthermore, the parallelised software could be used for other applications where pattern recognition is needed, the input discretized signals being of any type corresponding to other physical phenomena. But the C.P.U. time required is presently too important with respect to the present state of art of the parallel architecture.

However, without any modifications of both described algorithm and associated software, the convergence C.P.U. time will still be reduced, since new microprocessors and parallel architecture are foreseen. First tests using the T9000-20 transputer demonstrated that the speed-up factor can be improved by a factor of 4, and thus, signal analysis which have not been performed will soon be possible.

6. REFERENCES

[1] PINTI A. (1993) Algorithmes paralleles sur transputers applications en neurosciences, Ph.D. thesis, Mulhouse-France.

[2] RECHTSCHAFFEN A., KALES A. (1968) A manual of standardised terminology, techniques and scoring system for sleep stages, Public Health Service, US. Gover. printing office, Washington-USA.

[3] ALEKSANDER I., MORTON H. (1993) Neural computing, Ed. CHAPMAN & HALL.

[4] HETCHT-NIELSEN R. (1991) Neurocomputing, Ed. ADDISON WESLEY.

[5] CICHOCKI A., UNBEHAUEN R. (1993) Neural networks for optimisation and signal processing, Ed. WILEY.

[6] LIPPMAN P.R. (1989) Pattern classification using neural networks, I.E.E.E. communications magazine, November, 47-64.

[7] FREEMAN J. A., SKAPURA D. M. (1992) Neural networks algorithms applications and programming techniques, Ed. WESLEY.

[8] HERTZ J., KROGH A., PALMER R. G. (1991) Introduction to the theory of neural computation, Ed. WILEY.

[9] FLETCHER R., POWELL M. J. D. (1963) A rapidly convergent gradient descent method for minimization, The computer Journal, vol. 5, 163-168.

[10] FOO S. K., SARATCHANDRANDRAN, SUNDARARAJAN N. (1993) Parallel implementation of back-propagation on transputers, proceeding of IJCNN'93, Nagoya-Japan.

[11] HOARE C.A.R. (1988) Occam 2 reference manual, Ed. PRENTICE HALL.

[12] LENGELLE R., SCHALTENBRAND N., GAILLARD P. (1990) Pattern recognition using networks comparison to nearest neighbour rule, proceeding of IFAC, Nancy-France.

Section III : Applications

REAL APPLICATIONS ON THE TN300 T9000 BASED SCALABLE PARALLEL COMPUTER

M. Dominique DUVAL
TELMAT Multinode
ZI - 6, Rue de l'Industrie
F-68360 SOULTZ
FRANCE
email : duval@telmat.fr

Summary : The availability of the SGS-Thomson T9000 processor is one of the major events for the parallel computing community in 1994. Designed by Inmos Ltd. in Bristol (a company of the SGS-THOMSON Group), it has been integrated into several machines, including the TN300 series by Telmat Multinode a leading European parallel systems manufacturer. This paper presents the first results obtained by Telmat and some customers on the March 1994 versions of the T9000, in a work supported by the Commission of the European Communities in the GP-MIMD Esprit Project. It also compares these preliminary results with the one obtained on the previous generation T800 processors.

Keywords : parallel application, T9000 transputer, Telmat TN300, performances

1. Some words about the transputers.

The first transputer was designed in 1984, as a scalar processor with 4 high speed communication links. The first generation included a high performance arithmetic processor (T414) and a high performance floating point processor (T800). The T800 common clock speed is of 20, 25 or 30 MHz, achieving an instruction throughput of 10, 12.5 or 15 MIPS, and a sustained floating point rate of 1.5, 1.75 or 2.25 Mflops.

The T9000 on its side, though designed on the same principles as the T800, integrates a 32 bits Super-scalar processor, a 64 bits floating point unit, a virtual channel processor, 16 KBytes of cache memory and four 100 Mbits/sec parallel communications links. At 50 MHz, the processor can reach a peak performance of 200 MIPS and 25 MFlops (averaging 15 MFlops sustained). With 4 accesses per cycle to the 16 KBytes on-chip memory, the total memory bandwidth is of 800 MBytes/sec. T9000 processors may be linked together via a packet routing chip, the C104. A single C104 enables up to 32 T9000s to communicate at full bandwidth, and allows to build larger networks by cascading layers of switches.

2. The Telmat TN300 Series

Telmat, a leading European manufacturer of computer systems has been active in the design, manufacturing and marketing of parallel systems since 1985. Through its own products and the PCI global alliance with Meiko Ltd. and Parsys Ltd., Telmat is well placed in the scientific, commercial and embedded systems areas. The last system designed and manufactured by Telmat is strongly based on the T9000 processor, and allows to connect scalably and efficiently up to 512 processors.

The software tools available include full compiling systems for C and Fortran, the PVM, Parmacs and MPI message passing interfaces and debugging tools like Inquest or Paragraph. In addition to these, Telmat provides a very performing run-time kernel called RUBIS, which allow to manage tasks and threads, channels and ports, segments and pages in a parallel environment. Particularly adapted for fine grain parallelism, RUBIS has been partially supported by the European Commission in the Esprit Supernode 2 project.

3. Applications and numerical results

The hardware used for this series of experiments was in a very intermediate state, as systems were being fixed and deliveries were planned for Summer 1994. The T9000 processors were still running at 20 MHz with a peak performance directly proportional to this clock frequency. Compared with a T800 at 20MHz and 2.5 MFlops, the performance ratio between this version of the T9000 and the T800 used for the experiment should be (25/2.5)*(20/50)=4. This theoretical ratio will be compared with the ones measured and conclusions will be drawn concerning the full speck T9000. Obviously during the conference, a more updated version of these results will be presented fully.

Here follows a list of applications and kernels which have been ported and related conclusions. They are good case examples of real application requirements and allow to evaluate the response of the T9000 processor to their respective requirements at the time of the benchmark run.

3.1 Whetstone

This well known benchmark is a synthetic program which intends to represent a typical set of numerical applications. It is composed of 8 small modules, included in loops associated with different weights (loop bounds). Each loop is relative to one particular type of operation.

The total execution time is measured to evaluate the processor performance in K Whetstones/sec. According to the kind of operations performed, the performance of the T9000 is obviously varying. Integer operations are very well executed, but floating point arithmetic and libraries are not optimized yet. The cache influence is also significative. All original timings were divided by a factor of 3.5 to 4.0 when 8 KBytes or 16 KBytes of internal cache were used.

Whetstones decomposed :

Decomposing the operations involved in the whetstone benchmark has a lot of interest, as it allows to evaluate the respective performances associated.

Type of operation	T800 20 MHz - in ms	T9000 20 MHz 16 K cache - in ms	Ratio
array elements	237	51	4,65
array as parameters	1701	251	6,7
conditional jumps	1913	493	3,8
int arithmetic (+,-,*)	3851	793	4,8
trigonometric functions (sin, cos, atan)	16530	5761	2,86
procedure calls	12436	3155	3,94
array references	6540	1484	4,4
standard functions (sqrt, exp, log)	12783	5933	2,15
Total	56 sec	17,92 sec	3,12

We can see that effectively the T9000 at 20 MHz implements better the integer operations and array manipulations than standard functions. Looking at the respective time passed in every part of the benchmark, it appears clearly that the overall result will be greatly improved when this problem will be solved.

One can also see that the most usual ratio is greater than the expected 4 number, which is due to a good usage of the internal cache by the Inmos compilers. It should be noticed that the total time for the same T9000 but without using the internal cache was 63,39 sec, which stresses even further the conclusion.

3.2 Dhrystone

Dhrystone is a synthetic benchmark based on the distribution of source language features observed in the literature. It intends to represent non numerical, system type programs (operating system, compiler). It does not include any arithmetic operation. It is written in C and includes multiple strings, pointers and structure manipulations (initialization, copy, ...). The results are given in Dhrystone per second.

Results :

	T800	T9000 without cache	T9000 16 K cache	Ratio T9/T8
Dhrystone	79,95 sec 6253 dhry/s	60,52 sec 8262 dhry/sec	19,227 sec 26005 dhry/sec	4,16

Again, we can see a better than 4 ratio between the T9000 and the T800 operating in the same conditions, with a large impact of the T9000 internal cache.

3.3 Communications

Obviously, the results presented above concern mono processor benchmarks, which reflect the quality of the T9000 processor itself, but show little about the actual architecture of the TN300 series. The beast way to evaluate this is by first benchmarking separately the inter processor communication facilities, and then a fully parallelized application making use of both the T9000 and the communication infrastructure.

The following benchmarks give results obtained when connecting two T9000 20 MHz processors through a 30 MHz C104. In the first example the communication is monodirectional, while in the second it is bidirectional. Both are working with links at 50 Mbits/sec.

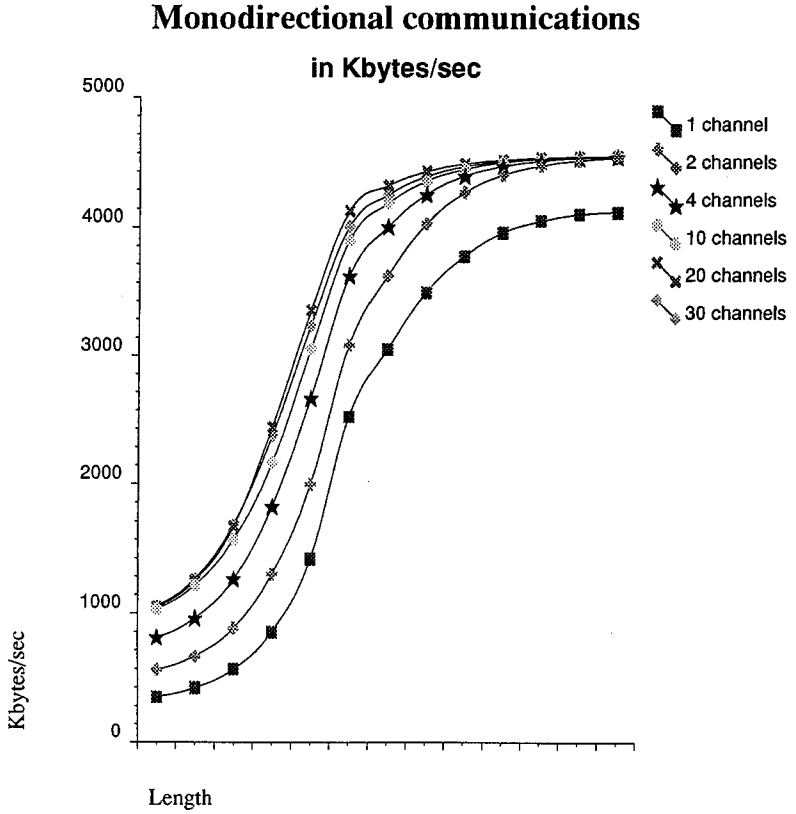


Fig. 1 Monodirectional Communications between two T9000 processors

The Fig.1 diagram clearly shows the impact of the number of virtual channels on the communications between two processors. This is particularly true for small messages. The effective sustained bandwidth triples when 30 virtual channels are used instead of 1.

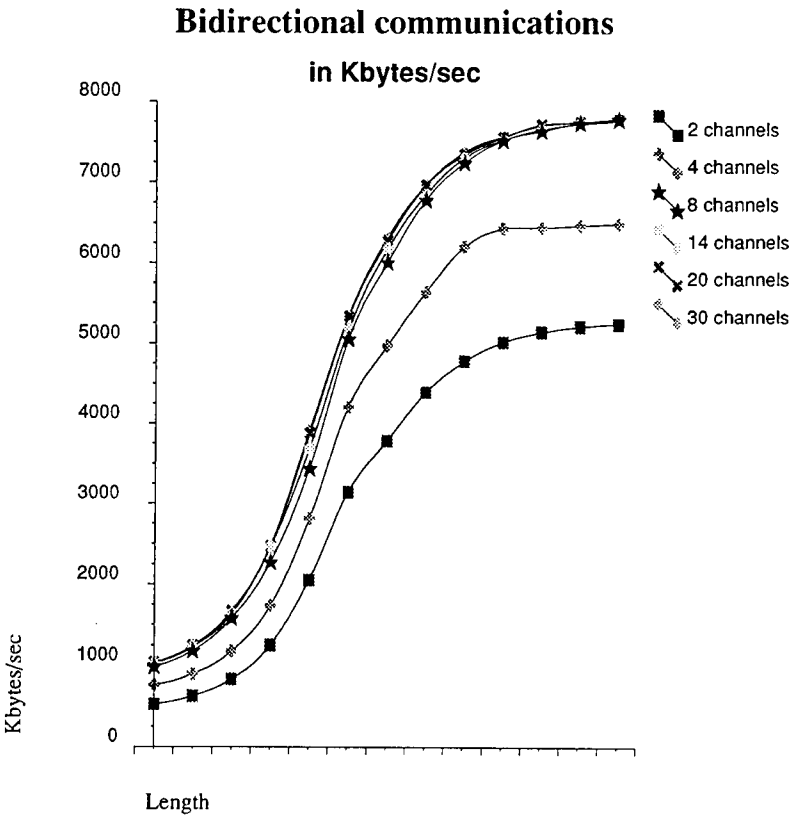


Fig.2 Bidirectional communications between two T9000 processors

The Fig.2 diagram reflects the improvement expected on the communication bandwidth when using bi-directional communications. It also stresses even more clearly the importance of using multiple virtual channels. We also have shown separately that the message latency is of less than 1 microsecond.

In general, we can see that there is a sufficient bandwidth for large fine grain parallel applications, even with relatively small messages. The TN300 System provides an unmatched inter processor communication technology, which impact will be clearly identifiable on real parallel applications.

3.4 Real parallel applications

As we evaluated separately the performances of the T9000 processor and of the inter connection network, we can now assess the effectiveness of the full architecture on a real application. These evaluations are currently being made in the Esprit Project GP-MIMD by Telmat Multinode, jointly with Southampton University (UK), Collège de France (F), CERN (CH) and Conservatoire National des Arts et Métiers (F).

The *Southampton University* activity is concentrating on evaluating the TN300 systems with benchmarks selected from the PARKBENCH and GENESIS suites, and methodologies recommended by the PARKBENCH committee [1]. The initial focus of the PARKBENCH benchmarks is on the new generation of scalable distributed-memory message passing architectures for which there is a notable lack of existing benchmarks. For this reason, the initial benchmark release concentrates on Fortran 77 message passing codes using the widely available PVM message passing interface for portability. The GENESIS benchmarks [2] mainly mirror the PARKBENCH ones, but with additional versions of codes using the PARMACS message passing interface. Future versions of PARKBENCH will undoubtedly adopt the proposed MPI [3] interface when it becomes generally accepted, but MPI versions of the GENESIS benchmarks are planned for release in July 1994, and will be used as soon as they are available.

The *Parallel Computing Group at Collège de France* has developed computational methods and techniques needed for implementing large simulation codes on the transputers. Experience has been gained through extensive code running on Telmat T.Node Systems. Simulation of a broad class of problems were done on a common basis provided by the experimental high energy physics software environment. The delivered processing power scales from 20 to 500 MFlops.

Through its activity the group was able to appreciate the intrinsic limitations of current sequential systems and it is also able to consider the benefits afforded by the developments of a new generation of massively parallel super computers based on a new class of transputers.

At the time of writing of this paper, researchers at Collège de France and Engineers at Telmat are busy evaluating the results of porting the GEANT particle physics collisions simulation package on the Telmat TN300 System and numerical results are already encouraging.

Conservatoire National des Arts et Métiers on their side are currently implementing their MIXAGE3D code on the Telmat TN300. MIXAGE3D tackles the problem of convective diffusion in heterogeneous media. The code has a sequential structure and its parallelisation is made following an adequate geometrical patching. Runs on the T800 based T.Node have shown that it was possible to obtain on a grid of 7x7 T800 processors computing powers comparable to that of a CRAY XMP. The TN300 appears to be particularly well adapted to its implementation.

The MIXAGE3D code solves numerically a parabolic partial differential equations (PDE) system. These equations describe many physical, chemical and biological phenomena. They are typically used in the simulation of diffusion in transitory or stationary media. The modelization developed by CNAM is based on stochastic differential equations. Evolutionary operators are defined, associated to the studied physical phenomena. Later on, based on a "MIXAGE3D" methodology of these operators, the general solution of the original system is built.

Numerical results and benchmarks will be detailed during the conference, showing the great potential of the TN300 System architecture.

References:

[1] R.W. Hockney and M. Berry (Editors). Public International Benchmarks for Parallel Computers, PARKBENCH Committee : Report-1, February 7, 1994, to appear in *Scientific Programming*, 1994.

[2] C. Addison, J. Allwright, N. Binsted, N. Bishop, B. Carpenter, P.Dallos, D. Gee, V. Getov, A. Hey, R. Hockney, M. Lemke, J. Merlin, M. Pinches, C. Scott and I. Wolton. The GENESIS Distributed-Memory Benchmarks. Part 1 : Methodology and general relativity benchmark with results for the SUPRENUM computer. *Concurrency : Practice and Experience*, 5(1):1-22, 1993.

[3] Message Passing Interface Forum, PMI : A Message-Passing Interface Standard, University of Tennessee, Computer Science Technical Report CS-94-230, April 1994, to appear in *The International Journal of Supercomputer Applications*, Volume 8, Number 3/4, 1994.

Specification of Distributed Hard Real-Time Systems

L.Carcagno - M. De Michiel - D. Dours - R. Facca - B. Sautet

I.R.I.T
118 Route de Narbonne
31062 Toulouse Cedex
FRANCE

e-mail dours @ irit.fr
Fax 61 55 62 58

Abstract

Distributed hard real-time systems are inherently complex and safety-critical. The concerned systems are dedicated application systems as in signal, space information, sound processing, robot control...

These systems are called « hard real-time » because the interaction with the environment raises time constraints that they must respect not to lose control on the process they manage. To cope with huge processes, distributed systems are required.

Although a number of specification techniques for real-time systems have been reported in literature, most of these formalisms do not adequately address to the constraints that the aspect of distribution and hard real-time impose on specifications. Further, an automatic verification tool is necessary to reduce human errors in the design process.

In this regard, this paper describes an executable specification language for distributed hard real-time systems. The analysis of timing properties is possible through the distributed synchronous data-flow computing model that we have defined. A designing tool is used to specify applications and to verify the description as well as the synchronization mechanisms, and to perform transformations of the concurrent code to match a target generic architecture.

1. Introduction

Real-time applications cover a lot of promising industrial fields (aeronautics, space projects, robotics, ...). Most of these applications are inherently complex and safety-critical. They use a set of sensors and actuators which can be diversely located, to obtain necessary information about mobile driving, supervision task performing or robot controlling, for example.

Theses applications involves systems that maintain a permanent interaction with their environment, reacting to inputs coming from this environment by sending outputs to it. These systems have **transformational characteristics** because of the importance of the calculus to be performed and **reactive** ones, to respond to discrete events. They are also characterized by **strict timing constraints**, to make sure that the system processing rate is consistent with the sensor data-flow rate. Reaction time must also be shorter than given bounds to keep the system mastering the process. They are called **"hard real-time"**, because the interaction with the environment raises time constraints that they must respect not to lose control on the process they manage. They are different from the **"interactive"** systems, that can require short response time without big consequences if not respected.

Such systems are developed following the classical steps : specification, design and implementation but new approaches are necessary to take the real-time dimension into account.

Specification and preliminary design are the first steps of development methods. But no existing method has already become a standard in this field. The S.A.R.T method which is derived from the S.A method (Structured-Analysis [1]), and which is an extension to the real-time case, is the oldest and the most widespread of real-time system design methods. Some research groups [2-3] have nearly attained the same result. In addition to the functional approach, they introduce finite state machines to model the system behavior.

The object approach, more recent, will certainly gain ground. The H.O.O.D method [4], which is recommended by the ESA for space system design, is based on objects and abstract machines. HRT HOOD [5] is a structured design methodology dedicated to hard real-time systems. It is an extension of HOOD that includes cyclic and sporadic activities.

But whatever the approach may be, the main difficulty comes from the **design step** and more especially from the programming design step which has to determine the internal performances needed to satisfy the specified constraints. One must schedule tasks or objects, regroup them into different modules, and hierarchically organize them in such a way as to meet the timing constraints.

The programming design can take several forms according to the various hypotheses about the system we have to realize. Synchronous approaches [6-9] make temporal problem study simpler, but when timing hypotheses are not confirmed, i.e the action execution time cannot be considered as instantaneous, then another approach must be considered.

The classical method in asynchronous approach is to process sequential tasks by a real-time kernel. But it cannot always ensure the execution time when tasks are complex. In that case, the system is similar to a transformational one. It can be seen as a complex operator which has to provide outputs in response to inputs that may be periodic or sporadic.

Parallel programming languages such as ADA or OCCAM have the advantage of possessing the specific primitives indispensable for real-time system programming, but they also have some disadvantages like non determinism, difficulty to debug programs and difficulty to parallelize them [9-10].

To conciliate reactive and transformational aspects, a mixed approach can be used [11]. This approach takes the ESTEREL model [7] to describe the reactive part of a system. This part can be seen as a synchronous scheduler which has to cadence the evolutions of the activities of the transformational part. A real-time multitasking executive, inferred from Sceptre [12], takes the transformational part into account.

A purely reactive or lightly transformational processing can easily be implemented on a centralized system. But to cope with huge processes, **distributed systems** are required. If the system has to take into account strict timing constraints i.e, if it cannot control its environment when these constraints are not met, to ensure the execution time it is necessary to process the tasks in parallel. But using classical parallel architectures is problematic because time is taken into account as a constraint instead of a performance criterion. So it seems better to define the architecture according to the application [13]. But it presents some inconveniences : it is difficult to implement it, it is necessary to define an architecture for every application, it is impossible to modify an application during its life cycle. So it would be better to define an architecture adapted to a whole application field. A modular and reconfigurable architecture must be defined from which a dedicated architecture can be inferred.

Although a number of specifications techniques for real-time systems have been reported in literature, most of these formalisms do not adequately address to the constraints that the aspect of distribution and hard real-time impose on specifications. Further, an automatic verification tool is necessary to reduce human errors in the design process. If the verification consists in checking of certain properties of a program (e.g. safety properties, liveness properties, fairness), time is then reduce to the ordering of events. It is a logical time that do not use metric properties of time. This is true even for many reactives systems [14]. The verification tool works with computational model of programs, as a rule. But for the development of distributed hard real-time systems, the verification tool must calculate an execution time by determining number of cycles of physical clock for example.

The inherent characteristics of these systems, such as occurence of events at specially separated components of the system, significant message transmission delay, absence of global clock, and hard time-bounds make it very difficult to develop an appropriate specification formalism.

In this regard, this paper describes an **executable specification language** for distributed hard real-time systems. It is intended to be used at the system specification level rather than for the initial requirements specification.

The analysis of timing properties is possible only on a suitable model of computations, simulation can not give the required level of confidence. This is the reason why a **distributed synchronous data-flow computing model** has been defined [15] as well as a specification tool that allows the capture of the description according to the model and the constraints to respect.This tool verifies the synchronization mechanisms. Unfortunately, the modular structure of the specification will surely be different from the modular structure of the implementation.This is why an **architecture compiler** [16] has been defined in order that the designer could concentrate on the logical specification of its problem without being preoccupied with the implementation, that has to respect temporal constraints (response time ...). This compiler is going to automatically translate the system specification in a implementation configuration. At the realization level, this representation can be seen as a particular configuration of a **generic architecture** [17] that allows to build the dedicated machine.

2. Computing Model

The major aim of this section is to accurately model both behavioral and operational aspects of distributed hard real-time systems handling events as well as continuous computations. For example, measurements (sampled data) are received from sensors, and processed to produce commands as outputs for actuators. Events are sporadic and sampled data are periodic.

A distributed system can be properly described in terms of processing components that are loosely coupled by communication channels. The processing components communicate by exchanging messages with their environment or with others components.

The potential parallelism of an application can be described through this scheme. But, one of the main hard real-time environment problems is that the maximum execution time must be known to be able to produce a correct result before a certain bound. The problem is not really the processing speed but the ability to predict the system timing behavior. A deterministic behavior must be ensured and the running time must be mastered.

The predictability of the systems is obtained by the choice of a **synchronous model** [10]. But the hypothesis of a purely synchronous formalism implies a higher speed for the computation than for the desired reactions, and the synchronization between inputs and outputs [18]. Thus the running time must be lower than a fixed bound.

The underlying computing model allowing a determinist evaluation of the running time is represented by a structure :

$$S = (\Omega, \Xi, \Gamma, \xi)$$

Ω is a set of nodes of the system. A node will be called a **module**. A module exchanges information either outside, or with other modules, using input and output ports. It has a cyclic running (data input, process, data output).

Ξ is a set of the unidirectional communication channels of the system. Each channel connects exactly two modules of the system. A channel carries a stream between an output port of a module and an input port of another one. A **stream** is defined as series of values, each value can be read only once. The successive values of a stream are buffered into a FIFO. The FIFO is variable and bounded size-wise.

A stream is specified through both : the values it carries and a total ordering of the "instants" at which these values are available at the input port.

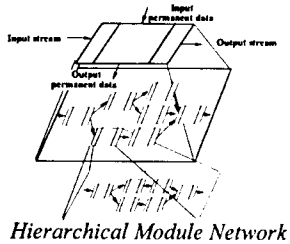
$\xi : \Xi \times \Omega \rightarrow \Omega$ is a partial mapping representing the interconnection among the modules.

Intuitively, for each module $m \in \Omega$ and channel $c \in \Xi$, $\xi(c,m)$ (if defined) is the module connect to m via channel c.

Streams are only top-down, so the interconnection function ξ can be represented by an acyclic graph, which visualizes a partial execution order, induced by a relation of timing precedence. In such a graph, the modules of the same row are executed according to a true parallelism, while modules placed on the same path are executed in a pipelined mode. It is a module network.

Γ is a communication channel to broadcast messages for which there is no temporal relation between the production time of the message and its utilization time. This channel connects every modules of the system and carries permanent data. A **permanent data** is always available and simultaneously present all over the system. It can be read as often as required by the modules that use it, but only the producing module can modify it.

A module may be a structure. So a system is modeled by a hierarchical structure. At the lower level of the hierarchy, a module is called **elementary module** and performs a sequential algorithm. An elementary module has a cyclic running and the processing complies with the data flow activation rule; a cycle can start only when all its inputs are available. The module data at each from its input ports, performs a calculus and produces results stored in its output ports.



Hierarchical Module Network

While asynchronous models do implicitly or explicitly refer to some external and universal time reference, the notion of time is completely different in this synchronous model. The time is local to a given module. The data-flow activation rule vouches the synchronous approach if streams going into a module have the same flow rate and the computing time of the module is lower than the incoming flow rate [18]. The advantage of the synchronous approach is that the non-determinism of external communication is strictly concentrated on this mechanism.

The first constraint is imposed to the "specifier". To automatically go from the specification to the implementation, we must be able to parallelize a sequential algorithm in an automatic way. The parallelization must be got according to the model. An elementary module must be decomposed until its computing time is lower than a fixed time imposed by the flow rate. So a just sufficient parallelism is obtained, that lowers down the realization cost.

To be able to parallelize an algorithm according to the model and evaluate the highest computing time while compiling, the activity of an elementary module is described by a static algorithm, i.e. that always computes the same process independently of the data. In a **structured imperative language**, a static algorithm is made with blocks that can be either an instruction sequence, a selection, or a bounded loop.

An activity described by a static algorithm can be split up into π-blocks. A **π-block** is either a block, or a set of blocks bound together by a recurrence [19]. The activity thus split up is still an module network. The application can be split up until every module has a processing time lower than a fixed time, so a **just sufficient parallelism** is obtained [20].

But it is not always possible to obtain a just sufficient parallelism. It means that the algorithm is not realistic, i.e. it cannot satisfy real-time execution constraints using the current technology.

We must either change the algorithm, or change the timing constraint, and if nothing is possible then the application is unrealistic.

3. System specification

In specification, it is necessary to capture functional requirements and timing properties, and express then through a specification language.

To describe a hard real-time application according to the model, a modular and hierarchical language, R.S.D.L,. has been defined. It is a synchronous specification and programming language that ensures a determinist behavior [21].

A designing tool is used to specify applications and to verify the description as well as the synchronization mechanisms. It also performs transformations on the concurrent code to match a target generic architecture.

To illustrate the description principles of a real-time system, an example will be presented.

3.1. Executable specification language

It is a three level hierarchy language. The first level is to describe the interface with the environment, the sensors and actuators are defined there, as well as the temporal constraints. Furthermore the system body is defined, i.e. the decomposition into sub-systems. The second level is to describe the part of the **transformational** aspect related to the parallelism of the application using **network modules**. The last level is to describe the sequential **behaviour** of an **elementary module**. The instructions are ADA type instructions adapted to the model (no recursion, for loops...). The only constraint is to declare the input and output characteristics and the timing constraints when describing the system interface. The system temporal behaviour can be predicted because of the language limitations, so the running times of the modules can be bounded.

Some aspects of the modeling , though they can be textually described, are intuitively visual. Thus, it appeared interesting to allow the user to describe directly in a graphic way the different levels of the hierarchy of the application, and the structural aspect. The behavioral aspects of the elementary modules will be described textually using a sequential code.

3.2. Designing tool

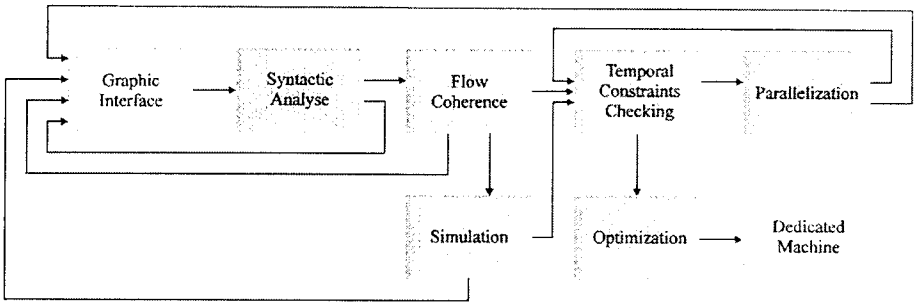
The designing tool combines a specification tool and an architecture compiler.

The **specification tool** allows the designer to specify his application according to the model and to verify the description and the synchronization mechanisms. The designer describes an application in a hierarchical way and specifies timing constraints through a **graphic interface**. After a classical **syntactic analyze** the tool verifies the functional structure and the **flow coherence**. Once the errors corrected, the designer can **simulate** the running of the specified system.

The **architecture compiler** checks the timing constraints. If they are not satisfied, a parallelization is necessary. The method consists in associating an **allocated time** to each module. If a module has a running time higher than the allocated one, it should be decomposed with a **parallelization** method that takes into account the flow rate and the critical response times. If the problem has a solution, the decomposition algorithm always finds one [15]. If despite the parallelization the real-time constraints are not satisfied the compiler asks the designer to modify the description. When the description complies with real-time running constraints, it has to be optimized. **Optimization** techniques using the model particularities (acyclic graph in which every module performs a repetitive process at a frequency determined by the time allocated to the module) are then applied. At first they allow the minimisation of the number of rows in the graph, in order to reduce the response time, and then the gathering of modules to minimise the computational structure (number of processors). A processor computes a module that correspond to the gathering of several initial modules. It is obvious that this gathering must take into account the allocated running times and the relations of precedence in the graph [15].

The optimization needs transformations that have been defined in a way that the semantics of the system is maintained [20]. So it is ensured that the transformed system has the same functionalities as the initial system specified by the designer.

Once these steps are completed, the **architecture configuration** is automatically determined and the code of every elementary module loaded to obtain the dedicated machine.



Designing tool : R.S.D.T.

3.3. Specification example

This system, inspired from [22], has to control the cruising speed of a vehicle and the opening of the engine injection system. The user can intervene on the system through push buttons and observe the system state through indicator lights.

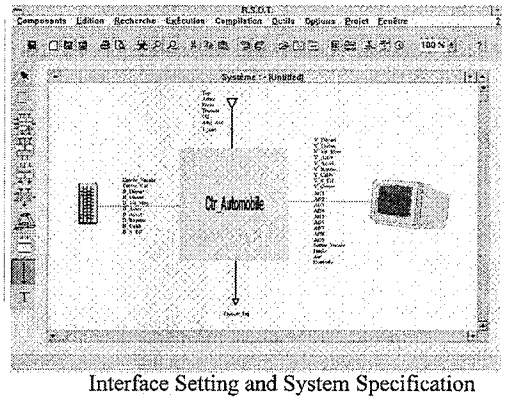
3.3.1.Specification of the associated package.

The types and sub-types used by the system are described in this package.

```
package CONTROLE_AUTOMOBILE_PKG is
   const
     ARBRE_PERIOD : TIME = 20 ms;
     PERIOD_INJ : TIME = 75us;
     PERIOD_INJ : TIME = 75us;
     MOTEUR_PERIOD:TIME = 5 ms;
     PERIOD_TEMP : TIME =75ms;
     PERIOD_COMMANDE : TIME = 500ms;
   subtype TEMPERATURE is INTEGER range -55 to 125 ;
   subtype ANGLE is INTEGER range 0 to 90 ;
   subtype RATE is INTEGER range 0 to 6000 ;
   type    O2_SAMPLER is (riche, pauvre);
   subtype TOURS is INTEGER range 0 to 1000 ;
end CONTROLE_AUTOMOBILE_PKG;
```

3.3.2.Specification of the system interface with the environment.

When the system is described using the visual language, the characteristics of external and internal information associated to inputs and outputs are defined with attributes. The textual version of these characteristics is automatically generated.



Interface Setting and System Specification

For the previous exampie :

ARBRE
external PULSE **rising edge frequency** 10 ms **cadency** ARBRE_PERIOD;
internal PULSE **used as** STREAM;

TOP:
external PULSE **rising edge frequency** 2.5 ms **cadency** MOTEUR_PERIOD ;
internal PULSE **used as** STREAM;

O2:
external LEVEL **frequency** MOTEUR_PERIOD;
internal O2_SAMPLER **used as** STREAM;

FREIN :
external LEVEL **frequency** MOTEUR_PERIOD;
internal LEVEL **used as** STREAM;

TRANSM :
external LEVEL **frequency** MOTEUR_PERIOD;
internal LEVEL **used as** PERMANENT_DATA;

ANG_ACC :
external CONTINUOUS **frequency** MOTEUR_PERIOD;
internal ANGLE **used as** STREAM;

T_EAU :
external CONTINUOUS **frequency** PERIOD_TEMP;
internal TEMPERATURE **used as** STREAM;

OUVER_INJ :
external LEVEL **frequency** PERIOD_INJ **response time** 110 μs (TOP);
internal LEVEL **used as** STREAM;

B_ACTIV, B_ACCEL, B_REPRISE :
external PULSE **frequency** 250ms **cadency** PERIOD_COMMANDE;
internal PULSE **used as** PERMANENT_DATA;

B_CALIB :
external PULSE **frequency** 250ms **cadency** PERIOD_COMMANDE **delay**20ms;
internal PULSE **used as** STREAM;

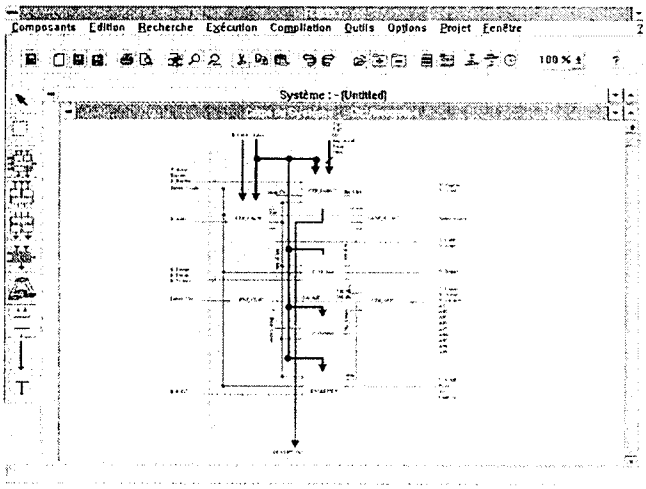
V_ACTIV, V_ACCEL,V_REPRISE,V_CALIB :
external LEVEL **frequency** PERIOD_COMMANDE;
internal LEVEL **used as** PERMANENT_DATA;

Different levels of description can be distinguished in the previous figure. The less tinted part corresponds to the system interface. An input module (i.m.) collects an input and generates the internal information corresponding to the information given in the interface specification. An

output module (o.m.) generates an external information suitable for the peripheral or the actuator that uses it according to the internal information and to the characteristics of the information type acceptable by the output user. Input and output modules are automatically generated by the design aid system. It has to be noticed that here the input TOP is sampled by two different input modules. A response time constraint has been formulated for the output **OUVERT_INJ** from **TOP**, so the input has to be sampled at a higher frequency. But another input module will keep sampling the **TOP** signal at the rate fixed by the external signal if other consumer modules do not need a higher rate.

3.3.3.System body : Decomposition into sub-systems

When defining the system body, all the sub-systems that make it up are described with their links. These links, when existing, are compulsorily permanent data type, because, if not, they are in the same sub-system.



CTR_AUTOMOBILE System Body

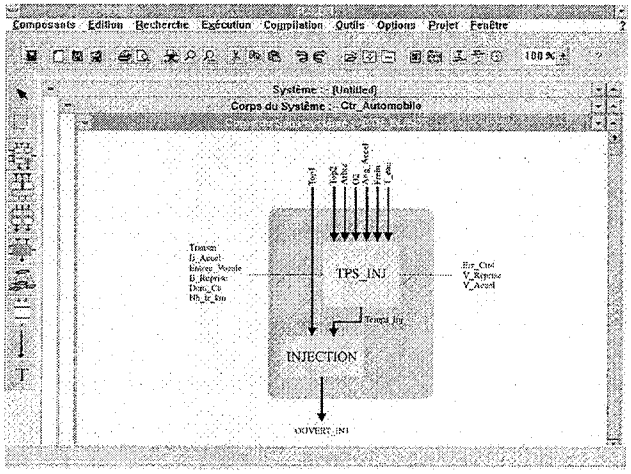
The following attributes are communicated to the system :

NB_TR_KM : TOURS **frequency** ARBRE_PERIOD; DEM_CTR: LEVEL **frequency** PERIOD_COMMANDE;

The **CTR_CALIB** sub-system interprets the external commands for measuring or starting the automatic speed controller. It measures, i.e. counts the shaft rounds per kilometer, when the measuring is active, and activates the automatic controller if the request is valid. Furthermore it indicates to the user, through indicator lights, if a received request is valid and considered. This sub-system consists in an elementary module only, so its description is already finished.

The **CTR_INJECT** sub-system is decomposed into two modules :

- **TEMPS_INJ** is a network module that determines the petrol injection time into the engine. Thus it can be decomposed.
- **INJECTION** is an elementary module that commands the effective opening of the valve, for a delay equal to **OUVERT_INJ**, in synchronism with the **TOP1** signal.



CTR_INJECT Sub-system Body

The following attributes are communicated to the system :

TEMPS_INJ : TIME **frequency** MOTEUR_PERIOD ;

As an example, here is the **INJECTION** elementary module R.S.D.L code :

```
elementary module INJECTION is
var   TOP0 : BOOLEAN;
      TOP : BOOLEAN := FALSE;
      TIME_COUNT : TIME;
begin
      OUVERT_INJ := FALSE;
      loop
      input TOP1, TEMPS_INJ;
      TOP0 := TOP; TOP := TOP1;
      if OUVERT_INJ then
            TIME_COUNT := TIME_COUNT + PERIOD_INJ;
            if TIME_COUNT>=TEMPS_INJ then
                  OUVERT_INJ:= FALSE;
            end if;
      elsif TOP0 and not TOP then
            OUVERT_INJ:= TRUE;
            TIME_COUNT := 0;
      end if;
      output OUVERT_INJ;
      end loop;
end INJECTION;
```

4. Conclusion

In the context of hard real-time applications we have proposed a computing model yielding both determinism and distribution capabilities. It is a distributed synchronous data-flow computing model in which a system is decomposed into concurrent deterministic modules that cooperate in a deterministic way. Deterministic concurrency is the key to the module development of distributed hard real-time systems.

To specify such applications we have developed a modular and hierarchical language allowing the expression of these applications, as a distributed network of modules. A specification tool is used to specify applications according to the model using this language, and to verify the description of the flow coherence. An architecture compiler performs transformations of the concurrent code in order to match a target generic architecture.

This work can be considered as a step towards a direct synthesis of distributed hard real-time systems from their specifications.

5. References

[1] De Marco, T., Structured Analysis and System Specification (Yourdon Press, 1978).

[2] Mellor, S.J. and Ward, P.T., Structured Developement for Real Time Systems (Yourdon Press, New-York 1985, 1986).

[3] Hatley, D.J. and Pirbhai, A.I., Strategies for Real Time Specification (Dorset House, 1987)

[4] H.O.O.D., H.O.O.D : Hierarchical Object Oriented Design (reference manual,1990).

[5] Burns, A. and Wellings, A.J., HRT HOOD : A Design Method for Hard Real-Time Ada 9X Systems, Towards Ada 9X, Proc of 1991 Ada U.K. International Conference, I.O.S. press 1992

[6] Harel, D., Statecharts : A visual formalism for complex systems, in : Science of Computer Programming (North-Holland, june 1987) volume 8, number 3.

[7] Berry, G. and Cosserat, L., The synchronous programming language ESTEREL and its mathematical semantics, in : Proc. seminar on concurrency (Springer-Verlag LNCS 197, 1985)

[8] Caspi, P. and al., LUSTRE, a declarative language for real-time programming, in : Proc.conf. on principles of programming languages (Munich, 1987).

[9] Le Guernic, P. and al., SIGNAL : a data flow oriented language for signal processing, in : IEEE-ASSP 34(2) (April 1986).

[10] Berry, G., Real Time programming : general purpose or special-purpose languages, in : IFIP World Computer Congress (1989).

[11] Andre, C. and L.Fancelli, L., A mixed implementation of a real time system, in : EUROMICRO 90 (August 27-30, 1990).

[12] B.N.I : Bureau d'orientation de la Normalisation Informatique., Sceptre : proposition de noyau normalisé pour les applications temps réel, in : TSI (1984) Vol. 3, n°1,.

[13] Stankovic, J.A., Misconceptions about Real-Time computing : a serious problem for next generation systems, in : IEEE Computer (October 1988).

[14] Kurki-Suonio, R. and al., Real-time Specification and Modelling with Joint Action. Proc. 6th Int. Workshop on software specification and design (Como, 1991).

[15] Carcagno L.et al. «From Specification to Implementation of a Real-time System», EUROMICRO 92 Conference, Paris, 1992.

[16] De Michiel. M.«Recherche de la Configuration Optimisée d'une Architecture cible pour une application Temps-Réel», thèse de doctorat UPS, Toulouse III, 1994

[17] Feki. A.«Architecture Parallèle Générique pour la réalisation de systèmes Temps-Réel», thèse de doctorat, UPS, Toulouse III, 1993.

[18] Berry, G. and al., Programmation synchrone des systèmes réactifs : le langage ESTEREL, in : TSI (1987) Vol. 6, n° 4 .

[19] Dasgupta, S., Computer architecture : A Modern Synthesis, in : Volume 2 : Advenced Topics - John Willey and Sons (1989).

[20] Magnaud, P., Méthodologie et outil de conception d'une architecture parallèle temps réel (Thèse de doctorat de l'Université Paul Sabatier, Toulouse III, Décembre 1990).

[21] Carcagno L. et al. «R.S.D.L : a real-time System description language», Rapport interne I.R.I.T, 1992.

[22] Airiau R. et al.«V.H.D.L : du langage à la modélisation», Presses Polytechniques et Universitaires Romandes, Col. Informatique, 1990.

A Flexible Approach to Parallel, Real Time Graphics.

R. J. Cant, Department of Computing, The Nottingham Trent University, Burton St., Nottingham NG1 4BU, United Kingdom. Email rcc@uk.ac.ntu.doc.

Summary

Real time graphics applications are varied but are often implemented on standard hardware systems which fix the algorithms that must be used. To better match the algorithm to the application a parallel, software based, architecture is proposed. Whilst it is expected that critical parts of the software will be re-implemented for each application, a default set of algorithms is suggested, together with a work distribution scheme. The processor loading implications of various levels of performance are analysed and the implications of future developments in hardware and algorithms are considered.

Keywords

Computer Graphics, Real Time, Parallel, Rendering.

(1) Introduction

Real time graphics applications are amongst the most resource-hungry in the whole computing field. Up until now the need for processing, bandwidth and response time capability has almost always been met by specialised hardware design. For a discussion of such a design see [1]. As a result of this, software and system designers are not always able to use the most appropriate algorithms for the task in hand - since they must always use what their hardware supports. In the past, the designers of specialised graphics applications could often afford to develop their own hardware to overcome the limitations of standard systems [2] but the level of investment required and timescales have now made this impracticable.

The limited impact of graphics standards is also a consequence of this situation - since the "native library" of a graphics workstation always provides more power. The effect of this is made worse by the fact that the algorithms supported by hardware have not remained constant over time - resulting in a "temporal portability problem" as well as the usual spatial one.

For example, early versions of graphics workstations did not provide Z buffering in hardware and so real time applications would have been forced to use other methods of hidden surface removal which require structured databases. Later the hardware Z buffer became available and these databases became obsolete.

To overcome these difficulties we must take a new approach to achieving the performance levels required by real time graphics. If special purpose hardware is discarded, the only available alternative is to use general purpose hardware in a parallel architecture. In the past all attempts to move in this direction have had at least one of two major defects. Either the system created has not been capable of real time response (usually because of bandwidth limitations in interprocessor communications) or the hardware configuration adopted had the effect of pre-determining the algorithms which could be used in exactly the same way as with a special purpose hardware system. We must define a set of requirements for our design which specifically exclude these possibilities.

(2) System Requirements and Expected Benefits

We are looking for a parallel graphics architecture which is real time capable, algorithm neutral and scaleable.

Algorithm neutrality will hopefully provide a good design "match" over a wide range of applications.

The scaleability requirement is important because we recognise that algorithm neutrality will extract a heavy price in terms of reduced performance compared to "dedicated" systems. To compensate for this we need to increase the number of processors and, whilst the cost of this can be offset against savings in design effort, any absolute ceiling on system performance will be disastrous.

We would also like to use "off the shelf" hardware and to program in a high level language as far as practicable. This will provide portability and reduce the cost of migrating to newer technology as it becomes available.

Obviously such an architecture will be (in the first instance) less cost effective in terms of hardware than existing "dedicated" designs but as time progresses hardware becomes cheaper, whilst the (re)development costs saved will become more and more significant.

(3) The Implications of Algorithm Neutrality

One objection that might immediately be raised to this concept is "if one is to be algorithm neutral - than what progress can one actually make?"

There are two answers to this. The first is that the implementation of graphics algorithms in a parallel architecture in itself gives rise to a relatively limited number of ways in which work can be distributed. One of our aims is thus to define a work distribution scheme which has minimal impact on algorithm choice.

The second is that graphics algorithms follow certain common patterns even though their detailed design may differ (i.e. design structures may be the same even if the meanings of their components vary).

Given that the system is to be algorithm neutral the items which we will provide will be different from the usual software package or hardware system. There are two categories, long lived and short lived items.The long lived items are (in order of decreasing generality):

(i) a prescription of how hardware should be connected together;

(ii) a work distribution scheme;

(iii) a basic graphics software package, provided in source code form which application developers can adapt to their needs.

The short lived items are:

(iv) hardware support to allow images to be displayed;

(v) software to allow (i) and (ii) to be implemented on a specific processor.

It is envisaged that the actual processing will make use of commercially available modular hardware systems. Currently there are two such systems on the market, the Transputer TRAM and the 320C40 TIM. It should be emphasised that nothing in this work will actually restrict implementation to these two systems. Any system with suitable connectivity and bandwidth would be useable.

Certain choices about algorithms will also have to be made. These choices will be made on the basis of providing maximal decoupling between one part of the algorithm and another and providing maximal generality.

For example it should be possible to change the shading algorithm without affecting the rest of the design and the number of possible shading algorithms supported by the design should be maximised.

This emphasis on generality does not mean that the sample system will attempt to be "all things to all men" at the detailed level. There will not be a large number of setup options and adjustable parameters to enable the system to be "customised" to the requirements of an individual application. This kind of generality usually creates more problems than it solves because it involves the creation of a new "language" which the user must learn before the system can be used effectively. Instead we will use algorithms which have wide applicability. Where customisation is required the intention is that it should be done by the application developer re-writing that part of the code which needs to change. This imposes a constraint of simplicity on the implementation.

(4) Parallelism and Graphics

Parallelism in graphics (and for that matter in any application) can be implemented in two ways. The task itself can be subdivided and allocated to separate processors or the data set can be subdivided and each piece given to a separate processor. In graphics there is a further choice within the "data subdivision" method in that either the input data (polygons or other geometric structures) or the output data (pixels or screen regions) can be used. It is also possible to combine any two - or all three of these subdivision principles within the same system.

The traditional, dedicated, graphics system does in fact use all three. At the top level there is an algorithm based decomposition which is implemented as a pipeline structure. Within each process there may be further - data based - parallelism. At the top level the input data set will be subdivided whilst the rendering process is likely to be accelerated by some kind of screen subdivision, with a processor being allocated to each part. In addition to this it is usual to employ specialised processors for the different parts of the pipeline in order to maximise performance. This last point is the main cause of the lack of flexibility (in terms of the ability to vary algorithms) of the traditional workstation but it is important to notice that some aspects of the parallelisation scheme also act as a constraint. In particular the algorithm based decomposition dictates the overall "shape" of the algorithm to be used and even prevents detailed changes, if they affect the overall workload of a subtask. The various data based parallelisms add to the inflexibility because of their incorporation into the the pipeline scheme. (They increase the specialisation of the pipeline steps that incorporate them.)

If we eliminate algorithm based parallelism as an option then we must choose some form of data based parallelism. The sheer size of the rendering task in real time graphics makes a purely input data based scheme difficult to construct unless the screen coverage of individual data items can be constrained in some way. This kind of constraint undermines what we are trying to achieve and so we are forced into output data based (screen based) parallelism as a necessary component of the current project.

Although this procedure is not completely "algorithm neutral" it cannot be too damaging because the process of scan conversion (the essential part of the rendering process) is itself a screen subdivision process (dividing the screen into lines and pixels) so one is merely anticipating what must happen later anyway.

This kind of parallelism on its own is inadequate because it cannot take effect until there is knowledge of where on the screen each component of the image will be. Therefore it does nothing for the early steps in the process. We must therefore speed up the early part of the graphics pipeline (geometry transformations and the like) by using input data based parallelism. At some point in the system this parallelism must be removed in order that a complete image can be displayed. If this is done early it will form a constraint on the methods that can be used since it enforces a certain "shape" on the algorithm and will probably fix the intermediate data format in which this process happens. The best way out of this problem is to leave the recombination until the very last minute - at which point we know that the data will be in the form of pixels. This technique has the further advantage that it permits different algorithms to be used for different objects within the scene.

We therefore arrive at an architecture in which processors are arranged in a rectangular array with one direction corresponding to different parts of the screen and the other corresponding to different objects or scene components. This arrangement for a simple system with the screen divided into four quadrants and three objects is shown in Fig. (1). The unconnected links at the top of the diagram go to the output hardware, which must contain some multiplexing capability to create a single image from the four separate parts. The links at the left hand side are used to provide input to the processors. Note that the upward going links must have sufficient bandwidth to allow an image to be created from their output at at least the frame update rate of the system. It is preferable to be able to output an image from the system at the field rate of the display to avoid an unnecessary transport delay being introduced.

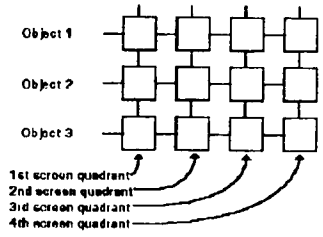


Fig. (1) Hardware Architecture.

This architecture is not particularly efficient if implemented on current hardware but it has three great virtues.

(i) It can be used with any real time graphics algorithms. Because each processor carries the complete set of algorithms any change to the system is a straightforward coding operation with no particular hardware consequences.

(ii) It is totally scaleable. An increase in rendering performance (in terms of resolution, depth complexity or rendering sophistication) is possible simply by increasing the number of processors in the horizontal direction. Conversely a greater capability in terms of numbers of objects rendered can be achieved by increasing the number of "layers" in the system.

(iii) The only major dedicated hardware required is in the output multiplexer, which in itself is a relatively "timeless" device needing to change only with advances in video display formats. Given a processor with a suitable link architecture, system porting should require only recompilation of code (with a few adjustments to "hard" addresses and the provision of an interface to the multiplexer). Systems using a shared memory architecture could also be constructed in principle but bandwidth constraints would limit scaleability.

(5) Graphics Algorithms

Although we wish to have an algorithm neutral architecture we need to provide default software for two reasons. Firstly we need to show that the idea works and secondly most users of the system will not want to supply all the code themselves, only the parts which are "mission critical" for their particular application. The algorithms which will be used fall into two categories: graphics algorithms and parallelism related (work distribution) algorithms.

Some graphics algorithms are intimately connected with the kind of image which is to be presented whilst others are more general and can be used over a wide range of applications. We will concentrate on the latter type here.

Firstly one must decide on the type of algorithm to be used. One can either use a screen based scheme (as is usually the case with ray tracing) or an object based scheme (the normal "graphics pipeline" approach). The disadvantage of the screen based scheme is that each processor must have access to the whole database. The problem of data distribution in parallel systems then emerges. The optimisation of such systems has been considered in the past [3]. In a real time system however, the implied random, high bandwidth, data movement is undesirable. Data movement in real time systems must be regular and predictable if it is to be managed successfully. The upshot of this is that we are forced to concentrate on the object based approach.

Having decided on a scheme which starts with database objects and then progresses to pixels at the end we must also fix the order in which the various sorting operations occur. This is intimately connected to the hidden surface removal algorithm to be used. Denoting the three sorting keys as P (sort by line of sight priority), X (sort by pixel in the X direction) and Y (sort by pixel in the Y direction) there are three different orderings to consider together with the possiblity that the sorts could be "bundled " in some way.

(i) XYP

This group of algorithms includes the commonly used Z buffer. The idea is to scan convert each database object into individual pixel contributions before addressing the problem of hidden surface removal. The hope is that the pixel-pixel comparisons are simple and hence can be performed quickly.

(ii) PXY

This is what Sutherland [4] called the priority list approach. It includes Newell's and Schumacker's algorithms. These algorithms perform well in anti-aliased applications because the depth comparisons are not compromised by quantization. The most efficient and reliable of these algorithms rely on pre-processing the database in ways which may not be possible for all applications. Without pre-processing there is always the possibility of the sorting algorithm being caught out by a particular configuration of objects

(iii) YPX and bundled sorts

Algorithms in this group have been used sporadically throughout the history of real time graphics. In the early days they were used mainly because they afforded the possibility of avoiding the need for a frame buffer. More recently the motivation has been to eliminate the rendering of multiply covered screen areas. This category includes the span buffer method and algorithms which maintain "Y ordered lists" of face data. There are also algorithms in which all three sorts occur together, the best known of which is Warnocks algorithm.

In principal we would like to support all of these methods because different applications may be more amenable to one or the other. It is worth noting , however, that there will rarely be a direct reason to make a particular choice, and so it may be useful to construct an algorithm of our own which is a good match to the hardware architecture. In doing this we must be careful to cater for as many of the indirect reasons which normally lead to a particular choice as possible.

The influences on our choice will include: support for arbitrary database (environment) structures; support for other aspects of image quality; compatibility with parallelism; decoupling from other algorithms; low and consistent execution times and spin offs such as rangefinding, collision detection etc.

In the current context the last consideration can probably be ignored but the others must all be addressed.

The best algorithm for flexibility is the Z buffer since it makes no demands on the general structure of the other parts of the graphics pipeline. Unfortunately, as has been remarked many times before, it is very difficult to combine with proper sampling. PYX algorithms support sampling well but either require extensive database preprocessing or have poor or unpredictable performance when presented with complex environments. They also require that the ordering of objects is maintained as they are rendered - creating problems for a parallel system. Some YPX and bundled algorithms also suffer from this latter problem and they all introduce considerable extra complexity into the scan conversion process.

It follows that none of the commonly used algorithms provides good support for all of the requirements in a cost free way. We must thus think in terms of modifying one of the algorithms to achieve a better fit. The choice lies between modifying the Z buffer to enable it to cope with sampling properly and modifying the space partitioning version of the PYX algorithm to allow it to cope with a more arbitrary database.

The Z buffer suffers from two specific problems which make it incompatible with proper sampling. The first problem is caused by the fact that the depth is itself only sampled at one point. The second problem arises from the fact that only a single "running total" is kept of the contributions to each pixel. If there are two contributions to a pixel then their shade values will be irretrievably mixed and cannot be unscrambled should a third contribution arrive which is intermediate in depth.

To overcome these difficulties two approaches can be taken. The first approach is to antialias by supersampling. This is simple but inefficient since it scales the rendering workload up by a noticeable factor. It is only really viable in the context of a dedicated hardware system. It is also inadequate in the case where there are a large number of small surfaces within one pixel. The second approach is to generate and retain more information about each pixel contribution and to hold on to more contributions until the status of each is clear. Experimental coding in this area has shown that a "perfect" implementation of this idea is also inefficient but that in many practical situations an adequate solution can be found which is likely to be more efficient than supersampling.

The basic problem of the space partitioning PYX algorithm is the need to pre-structure the database. This impracticable in applications such as CAD where the database is continually changing but can be done relatively easily in simple flight simulators where the viewpoint moves around in a pre-determined environment. There are also a number of in between cases where pre-defined objects move around a pre-defined environment and only the object interactions need special treatment. It is possible to extend the algorithm to cover such cases, with each object being separately pre-structured. Provided the geometric relationship between objects remains simple then the algorithm will remain efficient.

Since neither of these algorithms seems to be universal on its own perhaps the best compromise is to combine them. We propose therefore to divide the database up into objects which must be internally pre-structured to allow space partitioning methods to be used for intra-object hidden surface removal. We will also calculate and record depth values so that priority comparisons between objects can be performed on a pixel by pixel basis. Since many of the problems of the space partitioning PYX algorithm relate to the movement of objects whilst most of the Z buffer problems arise from the detail within objects the two algorithms cover each other's weaknesses. The resulting software will also be easily modifiable to pure Z buffer or pure space-partitioning if required. By allocating "objects" directly to the layers in our parallel scheme we also achieve a neat fit with the architecture.

(6) Work Distribution Algorithms

The distribution of work between processors must be done in two directions in our proposed architecture. Firstly we must consider the allocation of tasks to the horizontal layers of processors associated with object space parallelism. To maintain the advantages of our proposed default hidden surface removal method then all of the data processed by a given layer must belong to a single pre-structured object. This will not normally result in an efficient distribution of work however since objects are likely to be uneven in "size". We would like to be able to use a farm strategy to allocate

work to the layers. If we are to do this it means that the objects must be smaller so that each layer can deal with several of them. Processing several objects within one layer implies that contributions that belong to different objects may need to be kept apart until the end of the rendering cycle when all the object contributions are put together.

Allocation of work within the layers is more straightforward but there are still choices to be made between the advantages of coherence which come from allocating each processor to a contiguous area of screen and the more even work distribution that results from a scatter decomposition. The object farming scheme described above makes a scatter decomposition more attractive since an even workload across the screen will not normally be possible for individual objects even though it may exist for the scene as a whole. This must be balanced against the extra programming complexity which it requires and hence the issue cannot be resolved in general. The output multiplexer must thus support both schemes.

(7) System Performance

We have analysed a number of different rendering schemes in terms of the performance which they require from the system - for a given level of scene complexity. The results presented here are a mixture of figures derived from experience of existing real time graphics systems, figures derived from from experimental coding and estimates. For each rendering scheme figures have been calculated for the per-pixel, per-scanline and per-face workload. The total workload, in terms of processors equivalent to the Texas 320C40, has then been calculated.

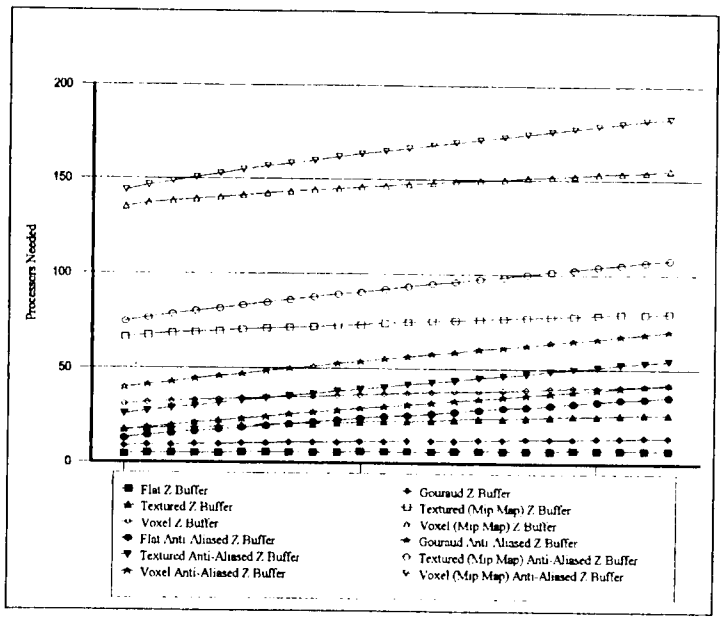


Fig. (2) Processors required as a function of number of faces.

Two hidden surface schemes have been considered, the Z buffer and an antialiased scheme using a Z buffer for inter-object comparisons as described above. Six different shading schemes have been considered: simple flat shading; Gouraud shading; texture, in simple form and with a mip-map antialiasing scheme and finally, as an example of the kind of specialised rendering scheme which can be supported by the present approach, a voxel based method of displaying spotlights with complicated beam patterns [5]. This last scheme is also shown in simple and mip-map form.

Fig. (2) shows processors needed as a function of the number of faces for a 500x500 screen with an average depth complexity of 3. In Fig. (3) the number of faces has been held at 4000 and depth complexity has been varied instead. This is equivalent to varying the resolution. From Fig. (2) it is clear that the processing requirement depends critically on the algorithm but is insensitive to the number of faces. Fig. (3) shows that processor loading is very sensitive to depth complexity or resolution. This is because non-specialised processors are good at geometry but relatively poor at rendering.

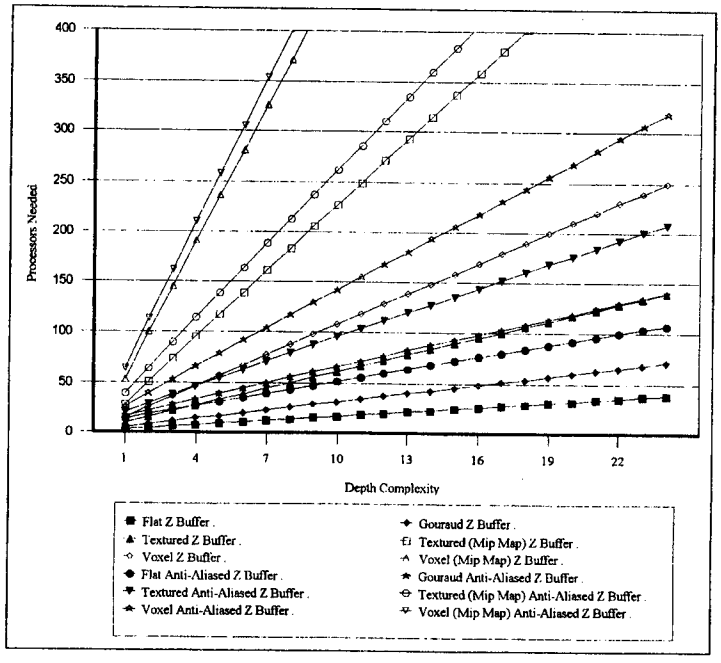


Fig. (3) Processors required as a function of depth complexity.

(8) Conclusions

Clearly a system which incorporates the most sophisticated rendering scheme at a high resolution will be impossibly large with current technology but by confining the more sophisticated techniques to those parts of the image which the application needs, costs can be contained. A 64 processor system could be cost competitive with high end dedicated graphics systems and would allow non-standard rendering techniques to be used in critcal areas. Such a system will not need re-developing as technology progresses and can take advantage of higher processor performance either by improved system capability or by reduced cost.

References

[1] Molnar, S. E., J. Eyles and J. Poulton "PixelFlow: High Speed Rendering Using Image Composition," SIGGRAPH '92, Vol 26, No.2, P23.

[2] Cant, R.J. and P. Sherlock, "CIG system for periscope observer training," Proceedings of the 9th Interservice/ Industry Training Systems Conference (1987) P 311.

[3] Green, S. A. and D. J. Paddon, "Handling Graphical Databases in Parallel Architectures" Proceedings of BCS Seminar "Parallel Processing for Display" (1989) P1.

[4] Sutherland I.E. , R. F. Sproull and R. A. Schumacker " A Characterization of Ten Hidden Surface Algorithms", Tutorial Computer Graphics , (1982) P387.

[5] Burton N.D. "Algorithms for Advanced Light Sourcing in Real Time," The Nottingham Trent University Department of Computing Project Report (1994).

TRANSPUTER NETWORK IMPLEMENTATION OF BOOLEAN OPERATIONS ON POLYHEDRAL OBJECTS

J.CHEPAITE,
Russian Academy of Sciences, Scientific Research Institute for System Studies, Avtozavodskaja 23, Moscow, 109280
e-mail: gruntal@systud.msk.su

The problem of Boolean operations on polyhedral objects implementing using transputer network is considered. The basic sequential algorithm is divided into several stages so that some of them allow parallel processing. The used method of parallelization is data partitioning. The proposed parallel algorithm is proved to keep two-faze protocol and therefore correct. An algorithm effectivity is estimated.
KEYWORDS:
<solid modeling> <boolean operations> <polyhedral> <transputer network> <parallel algorithm>

1.INTRODUCTION.

Any computer implementation of Boolean operations on solids (one of well-known problems of solid modeling) leads to the certain problems. If we choose CSG-representation for solids we should meet only time consuming methods for visualization. If we choose boundary representation (B-rep) we should find the increase of time consumed by Boolean operations performing.

We consider just the last case in this paper. There is a promising way to decrease the time: it's parallel processing. Boolean operations on polyhedral objects require many similar computations. Hence we can partition our data into several parts and process the parts simultaneously. This paper presents the topological method for data partitioning and proposes the scheme for parallel performing Boolean operations on transputer ring.

2. OBJECTS AND OPERATIONS.

Polyhedral solid (solid). A regular subset of R^3 (Euclidean space of three dimensions) which boundary is a union of a finite number of planar polygons will be called polyhedral solid, or simply solid.

These solids are represented by their topological boundary, i.e. by a set of polygons - solid's faces. A set of polygons representing a solid must satisfy five restrictions:

1) Each polygon of the set must be planar and convex.

2) No three vertices of the polygon may be collinear.

3) A polygon may no contain the same vertex twice.

4) The vertices of each polygon must be ordered clockwise when viewed from outside the solid, so that cross-products using the directed edges of the polygon may be used to determine the interior of the object.

5) No polygon may intersect any other polygon of the set, what means: polygons may share a vertex or an edge but they may not have shared internal points.

A vertex is a point of R^3 represented by its spatial location (coordinates). Also the structure of a vertex contains a list of pointers to all polygons which the vertex belongs to. So structures of vertices indicate polygons' connectivity.

Due to [1], conventional Boolean operations are not convenient for computer geometry, e.g. a class of regular sets is not closed under them. That is why we will use so called regularized Boolean operations: the class of regular sets (our solids) is closed under them and results are intuitionally true.

*Definitions. Let S be a subset of R^3, * mean any Boolean operation (union, intersection or difference), i(S) - the interior of S, k(S) - closure of S.*

Let A, B be regular sets. Define regularized union (intersection or difference) of sets A, B A'B = ki(A*B).*

Note that the boundary of the solid received by any Boolean operation on polyhedral solids is embedded into the union of their boundary and polygonal so as initials.

3. THE SEQUENTIAL ALGORITHM.

Now we describe the algorithm base used to create the parallel one. The base algorithm was developed by V.I.Vjukow, it is a modification of algorithm [2] and is explained in details in [3].

We operate on two polyhedral solids at a time. Both are given in boundary representation form described in section 2. The algorithm transforms each representation to a new one. The goal of this transformation is to make that for each solid all polygons from its new representation may be uniquely classificated as lying inside or outside the other solid, or on its boundary with the same outer normals to both solids or with the opposite outer normals: INSIDE | OUTSIDE | SAME | OPPOSITE.

If we obtain such representations for solids and mark each polygon correspondingly to its location, we may easily find boundary representation for the result of any Boolean operation on the two solids as a combination of the two representations. For detailed explanations and proves see [4].

Name two solids solid_A and solid_B, and their boundary polygons pol_A and and pol_B correspondingly. The algorithm for performing Boolean operations on two solids consists of three stages:

1) All segments of intersection of two polygons (pol_A, pol_B) are calculated. Vertices of such segments are marked "boundary". Two non-complanar polygons assume be intersected if a vertex of one lies in the interior of a face or edge of the other, or if an edge of one crosses an edge or face of the other. Polygons that share a vertex or an edge, or that are complanar, do not intersect. Complanar pairs of polygons are not considered because for any group of adjacent complanar polygons in one solid corresponding group in other solid covering the same region exists.

2) All polygons having non-empty list of intersection segments are subdivided into parts so that all segments become edges of new polygons. Note that the received representation is not correct in sense of polygons' convexity. This defect will be corrected at the end of the final stage for polygons belonging to the resulting solid.

We have now required representations for both solids.

3) Marking all polygons accordingly their location and assembling the resulting solid's representation with correction of defective (non_convex or/and inverted) polygons.

4. THE PARALLEL ALGORITHM.

To create parallel algorithm for Boolean operations we need quaranties of the parallel algorithm correctness.

The following scheme of performing Boolean operations on transputer ring is proposed. The ring configuration is one of the most popular configuration of transputer network. We work on two polyhedral solids at a time: solid_A and solid_B.

The polygons that do not intersect with other solid according simple checking boxes before partitioning, remain at place and are not processed. The larger solid (let it be solid_A having more polygons than solid_B) is partitioned into N parts, where N is a number of transputers, solid_B only undergoes checking. Each transputer receives a part of solid_A and a whole solid_B. Note that solid_A partition induce solid_B partition into parts intersecting with corresponding parts of solid_A. Parts of solid_B are intersected in general case. Then all transputers perform calculations of intersection segments for the part of solid_A and solid_B. So, all pairs of polygons (polygon_A, polygon_B) are examined on 1 step with N concurrent processes. We have parallelized the most time consuming stage of algorithm: the first one. The same method may be used for the second stage. The third stage is less time consuming than two others and uses polygons' connectivity essencially, therefore it remains sequential, may be only the correction of defective polygons be parallelized.

The parallel algorithm for processing geometric data is correct, if its result is equal to the result of its sequential version. If a sequence of processing steps performed by several concurrent processes gives the result equal to the result of these steps performing sequentially in some fixed order, the sequence is called serializable.

Let a model of data processing include the following three operations on data elements: lock | change | unlock. It is proved in [5], the theorem 10.2, that if the sequence of data processing steps maintain so called two-faze protocol, it is

serializable. Two-faze protocol means that each process that locks any data, changes it and then unlocks, do all locks before all unlocks.

See our algorithm. The data is partitioned to transputers and processed. Partitioning is done by a special process (root) which stores all data without changes before all transputers finish processing. Processing results will be a list of intersection segments, may be empty, for each polygon. Only the segments must be send to the "root" process, and a complete lists of segments are formed for all polygons and then added to them. Vertices are not changed at all, only new vertices are created and send to "root". Any vertices created on parts' boundaries may be obtained twice and must be joined in such cases.

So each polygon and each vertex, except solid_A parts' boundaries and solid_B induced parts' intersections, is subjected to following steps in process "root":

{lock solid_A}, {lock solid_B}, {send data to other processes},

{receive obtained segments and form complete lists of segments for all polygons},

{change data with boundaries checking},

{unlock of solid_A}, {unlock solid_B}.

In this sense, our algorithm maintains two-faze protocol. "Root" process which performs data partitioning and assembles a result, must check all results on parts' boundaries before data changing, but other data are processed correctly.

5. DATA PARTITIONING AND DISTRIBUTION.

Specific problems with data are connected with parallel processing:

1) we need a method for data partitioning.

2) we need a tool for geometric data distribution;

First, we check all geometric data, i.e two sets of polygons, to reject what of polygons do not intersect other solid and are not processed (for example, a simple checking of boxes intersection gives a result). The rest of data must be partitioned into N parts, where N - a number of transputers.

The simplest is a geometric way: space division into equal

volumes. But it has a number of disadvantages: it is not invariant by continuous space transformations, it gives bad balancing for non-symmetrical solids and for solids with non-homogeneous surface and haves problems with polygons refering to.

We present a topological partitioning method free from above-mentioned. All parts must contain nearly equal numbers of polygons (a number of remained polygons divided to N, one more or less). Each part must be connected so as it is possible. The following algorithm of linear complexity is implemented for data partitioning. The first part is beginning to assemble from an arbitrary polygon. All polygons connected to it are adjoined Then all polygons connected to assembled region are adjoined, and so on, while a necessary number of polygons are adjoined. If any polygon connected to assembled region does not exist an arbitrary polygon is adjoined. The second (and other) part is beginning to assemble from a polygon which would be be adjoined following.

So obtained parts are connected as it is possible and contain nearly equal numbers of polygons. The method gives good load balancing what is important for parallel algorithm effectivity, and (due to parts' connectivity) reasonable length of parts' boundary what is checking after parallel processing.

Second, the properties of geometric objects require to use pointers (or any their equivalent) for computer representation: lists of changeable lengths are natural for geometry. It leads to some difficulties in the data distribution process. The author introduces special structures without pointers exclusively for data exchange and uses data converting procedures. Required time linearly depends on data structures' number, speed of one sort's data structures sending is nearly constant but comparatively slow. For example, for polygon's structures the speed is obtained 232620 bytes/ sec. The obtained time of data transmitting is given in the table No1. The transputer board VMTM from Parsytec GmbH, installed on the workstation BESTA-88 was used.

Table No1.

P pol.	V vert	Volume bytes	Time of transmitting, sec	Speed, bytes/sec
6	8	644	0.004608	139757
32	25	2782	0.018816	147853
128	114	11568	0.061440	188281
512	482	47122	0.223872	210477

6. ESTIMATIONS OF EFFECTIVITY.

We estimate an effectivity of the first and the most time consuming stage of our algorithm. Let N_A *and* N_B *be numbers of polygons of the solid_A and solid_B correspondingly. Then the sequential algorithm consumes the following time:*

$time_{seq} = a*N_A\ N_B + b*N_A$, *there the first term corresponds to polygons boxes checking and the second with the coefficient* $b = b(N_B)$ *corresponds to polygons intersections calculation.*

The parallel algorithm consumes the following time:

$time_{par} = k*time_{seq} + time_{part} + time_{distr} + time_{asmbl}$,
where k must be $1/N_{tr}$ *if* N_{tr} *transputers work with ideal load balancing. Really k is larger because the load is not uniformly distributed, for four transputers k is nearly 1/3. The times for partition, data distribution and result assembly may be estimated as follows:*

$time_{part} = c*(N_A + N_B)$,
$time_{distr} = d*(N_A + N_B)$,
$time_{asmbl} = e(N_B)*N_A$,

The effectivity may be estimated:

$$e = \frac{time_{seq}}{time_{par}*N_{tr}} = \frac{a*N_A\ N_B + b*N_A}{N_{tr}*k*(a*N_A*N_B+b*N_A)+N_{tr}*((c+d)(N_A+N_B)+e*N_A)}$$

$e \to N_{tr}*k$, *when* N_A, N_B *converges to infinity,* N_A/N_B = *constant.*

Experimental results obtained on four transputers are presented in the table No2.

Table No2.

Polygons (polygons processed), vertices	6(5), 8 6(5), 8	32(24), 26 6(6), 8	128(32), 114 18(18), 32	512(128), 482 18(18), 32
$time_{part}$, in sec.	0.006528	0.024384	0,040960	0.141184
$time_{distr}$, in sec.	0.030720	0.048384	0.078528	0.151296
Parallel step, max $time_{par}$, in sec.	0.005376	0.050176	0.057472	0.086400
$time_{asmbl}$, in sec.	0.003200	0.006592	0.010368	0.016000
parallel algorithm $time_{PAR}$, in sec.	0.045824	0.129532	0.187320	0.394874
$time_{SEQ}$, in sec.	0.032832	0.268544	0.227840	0.631040
the speedup: $time_{seq}/time_{par}$	0.0716480	2.073187	1.216314	1.598079

It is seen from the table the speedup differs greatly for different data. At first, the parallel algorithm gives speedup only for large amount of data (see the first column). Data distribution is comparatively slow and it is significant if it is possible to reject a sufficient number of polygons which do not intersect other solid by previous checking boxes when partitioning. This depends on solids and boundary polygons spatial location and influences the speedup.

7. BIBLIOGRAPHY.

[1] Tilove R.B., Requicha A.A.G., Closure of Boolean Operations on Geometric Entities, Computer Aided Design, Vol 12(5), September 1980, 219-220.

[2] Laidlaw D.H, Trumbore W.B. and Highes J.F., Constructive Solid Geometry for Polyhedral Objects, Computer graphics, 1986, Vol 20(4), p.p. 161-168.

[3] Вьюков В.И., Реализация алгоритма конструктивных операций для многогранников, НСК, препринт, 1994.

[4] Tilove R.B., Set Membership Classification: A Unified Approach to Geometric Intersection Problems, IEEE Transactions on Computers, Vol C-29, No 10, Oct 1980, 874-883.

[5] Ульман Дж., Основы систем баз данных, М., Финансы и статистика, 1983.

[6] Чяпайте Ю.Р., Алгоритм разделения множества граней поверхности многогранного тела на примерно равные связные части, НСК, препринт, 1994.

Section IV : Architectures

The Balanced Hypercube and Fault-Tolerant Ring Embeddings

Jie Wu and Ke Huang
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, Florida 33431
jie@cse.fau.edu

Abstract

The *balanced hypercube*, as a variant of the standard hypercube structure for multicomputers, has the desirable properties of strong connectivity, regularity, and symmetry. This structure is a special type of *load balanced graph* designed to tolerate processor failure. In balanced hypercubes, each processor has a spare (matching) processor such that they share the same set of neighboring nodes. Therefore, tasks that run on a faulty processor can be reactivated in the corresponding spare processor to provide an efficient system reconfiguration. This paper studies the embedding of the ring structure in balanced hypercubes. Fault-tolerant capability of the embedding scheme is also discussed. The concept of *consistent* (k, k) *embedding* is used to provide a uniform recoverable embedding which is independent of the number and location of faults to be tolerated.

Key Words: Fault-tolerant Embedding, Hypercubes, Ring Structures, System Reconfiguration

1 Introduction

As one of the most popular parallel structures, hypercubes [6] have received much attention over the past few years. The hypercube structure offers a rich interconnection with a large bandwidth, a short (logarithmic) diameter, and a high degree of fault tolerance. *Balanced hypercube* [3], proposed by the authors, is a variant of the hypercube structure intended to enhance the fault-tolerant properties of the regular hypercube. It has similar desirable properties as those in a standard hypercube. In addition, it has better fault tolerance properties. Actually, a balanced hypercube is a special type of *load balanced graphs* [1], [7]. Suppose a undirected graph is denoted as $G = (V, E)$ with V as the node set and E as the edge set. In a load balanced graph $G = (V, E)$, for every node v there exists another node v', such that v and v' have the same adjacent nodes (see Figure 1). Such a pair of nodes v and v' is called a *matching pair*. In a load balanced graph, a job can be scheduled to both v and v' in such a way that one copy is active and the other one is passive. If node v fails, we can simply shift jobs of v to v' by activating copys of these jobs in v'. All the other jobs running on nodes other than v and v' do not need to be reassigned to keep the adjacency property, i.e., two jobs that are adjacent are still adjacent after a system reconfiguration.

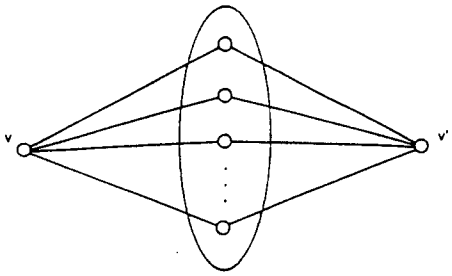


Figure 1: Nodes v and v' that share the adjacent nodes

In general, the ability of embedding a basic interconnection network is one of important measures of a newly proposed network. Of equally importance is the fault-tolerant embedding capability, which can be divided into two approaches [5]: the embeddability of the faulty network (the faulty balanced hypercube in this paper) [2] and the recoverable embedding used to recover from faults as well as maintain certain level of quality of the embedding [4]. We consider the fault-tolerant embedding of the later type. More formally, an embedding ϕ of an application graph G_a into a host graph G_h is an injective function from V_a to V_h. Note that although we can define a more general concept of embedding by replacing the injective function by the regular function in the definition, it is beyond the scope of this paper. The *expansion* of ϕ is $|V_h|/|V_a|$. The *dilation* of ϕ is $max\{d(\phi(v_i), \phi(v_j)\}$, for every $(v_i, v_j) \in E_a$. The expansion is a measure of processor utilization. The dilation represents the maximum communication delay between the communicating nodes. The recoverable embedding studied in this paper consists of two steps: (1) Determine an embedding (called original embedding) ϕ of G_a into G_h. (2) Determine a new embedding (called recovered embedding) ϕ' based on ϕ in the faulty G_h by replacing each faulty component by a close-by spare one.

In general, four parameters can be used to characterize a fault-tolerant embedding [4]: (1) The number of tolerated faults, k. (2) The number of recovery steps, t, which measures the number of steps to replace faulty components by spare ones. (3) The dilation of the original embedding ϕ. (4) The dilation of the recovered embedding ϕ'.

In this paper, we study only those embeddings with unit dilation in both original and recovered embeddings. These recoverable embeddings require that each recovered embedding is isomorphic to the original one. Moreover, we only study *k-fault-tolerant k-step-recoverable embeddings*, or (k, k) embeddings [4]. That is, only k steps are required to recover k faults. We exclude trivial embeddings which have a small G_a and a large G_h. More precisely, suppose both the size of V_a and V_h are function of a variable n, an embedding ϕ is trivial if its expansion approachs zero when $n \rightarrow \infty$. In general, the property of high fault tolerance and small expansion are two contradicting goals. High fault tolerance requires more spare components while small expansion means few spares. Therefore, a reasonable balance is necessary in real applications. In this paper , we concentrate on achieving high fault tolerance in non trivial embeddings.

We propose a concept of *consistent* (k, k) *embedding* which provides a uniform recoverable embedding which is independent of the number of faults to be tolerated. More specifically, a consistent (k, k) embedding requires that each node has a distinct spare node which is independent

Table 1: The spare node set of a 3-dimensional hypercube

node v_a	1	2	3	4	5	6	7	8
imaging node $\phi(v_a)$	0000	0010	0011	0111	1111	1101	1100	1000
spare node $\phi'(v_a)$	1010	0001	0110	1011	0101	1110	1001	0100

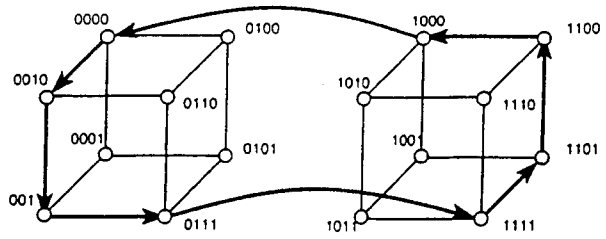
of the number and location of faulty nodes in the system. To illustrate the concept of consistent (k,k) embedding, we consider the following rings embedding in hypercubes. When G_a is a ring of even length and G_h is a hypercube, $(1,1)$ and $(2,2)$ embeddings have been identified [4]. However, in general the recoverable embedding in those schemes are dependent on the actual number of faulty nodes and their locations. For example, Figure 2 (a) shows an embedding ϕ of a ring of 8 nodes in a 4-cube. Figure 2 (b) shows an recovered embedding ϕ' after replacing a single faulty node 0000 by a spare node 1010. The one-step recoverability in shown in Table 1 where each image of v_a has a distinct spare node in G_h. Therefore, the embedding in Figure 2 (a) can provide a $(1,1)$ embedding. However, a simple replacement of faulty nodes by their space nodes does not work in multiple-fault cases, i.e., in a general (k,k) embedding. Suppose nodes 0011 and 0111 are faulty, a simple replacement of their spare nodes 0110 and 1011 does not work. Actually, nodes 0110 and 1110 should be used here to replace faulty nodes 0011 and 0111. In another case, suppose nodes 0011 and 1111 are faulty, then a simple replacement their respective spares (0110 and 0101) works. Therefore, in order to achieve a $(2,2)$ embedding, a table similar to Table 1 is required to differentiate combinations of faulty node pairs and their spare node pairs. Similar reasoning can also be applied to other k, with $k > 2$. Although tables can be condensed in other more efficient formats, different cases have to be treated separately. Actually, to the best of our knowledge, a general (k,k) embedding of a ring into a hypercube has not been identified. In the example of Figure 2, when nodes 0011, 0111, and 1111 are faulty, there does not exist a recoverable embedding which uses three steps. In general, it is highly desirable that we can find a uniform (k,k) embedding which can be applied to all k's. This is captured in the proposed consistent (k,k) embeddings. Note that not all the networks (including hypercubes as will be shown later) can have such an embedding of a certain host graph. We show that there exists such an embedding of ring in the balanced hypercube.

2 The Balanced Hypercube (BH) Structure

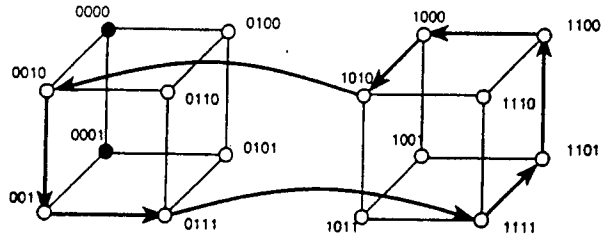
In the following, we define the *balanced hypercube* (BH) structure. For convenience, we assume that all the arithmetic operations on addresses of nodes in a BH are modulo-4 operations.

Definition 1: *An n-dimensional balanced hypercube BH_n consists of 2^{2n} nodes with addresses $(a_0, a_1, \cdots, a_{n-1})$, where a_i $(0 \le i \le n-1)$ is a number from $\{0,1,2,3\}$ and $n \ge 1$ is the dimension. Every node $(a_0, a_1, \cdots, a_{n-1})$ connects the following $2n$ nodes:*

(i) $(a_0 \pm 1, a_1, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_{n-1})$, *and*



(a)



(b)

Figure 2: A ring embedding in a faulty 3-dimensional hypercube. (a) The original embedding (before the occurrence of the faulty node 0000). (b) The recovered embedding (after occurrence of the faulty node 0000)

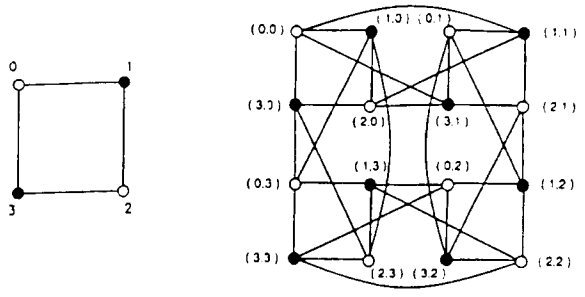


Figure 3: The structures of (a) a BH_1 and (b) a BH_2

(ii) $(a_0 \pm 1, a_1, \cdots, a_{i-1}, a_i + (-1)^{a_0}, a_{i+1} \cdots, a_{n-1})$

where, integer i ranges from 1 to $n-1$, and is called the ith dimension (or outer address) of BH_n.

Figure 3 shows graph representations of BH_1 and BH_2.

In a BH_n, the first element a_0 of node $(a_0, a_1, \cdots, a_{n-1})$ is the *inner address* of an inner cube consisting of a ring of four nodes, and the other elements a_i $(1 \leq i \leq n-1)$ are *outer addresses.* Clearly, nodes within the same inner cube has a different inner address and the same outer addresses. Any node in a BH_n has two types of adjacent nodes: *inner* (Definition 1 (i)) and *outer* (Definition 1 (ii)). Clearly, every node in BH_n has 2 inner adjacent nodes and $2(n-1)$ outer adjacent nodes. For example, in a BH_3 (in Figure 4), node $(1,3,2)$ has an inner address: 1, and two outer addresses: 3 and 2. By adding and subtracting 1 to the inner address of node $(1,3,2)$, two inner adjacent nodes, $(0,3,2)$ and $(2,3,2)$, are derived. Since the inner address of $(1,3,2)$ is odd, the four outer adjacent nodes $(0,2,2)$, $(2,2,2)$, $(0,3,1)$, and $(2,3,1)$ can be determined by decrementing one of outer addresses by 1 and by changing the inner address by 1. A BH_n can also be derived from four BH_{n-1}'s by adding a new dimension as the $(n-1)$th outer address of each node [3]. Figure 3 shows a BH_3 which consists of four BH_2's, where nodes in each of BH_2's are connected as in a complete BH_2. To simplify the notation, connections among nodes from different BH_2's are incomplete in Figure 3, where only nodes $(0,0,0),(1,0,0),(2,0,0)$, and $(3,0,0)$ (which form the shadowed square shown in Figure 2) have complete connections. It is easy to complete all the connections following the connections demonstrated in the shadowed square.

Some basic properties of the balanced hypercube are listed as follows:

Property 1: *A BH_n is a load balanced graph, and its nodes can be partition into a set of matching pairs $v = (a_0, a_1, \cdots, a_{n-1})$ and $v' = (a_0 + 2, a_1, \cdots, a_{n-1})$.*

We use two different colors, black and white, to indicate two different matching pairs in each inner cube.

Property 2: *Every BH_n has 2^{2n} nodes, each of which has $2n$ adjacent nodes.*

Property 3: *A BH_n is a symmetric graph, i.e., for any pair of nodes v and u in BH_n there*

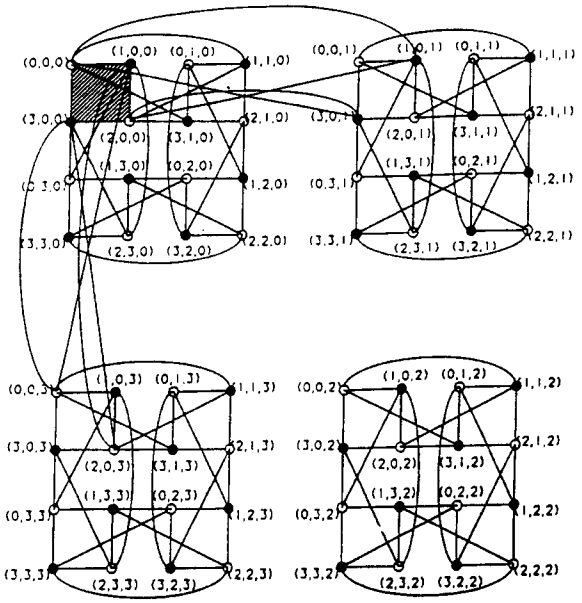


Figure 4: The structure of a BH_3

is an automorphism T *of the* BH_n *such that* $T(v) = u$.

If $v = (a_0, a_1, \cdots, a_{n-1})$ and $u = (b_0, b_1, \cdots, b_{n-1})$, then the automorphism T used here can be represented as:

$$T(w) = (b_0 + (-1)^{a_0}(c_0 - a_0), b_1 + (-1)^{a_0}(c_1 - a_1), \cdots, b_{n-1} + (-1)^{a_0}(c_{n-1} - a_{n-1}))$$

for any node $w = (c_0, c_1, \cdots, c_{n-1})$ in the BH_n.

Property 4: *A* BH_n *is a bipartite graph, i.e., its node set can be divided into two disjoint subsets such that every edge connects two nodes from different subsets.*

In Figures 3 and 4, nodes in white form a subset and nodes in black form another subset.

Property 5: *A* BH_n *is* $2n$*-connected, i.e., for any pair of nodes in the* BH_n *there exist* $2n$ *node disjoint paths between them.*

Property 6: *The diameter of a* BH_n *is* $2n$ *when* n *is even or* $n = 1$*, and is* $2n - 1$ *otherwise.*

Note that a $2n$-dimensional hypercube H_{2n} has a diameter of $2n$. Therefore, A BH_n has a smaller diameter than a compatible H_{2n} when n is odd and $n > 1$.

3 Ring Embedding in HB

To embed rings in balanced hypercubes, we need to find out, for every node v, the next adjacent node $f(v)$ on the ring. The basic idea used here is to find directly spanning rings in a BH_1 and a BH_2, and then use the ring embedding in a BH_2 as the building block to construct the embedding in a BH_3. In general, the spanning ring in a BH_n is constructed by four spanning rings in four BH_{n-1}'s. This is done by breaking each spanning ring and then connecting each broken ring using four links that connects these four BH_{n-1}'s. The following theorem shows one of the possible solutions based on the above idea.

Theorom 1: *A map f_n on the node set of an n-dimensional balanced hypercube, $BH_n = (V_n, E_n)$, is as follows:*

1. $f_n(2, \underbrace{2, \cdots, 2}_{l}, 0, a_{l+2}, a_{l+3}, \cdots, a_{n-1}) = (1, \underbrace{2, \cdots, 2}_{l}, 0, a_{l+2}+1, a_{l+3}, \cdots, a_{n-1})$, *where* $0 \leq l < n-1$, *and*
2. $f_n(0, a_1, a_2, \cdots, a_{n-1}) = (3, a_1+1, a_2, \cdots, a_{n-1})$, *and*
3. $f_n(i, a_1, a_2, \cdots, a_{n-1}) = (i-1, a_1, a_2, \cdots, a_{n-1})$, *if node* $(i, a_1, a_2, \cdots, a_{n-1}) \in V_n$ *does not match 1 or 2 as above.*

then, the set of edges $E'_n = \{(v, f_n(v)); v \in V_n\}$ *forms a Hamiltonian cycle on the* BH_n.

Corollary : *Any rings of size of* 2^{2l}, $1 \leq l \leq n$, *can be embedded in a* BH_n.

Based on the proof of Theorem 1, we can always construct a spanning ring of size 2^{2l} in each subcube BH_l of a BH_n. For example, rings with 4, 16, 64, 256 can be embedded in a BH_4 which consists of $2^8 = 256$ nodes.

4 Consistent (k, k) Embeddings of Rings in BHs

We first define the concept of the consistent (k, k) embedding, then we propose a consistent (k, k) embedding of a ring in a balanced hypercube.

Definition 2: *In a consistent (k, k) embedding, each node has a distinct spare node. If the reconfiguration method is defined as replacing each faulty node by its spare, then the resulting recovered embedding is isomorphic to the original embedding.*

In the above definition, since each faulty node is replaced its spare in one step, a total of k steps are required in a system with k faults, Hence it is a (k, k) embedding. Also, with each node having a distinct spare node, the configuration is independent of the number and location of faults in the system. Therefore, it is consistent. Clearly, conditions associated with the consistent (k, k) embedding are strong. The following results show that there does not exist such an embedding of rings in standard hypercubes.

Theorem 2: *There does not exist a consistent (k, k) embedding of rings in standard hypercubes.*

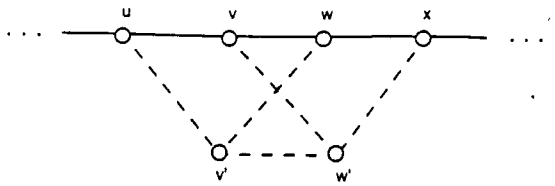


Figure 5: The connection requirement of a consistent (k,k) embedding of a ring

Proof: Suppose nodes u, v, w, x (see Figure 5) in sequence are used in the original embedding of a consistent (k,k) embedding of a ring in a hypercube where v' and w' are spares for v and w, respectively. If the system has one faulty node, v, then the recovered embedding is obtained by replacing v using v'. Since this recovered embedding is isomorphic to the original embedding, v' must be adjacent to both neighbors, u and w, of v. Similarly, w' must be adjacent to both v and x. Now, if the system has two faulty nodes, v and w, based on the definition of consistent (k,k) embedding, v' and w' are used to replace faulty nodes v and w, respectively. Since the recovered embedding is isomorphic to the original embedding, nodes v' and w' are connected (see Figure 7). This implies that there are three length two disjoint paths (via nodes u, w, and w', respectively) between nodes v and v', this contradicts the hypercube property [6] that there are only l node-disjoint paths of length l between two nodes separated by Hamming distance l in a hypercube. □

The usefulness of concept the consistent (k,k) embedding has to do with the reconfiguration method used in practical distributed systems. Since the reconfiguration is consistent, there is no need for each local supervisor to know the global information of the number and location of faulty processors. Therefore, the reconfiguration is much simpler than the one used in a regular recoverable embedding where a certain format of global information collection process is required. The detailed discussion of the usefulness of the consist (k,k) embedding is studied in the next section.

Since each processor used in a consistent (k,k) embedding requires a distinct spare processor, the minimum expansion, which is defined as the ratio of the number of processors in the host graph and the number of processors in the application graph, is two.

In balanced hypercubes, the reconfiguration method associated with a consistent (k,k) embedding can be simplely expressed as follows: whenever a processor with a node address $(a_0, a_1, ..., a_{n-1})$ fails, this processor should be replaced by its matching node, i.e., a processor with a node address $(a_0 + 2, a_1, ..., a_{n-1})$. The spare (matching) node set of a ring of eight processors in the BH_2 of Figure 5(a) is shown in Table 2. A consistent (k,k) embedding of rings in BHs with an expansion two (which corresponds to an embedding with the best possible system utilization) is shown as follows:

Theorem 3: *In an n-dimensional balanced hypercube $BH_n = (V_n, E_n)$, we define a map g_n on the node subset*

$$V_n' = \{(a_0, a_1, \cdots, a_{n-1}) \in V_n; a_0 \in \{0,1\}\}$$

Table 2: The spare node set of a ring of eight processors in a BH_2

node v_a	1	2	3	4	5	6	7	8
imaging node $\phi(v_a)$	(0,0)	(1,0)	(0,3)	(1,3)	(0,2)	(1,2)	(0,1)	(1,1)
spare node $\phi'(v_a)$	(2,0)	(3,0)	(2,3)	(3,3)	(2,2)	(3,2)	(2,1)	(3,1)

as follows:

1. $g_n(0,\underbrace{2,\cdots,2}_{l},0,a_{l+2},a_{l+3},\cdots,a_{n-1}) = (1,\underbrace{2,\cdots,2}_{l},0,a_{l+2}+1,a_{l+3},\cdots,a_{n-1})$, *where* $0 \leq l < n-1$, *and*
2. $g_n(0,a_1,a_2,\cdots,a_{n-1}) = (1,a_1,a_2,\cdots,a_{n-1})$, *if node* $(0,a_1,a_2,\cdots,a_{n-1})$ *does not match 1. above, and*
3. $g_n(1,a_1,a_2,\cdots,a_{n-1}) = (0,a_1-1,a_2,\cdots,a_{n-1})$

then, the set of edges $E_n'' = \{(v,g_n(v)); v \in V_n\}$ *forms a cycle with* $2 * 4^{n-1}$ *nodes in the* BH_n.

This theorem can be proved in a similar way that is used in proving Theorem 1. The following equations are useful to construct such a proof:

$$g_{k+1}(v,i) = \begin{cases} (g_k(v),i) & \text{if } v \neq (0,\underbrace{2,\cdots,2}_{l-2},0) \\ (1,\underbrace{2,\cdots,2}_{l-2},0,i+1) & \text{otherwise,} \end{cases}$$

and

$$\begin{cases} g_2(0,i) = (1,i) \\ g_2(1,i) = (0,i-1) \end{cases}$$

In it clear that this map g_m provides a consistent (k,k) embedding of rings in balanced hypercubes. Figure 6 (a) shows a consistent (k,k) embedding of a ring of eight processors in a BH_2.

Corollary: *For any rings of size* 2^{2l+1}, $1 \leq l < n$, *there exists a consistent* (k,k) *embedding in a* BH_n.

5 Conclusions

The balanced hypercube, a variant of the hypercube structure, supports efficient reconfigurations through fast shifting of a job from a faulty node to its matching node. The concept of consistent (k,k) embedding has been proposed, where the reconfiguration method used is independent of the

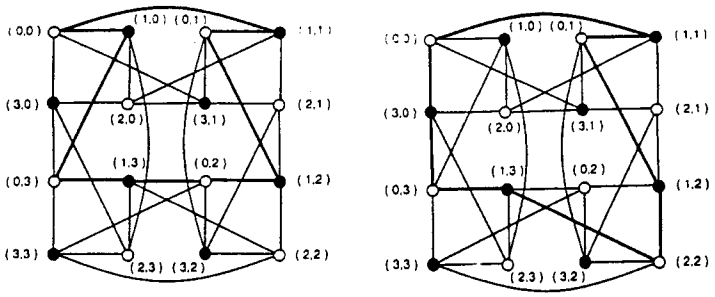


Figure 6: The consistent (k,k) embedding of a ring in a BH_2. (a) The original embedding before the failure of nodes (1,0) and (0,2). (b) The recovered embedding

number and location of the faulty processors. This process not only provides an efficient reconfiguration, but also keeps adjacency relationships among jobs in the previous system configuration. We believe that the proposed method is a practical and useful solution to minimize reconfiguration time in a decentralized control system. The balanced hypercube proved to be an ideal supporting architecture.

References

[1] H. Ali, A. Boals, and N. Sherwani. Generalized load balancing graphs. *Congressus Numeratium.* 79, 1990, 17-25.

[2] Y. C. Chang and K. G. Shin. Load sharing in hypercube multicomputers in the presence of node failures. *Proc. 21st International Symposium on Fault-Tolerant Computing.* 1991, 188-195.

[3] K. Huang and J. Wu. Balanced hypercubes. *Proc. of the 1992 International Conference on Parallel Processing.* Vol 3, Aug. 1992, 80-84.

[4] T.C. Lee. Quick recovery of embedding structures in hypercube computers. *Proc. 5th Distributed Memory Computing Conference.* 1990, 1426-1435.

[5] J. Liu and B. M. McMillin. A divide and conquer ring embedding scheme in hypercubes with efficient recovery ability. *Proc. of the 1992 International Conference on Parallel Processing.* Vol 3, Aug. 1992, 38-45.

[6] Y. Saad and M. H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers.* 37, (7), July 1988, 867-872.

[7] N. Sherwani, A. Boals, and H. Ali. Load balancing graphs. *Congressus Numerantium.* 73, 1990, 205-214.

[8] R. M. Yanney and J. P. Hayes. Distributed recovery in fault-tolerant multiprocessor networks. *IEEE Trans. on Computers.* 35, 1986, 871-879.

On System Interconnection Strategies For A Parallel Machine

F. F. Cai

Department of Computing & Information Systems
London Guildhall University
100 Minories
London EC3N 1JY
United Kingdom

Email: F_Cai@uk.ac.lgu.tvax

Summary

This paper investigates cell interconnection strategies for the Group Processor System, a parallel machine based on a loosely-coupled cellular architecture. A brief overview of the system configuration and its distributed operating system is presented. Several cell interconnection strategies are then discussed and assessed in terms of their capability in maximising system resources utilisation, while minimising communication overheads. A software simulation exercise has been undertaken and some results are included in the paper. It is hoped that the simulation outcome will not only provide useful indications to the effectiveness of the different strategies discussed, but also offer some constructive input to the design and implementation of other parallel processing systems.

Key Words

Parallel processing, system interconnection, communications, operating systems.

1. Introduction

Intense research activities have been witnessed over the last decade or so in the area of designing and implementing parallel computing systems. With remarkable advances in both hardware and software technologies, many parallel machines may now be viewed as an *integrated* software *environment* with a high degree of functional distribution as well as co-operation.

In this paper the issue of cell interconnection strategies is investigated for a parallel machine, named the Group Processor System (GPS), based on cellular architecture. A cellular GPS may essentially be visualised as a homogeneous array of intelligent memory cells, which offers a simple, regular yet extensible structure, while maintaining a high level of availability and flexibility [7,8]. A large GPS can easily be built up of several sub-systems through the use of a reconfigurable interconnection network. A distributed operating system, on the other hand, creates a parallel execution environment by distributing overall tasks among available resources in both single user and multi-user configurations.

It is self-evident that one of the basic requirements for the GPS is to facilitate fast communications between parallel processes [4] residing in cells. The problem is compounded by the fact that enhanced parallelism achievable by employing a large number of co-operating cells may also result in frequent interaction between them, potentially generating heavy traffic and contension on various system buses.

With this in mind, an investigation into various cell interconnection strategies were undertaken, in an attempt to assess their potential capability in maximising the utilisation of system resources while minimising communication overheads. These desirable capabilities are of paramount importance to other truly parallel systems as well [3,5].

A software simulation was conducted to examine the relative behaviour of these strategies within the GPS environment. A set of criteria were used to quantify the evaluation and some results are presented in the paper. The simulation outcome, together with a study on cell allocation policies [2], provides useful indications to the impact of cell organisation and management on the overall system performance, and to the feasibility of the GPS as a multi-user parallel machine.

2. A GPS Based On Loosely-Coupled Cellular Architecture

The realisation of multi-user operations in the GPS is supported by its parallel access hardware structure. This section describes briefly the main features of the GPS based on the loosely-coupled cellular architecture, and its distributed operating system [2].

A cellular GPS may be visualised as a homogeneous array of intelligent memory cells. A complete system is built up of sub-systems of common elements which are cells, modules and bus structure. A *cell*, in its simplest concept, can be viewed as a typical uniprocessor system. It provides the processing ability through its processing unit, with the available local memory accommodating the storage needs. Moreover, the input/output communication capability supported by the structure allows each cell to interact with others. This offers a hardware base for the efficient parallel execution of processes mapped to different cells throughout the system.

A *module* has an array of cells and is coupled with other modules via an interconnection network. Co-operation between modules is achieved by allowing communication through a common *bus structure*. This flexible structure consists of a number of functionally dedicated buses (including global, inter-module, intra-module and I/O buses) which are available for use by any cell within any module. With many of these cells per module, the system has the resources needed for a fully distributed machine. Figure 1 illustrates a multi-user image of the system, where different users are working in their own environment within a *reconfigurable* user boundary.

In order to further appreciate the cellular hardware/software features the concept of an *abstract processor* is introduced. An abstract processor is a group of cells configured so as to execute a given process. These cells are connected by physical, as well as logical, communication paths. In other words, it is the equivalent of a multiprocessor/local memory architecture, with a range of possible interconnection strategies.

The concept of an abstract processor is not too distant from that of a virtual processor. Whereas a virtual processor eventually maps to a physical processor when the process is being executed, the abstract processor may map to one or more cells in the GPS. With many such abstract processors within a system, the Group Processor structure can be seen as a network of interconnecting abstract processors.

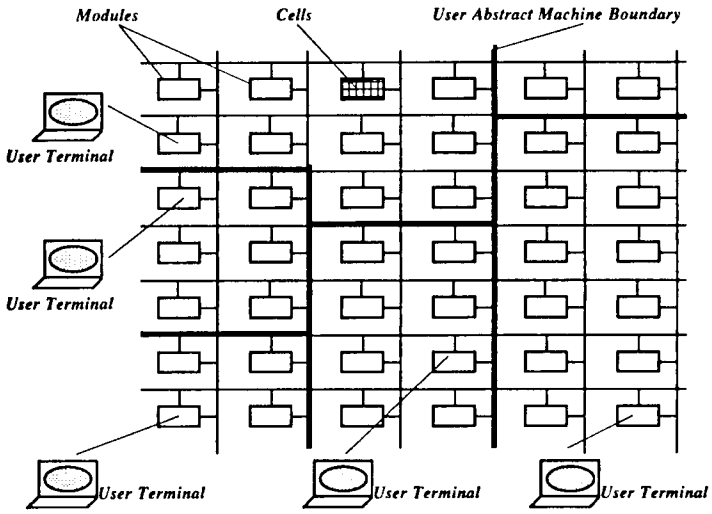


Figure 1. Multi-User Group Processor System

Referring to Figure 1 again for a conceptual relationship between various users and their *abstract machines*. These users are working in their own abstract machine boundaries created by the operating system. Processes are executed by cooperation among those logically-threaded executing abstract processors within the abstract machine. Each user's abstract machine may grow during processing, simply by expanding its logical boundary.

Whilst the hardware structure of the GPS provides the potential base for parallel processing, it is this loosely coupled architecture that imposes inherent complexity on overall system management. Clearly, the control of this complexity is crucial to the operating system design. To this end, a distributed operating system structure was proposed and described in [2], which partitions the overall control into multi-level management by providing a hierarchy of different abstractions. Specifically, the complexity of the system is reduced by dispersing the control from global level down to users's abstract machine level, to the abstract processor level, and also to the cell level. Supported by a flexible cellular architecture and an efficient operating system, a GPS can be seen as an extension in computation power over and above that provided by some conventional architectures.

3. Cell Interconnection Strategies

Apparently, a cellular GPS leads itself to many possible interconnection strategies. They will have a profound impact on overall system performance, particularly on cell management and system communication. Whereas there has been a wealth of research papers on the topic of multiprocessor interconnections in general, this section discusses requirements for cell interconnections within the GPS environment.

3.1 Regularity and Extensibility

The perennial problem in a parallel machine like GPS is how to partition and distribute the overall system workload among available cells in which concurrent processes are being executed in parallel. In a multi-user environment where several highly interactive programs require access to shared system buses, the frequent interaction between cells is likely to cause serious bus contention, data transfer delays and, consequently, a degradation in system performance. Common to parallel systems the implementation of an effective message routing, dynamically generating logical communication paths, is often a complex yet unavoidable issue [1]. It is clearly desirable that the underlying physical interconnection patterns are kept simple and regular so that logical communication paths may easily be constructed when required.

Extensibility is another major consideration when contemplating interconnection strategies for the GPS. As mentioned earlier, in a GPS there are no fixed physical user boundaries, only conceptual ones. A user in need of more cells, instead of waiting for a cell within its own domain to be released, may request extra cells through the global operating system. Each user's abstract machine may grow into another user's free workspace simply by reconfiguring its boundary. In order for the system to strive for this "optimal" cell utilisation [2], cells in the network are required to be readily accessible from others. Moreover, the cell network should be configured in such a way that it can be extended should the need arise.

3.2 Realiability Enhancement

Many computing applications require a high degree of realiability. This realiability may be supported by some forms of software and/or hardware redundancy so that a non-catastrophic fault does not force a complete shut-down and the system continues its operation with a reduced capacity. This so called "graceful degradation" is dependent to a great extent on the system interconnection network.

The realiability of the cell network in the GPS can be enhanced if each cell is reachable from any other cells through more than one possible route, thus offsetting the effect of potentail cell or link failures. In cases where any link or cell fails an alternative route can be taken. This implies that there exist at least two direct or indirect paths between any pair of cells. Ideally, each cell should be directly coupled to every other cells but this is highly impractical for a system with a large number of cells. What is required is an interconnection strategy which makes the system configuration both fault-tolerant and cost-effective.

3.3 Communication Efficiency

Each cell in the GPS has a number of links which connect it to its neighbouring cells. Whereas a message routing is concerned with deciding a path along which information travels between two interacting cells, the term *communication path length* is defined here as the total number of links that have to be passed before a message from the "source" cell reaches its "destination" cell. Therefore, a path length is closely associated with message transfer delays.

Fast communication between cells is essential in the GPS environment. The improvement of the communication efficiency may be brought about in several ways. Firstly, an effective algorithms for cell management is critical to minimise unnecessary inter-cell (especially inter-module) communications[2]. Secondly, message transfer delays may be reduced by implementing a flexible message routing mechanism. Thirdly, an efficient interconnection strategy can again play an important role in increasing the communication bandwidth of the system.

4. Interconnection Patterns

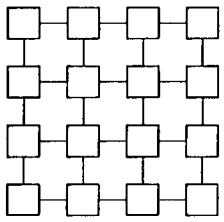
It is obvious that, individually, the requirements identified above are by no means unique to the GPS design. However, when they are considered together a careful examination of possible candidates for cell interconnection patterns is indeed necessary. This section discusses four such patterns based on the array structure. They are two dimensional array, loosely coupled array, tightly coupled array and doubly-twisted torus.

4.1 Two Dimensional Array

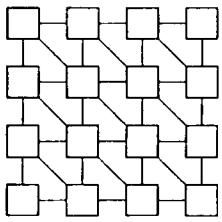
Cells can be mapped onto a two dimensional array, as illustrated in Figure 2(a), in which the number of links per cell is four. In this open-ended network each cell except those along the boundary has identical relationships with its neighbours. Direct links are provided for each cell to access its "left", "right", "upper" or "lower" cells. For an array with n rows and m columns, the position of a cell C can be represented by $C(i_c,j_c)$ where i_c indicates the row number and j_c the column number of the cell in the array. The minimum path length, PL, between any pair of cells A and B is therefore defined as

$$PL(A, B) = \mathbf{abs}\ (i_a - i_b) + \mathbf{abs}\ (j_a - j_b) \quad \text{where } i_a, i_b: 1...n \text{ and } j_a, j_b: 1...m$$

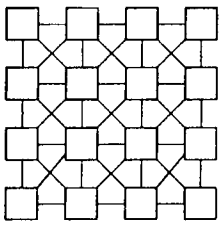
This organisation offers a simple and regular structure, with an addded advantage of easy extensibility. Furthermore, there exist multiple communication routes between cells so that failures of some links/cells may not cause a total breakdown of the network.



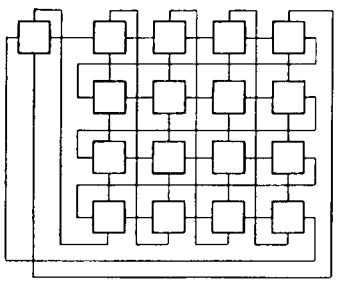
(a) Two Dimensional Array



(b) Loosely Coupled Array



(c) Tightly Coupled Array



(d) Doubly Twisted Torus

Figure 2. Variations of Array Structure

4.2 Loosely Coupled Array

In this structure two additional links are introduced to each cell so that the number of its direct neighbours reaches six, as depicted in Figure 2(b).

Due to the increase of the number of links per cell, the communication path length may be reduced. As long as a cell interacts with another cell whose position is along its upper-left or lower-right direction, it is possible for a message to travel through at least one oblique link during the transmission, instead of taking an otherwise vertical-horizontal-only path. This reduces the path length between these two cells by at least one link. With the potential reduction in the minimum path length between cells, the average path length in the network is likely to be shortened as a result.

4.3 Tightly Coupled Array

If another two links are added as shown in Figure 2(c), each cell in the array except those along the border lines, has direct access to its neighbours in all eight possible directions. Having all the other features of the previous two patterns this configuration clearly offers a higher degree of fault tolerance.

Moreover, the structure may facilitate a further speedup in communications among interacting cells. Messages between any pair of cells whose positions are not on the same row or colume of the array may pass at least one oblique link. If the shortest route are taken in message transfer, the minimum path length between cells is now reduced to

$$PL(A, B) = \mathit{max}\ [\mathit{abs}\ (i_a - i_b), \mathit{abs}\ (j_a - j_b)] \qquad \text{where } i_a, i_b: 1...n \text{ and } j_a, j_b: 1...m$$

4.4 Doubly Twisted Torus

Obviously, the reduction of communication path length in both loosely coupled and tightly coupled array is achieved at the cost of additional links coupled to each cell. This adds to the complexity of the message routing and bus scheduling of the system. However, it is not difficult to observe that any reduction in the minimum path length is ultimately constrained by the length of the route between any pair of cells sitting at the opposite corners of the array. In such cases messages between them have to pass many intermediate cells. This may not only delay the transfer of the messages themselves, but also prevent many other interacting cells from using the same links simultaneously, with a consequence of slowing down communication traffic in the system.

Figure 2(d) depicts an alternative structure known as the doubly twisted torus [9]. In conjunction with the twisted links between cells, the opposite pairs of edges in the array are also connected to form a topologic equivalent of a torus. In this structure cells at the edges can interact much more conveniently than in other array-like configurations, by using the available twisted links. Although the number of links per cell is still four, this end-round structure is expected to have a better communication performance than that of the simple array in Figure 2(a).

An interesting point to note is that by introducing an additional node in this pattern, a network configuration withour boundaries is presented [9]. This feature is particularly attractive when considering applications which require that their computational tasks be evenly distributed among available resources. Sequin demonstrated the effectiveness of this scheme by mapping a binary tree onto it, resulting in a relatively uniform distribution over the network.

5. Evaluation Study

A software simulation exercise was carried out to observe the potential performance factors of the interconnection strategies discussed above . This section presents some of the results from this study.

The evaluation was conducted in two ways. Firstly, the simulation program was run for a given interconnection structure while varying the total number of cells in the system from 64 to 1024. Secondly, it was run for a fixed number of cells but varying the interconnection structures. Both single user and multi-user environment were simulated. In order to establish average values for various parameters evaluated a large number of simulation runs were executed with different random seeds.

5.1 Simulation Environment

The basic architecture of the GPS simulated in this exercise consists of three main components as described in Section 2. They are cells, modules and various system buses. The system configuration is flexible in the simulation, allowing values to be changed for the following parameters:

1) The number of mudules in the system;
2) The number of cells per module;
3) The number of global buses;
4) The number of inter-module buses;
5) The number of intra-module buses.

The simulator also includes a set of software parameters and some assumptions were made on timings for various operations. The execution of the program is measured by time units of equal length. The length of time consumed by communication between two cells is the function of the message length and transfer delays caused by bus conflicts. This delay depends on the availability of the system buses needed for the transmission and on the communication path length between the two interacting entities. Message routing algorithms were developed so that communication takes the shortest path whenever possible.

Due to a very limited paper size it is not feasible to present the whole range of outcome obtained from the evaluation. Consequently, this section includes only some of the results on the system measurement tested, such as the bus service rate, the number of cells in waiting and the communication path length.

5.2 Some Results

In order to examine the difference between various interconnection structures a heavy communication load on system buses was generated. For each of the given size of the network, such as 64, 128, 256, 512 and 1024 cells, different structures were simulated and evaluated.

Figure 3 describes the outcome of the simulation runs for the network size of 1k cells configured as 16 modules with 64 cells per module. Numbers listed under the heading of "cells in waiting" represents the average number of cells waiting to gain access to various system buses. The average bus service rate (in percentage) for a given structure is calculated by the formula:

Service Rate (%) = communication requests made / communication requests served

Interconnection Structures	No of Links Per Cell	Average Bus Service Rate	Average Path Length	No of Cells in Waiting
Two dimensional	4	89.3	6.0	25
Loosely coupled	6	92.2	5.1	20
Tightly coupled	8	97.0	4.2	14
Doubly twisted	4	93.5	4.9	20

Figure 3. Different Structures (16 Module, 64 Cells/Module)

It can be seen from the table that different structures produced different service rates. The two dimensional array, as expected, relates to the lowest rate among all entries. This is mainly due to the fact that less links are coupled to each cell in this structure than in others (except in the twisted torus). With the increase in the number of links per cell in other structures, their service rates have improved accordingly. The tightly coupled array shows the highest service rate as more links are used here than in other structures.

Observations can be made from the first three entries in the table where the number of links are 4, 6, and 8 respectively. The increase of the number by 2 seems to have brought about an improvement on bus service rate by roughly 3 to 4 percent, and a decrease in the average number of cells waiting for bus services by 5 to 6. This was found to be consistent propotionally with most of the results obtained from simulations with other network sizes. Although the generalisation of the correspondence is not attempted here it does give an indication to the co-relation between these factors monitored in this study.

An interesting exception to this ratio is shown by the fourth entry in the table. In spite of having the same number of links per cell as in the two dimensional array, the twisted torus achieved 4 percent improvement on the service rate by utilising the alternative configuration. This result is even a little better than that of loosely coupled arrays where two more links are used. It is clear, therefore, that the number of links per cell in an interconnection structure is only one of the factors affecting performance. The twisted torus structure presents an example of improving bus service rate of the system by employing different cell coupling patterns, with no extra links being added.

Figure 4 gives a set of curves whose data was colated from a number of simulation runs. Each curve corresponds to a particular structure, reflecting the relationship between the average communication path length and the size of the cell network. As shown in the figure, when the number of cells in the system increases every curve is extended in an upward trend. This is self-evident because as the network grows bigger the average distance between any pair of cells becomes longer, and communications among them generally take longer time to complete.

The differences in path length between various structures are also becoming more apparent with the increase of the network size. For configurations of 64 and 124 cells the total difference between the longest and shortest average path length is about 1 unit in the figure. This is nearly doubled when the network grows to the size of 1k cells. This indicates that for a GPS with a large number of cells the interconnection structures indeed have a significant impact on the system communication performance.

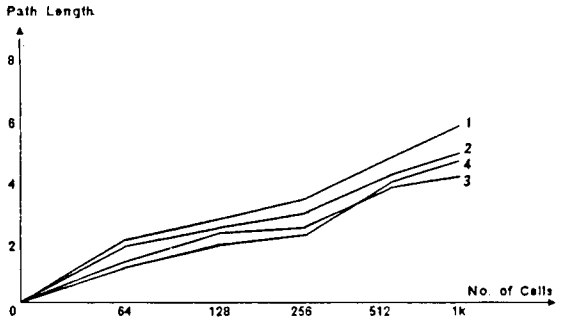


Figure 4. Curves of Average Path Length

In comparing the interconnection structures it is somewhat difficult to identify the best candidate satisfying all of the system requirements discussed in section 3. Generally speaking, the more links coupled to each cell the shorter avarage communication path length and the better realiability. These are only achievable, however, with the aid of a more complex bus scheduling and message routing algorithm. Therefore, a sensible compromise have to be made when considering the requirements of a particular system against its hardware cost and software complexcity. In this respect the doubly twisted torus appears to offer a more balanced choice compared with other structures.

6. Summary

This paper has investigated the interconnection strategies for the Group Processor System based on the cellular architecture. An important property of the GPS is that it offers a simple, regular and extensible structure whilst maintaining the necessary availability and reliability. On the other hand, the efficient management of cells, being one of the most important resources in the GPS, presents a serious challege to operating system designers. A distributed operating system model [2] based on a hierarchical structure was briefly described in the paper along with the main features of the GPS architecture.

If the performance of cell management is to be enhanced the issue of the system communication capability can not be overlooked. To this end, various system interconnection strategies for the GPS have been discussed. In a simulation study, possible candidates based on the array structure have been evaluated with respect to their potential communication performance. Although only a very limited set of results has been included here due to the paper size, the evaluation exercise has nevertheless provided some useful indications to the relationships between various system performance factors, and to the relative merits of different interconnection structures for the Group Processor System.

Over the last decade or so there has been an enormous research effort in the area of parallel processing. Many new parallel architectures have been proposed, associated software developed and systems built. However, the ever increasing types of new applications have demanded even

more efficient parallel systems, the design of which has attracted some concerted efforts by researchers from various computing disciplines (e.g., the integration of parallelism and artificial intelligence). It is safe to predict that this effort will continue unabated in the next decade. Whereas different parallel machines may present very diversified features, common to them all are two basic requirements; efficient resource management and satisfactory communication performance. It is hoped that this study will make a positive contribution.

References

1 Agrawal,D.P. & Leu,J. "Dynamic Accessibility Testing and Path Length Optimization of Multistage Interconnection Networks", *IEEE Trans on Computers*, Vol c-34, No 3 (1985).

2 Cai,F.F. & Hull,M.E.C. "Operating System Support for a Parallel Cellular System", *Applications of Supercomputers in Engineering II*, Elsevier Science Publishing (1991).

3 DeWitt,D. & Gray, J. "Parallel Database Systems: The Future of High Performance Database Systems", *Comm ACM*, Vol 35, No 6 (1992).

4 Dubois,M & Briggs,F.A. "Efficient Interprocessor Communication for MIMD Multiprocessor System", *ACM Conf. Proc. 8th Annual Symposim on Computer Architecture* (1981).

5 Inmos Limited, "Transputer Development System", Prenticel-Hall International (1988).

6 Joseph,M., Prased,V.R. & Natarajan,N. " A Multiprocessor Operating System", Prentice-Hall International (1984).

7 Quick,G.E. "Intelligent Memory:- A Parallel Processing Concept", *ACM SIGARCH*, Vol7, No 8 (1979).

8 Quick,G.E. "Intelligent Cellular System:- A New Direction for Total System Design", *NATO-ASI on Relational Database Machine Architecture* (1985).

9 Sequin,C.H."Doubly Twisted Torus Networks for VLSI Processor Arrays", *ACM Conf. Proc.8th Annual Symposium on Computer Architecture* (1981).

On FPGAs as a new hardware support for parallel discrete event simulation

C. Beaumont, J. Champeau, J.-M. Filloque and B. Pottier*
LIBr, Université de Bretagne Occidentale
BP 809, 29285 Brest, France
e-mail: <name>@univ-brest.fr

P. Boronat†
Universidad Jaume I,
Apart. Correos 224, 12080 Castellón de la Plana, Spain
e-mail: boronat@inf.uji.es

Abstract

Parallel simulation is a good domain for the study of the adequation between algorithms and architectures. The ArMen machine is a general purpose MIMD machine including FPGAs which can be configured as parallel coprocessors. This paper describes parallel simulation sub-domains and studies their respective main architectural requirements. Specific support circuits can be integrated as required in our machine due to its programmability. The FPGA technology and the ArMen machine are outlined. Two examples are given at the end: a fine-grained time-driven simulation and a synchronous event-driven one.

Introduction

The discrete event simulation of dynamical systems requires more and more computational power. Numerical simulation is one of the big challenges for computer scientists and high performance parallel architectures appear to be a natural support simulators for discrete event systems. Several methods have been studied to exploit parallelism. The more important ones are:

i) Partition of the simulated model between several processors. These processors must be well suited to the application granularity[1].

ii) Efficient algorithms for processors and processes synchronization.

iii) Distribution of functions among processors.

*ArMen project is supported by the French PRC-ANM and C^3 (Ministry of research and CNRS), région Bretagne, Brest municipality and ANVAR.

†The work of P. Boronat was done in collaboration with Telecom Bretagne and was supported by BANCAIXA (Spain) under grant A - 38 - IN.

In [2] and [3] first solutions have been proposed addressing the first and second points. Later on, much work has been carried out to propose efficient solutions for distributed simulation on MIMD machines. Fujimoto gives a survey of the best-known approaches[4]. The function distribution among different processors can be considered as a supplementary way. The activity analysis during discrete event simulation shows that a large part of compution is consumed by queue management, random number generation or input/output. These activities can be supported by specific processors or standard ones. In this paper, we focus on model distribution and synchronization problems. Architectural requirements are established for different kinds of parallel simulation. It is clear that each of them could get benefits from executing on a specific machine. The aim of our work is to show that a general purpose parallel architecture including runtime-configurable hardware, built with FPGA-circuits, can be a good solution. Section 1 analyses the parallel simulation field, by distinguishing different characteristics and giving their main requirements. Section 2 describes the hardware support, the ArMen architecture, and its main computing capabilities. Section 3 gives two examples of machine configurations for fine-grained time-driven simulation and synchronous coarse-grained event-driven simulation. It also presents experimental results obtained on an eight-node machine prototype. The last section it devoted to summarize and to indicate further work in progress.

1 Architectural requirements for parallel simulation

The parallel simulation domain can be analized using execution model, granularity and synchronization techniques as criteria[5]. In *time-driven simulation*, the granularity is more important whereas it is the synchronization method in *event-driven* one. The choice of an execution model depends greatly on the real system to be simulated. The architectural requirements for each solution are quite different. They are summarized below. Figure 1 shows our four main categories.

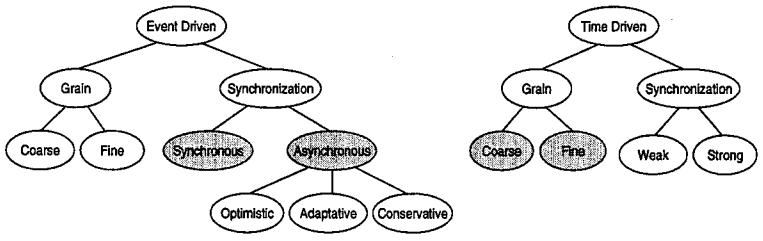


Figure 1: A classification for parallel simulation domain

The first one is **fine-grained time-driven PDES**. If the simulated system can be decomposed into a set of small identical processes with only neighbourhood interactions, then the model can be described as a state-matrix and a transition function. This function enables evaluation of all matrix points at step $p+1$ from the state of their neighbourhood at step p. This model is close to the *cellular automata model*[6]. The requirements are: fine-grained processing unit, local communication and synchronization between these units and large memory to store

the state-matrix. [7] shows that this kind of requirements is implemented efficiently with a linear array of small computing units.

Coarse-grained time-driven simulation is necessary when processes are complex and have many interactions, i.e., there are many events in the system at each step. It requires larger processing units, as example microprocessors, communication capabilities and a global synchronization barrier at each step.

If events occur at irregular time intervals, it is more efficient to use **event-driven simulation**. The **synchronous** approach of this kind of simulation is close to the coarse-grained time-driven PDES, but now the simulators need to assess the minimum of all waiting event timestamps. This evaluation requires global computation over the whole machine, together with a synchronization barrier. Look ahead computing, as defined by Chandy and Misra in [2], can rise performances, allowing more processors to compute events without causal dependencies.

The last category is **asynchronous event-driven PDES**. Asynchronous event-driven simulations use different synchronization protocols. Each process uses a local virtual clock and tries to go faster by only taking care of the processes with which it has interactions. A conservative protocol[2] ensures that no message with a lower timestamp than the current local time can arrive. An optimistic protocol[8] doesn't take care of the time and goes as fast as possible only with local information. When an older message arrives, the protocol rolls back the process to this old time. The adaptive protocols use feed-back information (number of rollbacks for example) to limit or increase the aggressiveness of the protocol. In any case, global evaluations must be achieved[9]. The main one is computation of the minimum over distributed virtual clocks. Conservative protocols require exact evaluation of this minimum but optimistic ones need only approximation of the lower bound.

In conclusion, a *good* parallel machine able to execute efficiently several simulation models should offer the following possibilities:

- Fine-grained operative units with local communication.
- Coarse-grained units with global communication, synchronization barrier and efficient parallel reduction operators.

We propose to use a machine with several levels and grains of computation and communication. The latter uses FPGA technology included into a classical MIMD architecture.

2 ArMen, a multi-layered architecture with FPGAs

The proposed architecture, called ArMen, is a modular distributed memory architecture where processors are tightly connected to an FPGA ring (see figure 2). This section gives information on FPGA technology and shows the main characteristics of the proposed architecture taking into account the simulation requirements.

2.1 FPGA technology

Field Programmable Gate Arrays can be seen as an evolution of the Programmable Logic Devices (PLD). The main drawbacks of the first generation of PLDs are their limited amount of inputs and outputs, on chip logic and registers, and, above all, their out-board programming

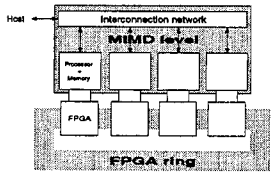


Figure 2: Principle scheme of the ArMen machine

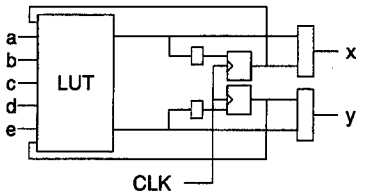


Figure 3: Logical structure of a 3000 serie CLB

with specifc tools. In 1986, Xilinx Inc. has proposed the first static RAM-based field programmable gate array which solves some of these problems, offering a large amount of logic resources and a high number of input/output capabilities. The static RAM memorizes the configuration of the circuit: look-up tables for combinatorial logic and routing point states for the internal connections. Configuration can be loaded dynamically on board and does not require specific programming tools.

Conceptually, an FPGA is a superposition of two planes. The upper one is the configuration memory while the lower one is a programmable ASIC. This one offers an array of Configurable Logic Blocks (CLB, figure 3) surrounded by Input/Output Blocks (IOB). These blocks are interconnected by resources like buses, crossbars or programmable points.

Each CLB can produce one or two combinatorial logic functions by look-up table (LUT) the content of which is determined by bits of the SRAM. The combinatorial outputs may feed the two registers or be used directly (Figure 3).

Current circuits from Xilinx[10] offer up to 1024 CLBs, 256 IOBs and are equivalent to 25.000 gates (Xc4025). Registers can support a clock rate of 270 MHz. The configuration bit streams are built by specific software tools and stored in a local EPROM or downloaded by a host processor. FPGA technology is promising and follows the progress of VLSI integration. Circuits with up to 40.000 gates are announced for '95, while 100.000 gates are promized at the end of the century.

Several kinds of circuits can be designed using FPGA: combinatorial functions, synchronous or asynchronous state machines, ALU, counters, shift registers of different size,... The main limitations of FPGA compared to ASIC come from:

1. limited logic ressources (number of gates eq.)
2. more limited routing resources
3. delays introduced by routing interconnections which lead to unpredictable system clock rates.

Nevertheless, efficient designs have been built. The ArMen architecture, as many other on-going projects, uses these circuits to bring additional possibilities to standard processor systems.

2.2 ArMen architecture

A general purpose MIMD parallel machine including FPGAs has been proposed in [11]. This machine can be specialized at runtime and we outline in the next section some useful operators

in parallel simulation. These operators are designed using high level software tools developed in the project[7, 12] together with specific vendor tools.

Figure 2 shows the architecture of the machine. Each ArMen node has a processor connected via its system bus to a bank of memory and an FPGA. The processor can load dynamically new configuration bit streams into its FPGA. The input/output pins of each circuit are divided into four ports. One (the North port) is connected to the processor bus and the three others are free. Connecting several nodes by the East and West ports in a ring topology enables to build a *logic layer* where global shared coprocessors can be implemented.

This architecture provides [13]:

- **Processor/FPGA interactions**

  Communications. The processor writes values in its FPGA which performs then hard-wired functions. Then the processor reads back the results. The experimentation shows that the throughput is limited by the processor read/write speed. Data-type can be either Boolean or integer.

  Synchronization by interrupt channel. The FPGA accesses interrupt signals of the processor.

  Synchronization by wait-state generation. The processor speed is controlled by the FPGA activity.

- **FPGA/FPGA communications**

  The large inter-FPGA data path is used with either synchronous or asynchronous exchanges. In the first case, the same clock signal is used for two adjacent nodes. In the second one, a *ready-ack* protocol is implemented.

Shared coprocessors are built using these capabilities together with the application specific functions loaded into each node. These configurations may be different for each node.

2.3 Global coprocessors for simulation

Global coprocessors can combine operative and control functions. Operative functions combine the actions of the processor-array with FPGA synchronous processing and fine-grained communications. These functions can be either combinatorial or sequential and are executed in one read/write cycle of the processor. It pushes memory-data into the FPGA interface during the write cycle and then reads back results. This kind of operative unit implements quickly a cellular automata model[7].

Control functions are usually based on a pipeline execution model and can be synchronization barriers, mutual exclusion protocols, predicate evaluations...

A pipelined controller, with one stage within each node, can collect and diffuse data encapsulated into tokens. Sequential or combinatorial operations are performed within each stage.A centralised or distributed finite state machine has charge of the coordination of pipelines [14]. The control task is performed in real parallelism with the application and consumes almost zero processor time.

Section 1 has shown different kinds of operation that can be implemented in a global coprocessor. Let us outline some of them.

An operator for fine-grained model evaluation The result of the operator evaluation at step p depends on values of step $p-1$, stored in a built-in shift register. A compiler generates the configuration of the FPGA automatically. The design entry uses a description in CCEL formalism[7]. The number of operators implemented in each node depends on the data-size and the complexity of the function to be applied. This kind of operator is used in section 3.1 for combinatorial evaluations and in section 3.2 for bit-serial computations.

A synchronization barrier A simple synchronization barrier is implemented by using two pipelines (figure 4). A processor, reaching the barrier, sets to TRUE its *ready* flag. The first pipeline performes a global AND with all the ready flags. The second one diffuses a synchronization signal when all the processors have reached the barrier.

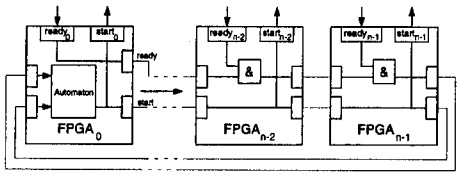


Figure 4: A simple synchronizer

A termination detection circuit If the *ready* conditions are not stable (i.e., a ready node can return to a non-ready state if it receives a message (an event) from an other non-ready node until all the nodes become ready), a simple synchronizer can detect false conditions. This kind of situation is known under the generic name of the termination detection problem[15]. A solution is proposed in [14] and uses four pipelines (figure 5). The first one collects a partial

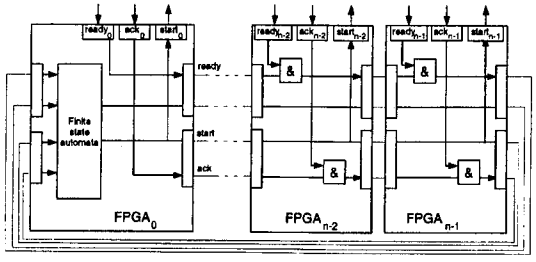


Figure 5: Four pipelines for the termination detection problem

result of an AND-operator applied to the *ready* flag whereas the second one tries to confirm this partial result. We have proven that when a confirmation occurs, a global *ready*-state is detected. Then the third pipeline diffuses an interrupt wave, which enables nodes to go ahead. The fourth pipeline synchronizes nodes before the next detection.

This kind of circuit can be used to compute an approximation of a global minimum for distributed virtual clocks for example, using a windowing mechanism, as stated in an earlier paper[14].

3 Examples and performances

In order to illustrate the benefits of the FPGAs, this section presents two implementations already realised. They are oriented towards PDES and differ in the way FPGAs are concerned. The results presented have been obtained with an eight-node machine.

3.1 Time-driven simulation of a gas model

Although the best known application for the cellular automata model is the game of life, this model can specify the evolution of physical systems like a flow of granular material[16], and fluid or gas dynamics[6]. Simulating such models implies the evaluation of all cells at each time step. This leads typically to a *time-driven* approach for the simulation.

The gas model of Margolus[17] has been implemented on the ArMen machine. Four bits per cell are needed[7]. With on the shelf processor, such tiny data lead to an inefficient use of the computation resources. Furthermore, the simulation of the whole data space on a parallel machine requires a large amount of communications between the nodes. These are implied by the edges problem resulting from the data distribution among the processors.

FPGAs offer good solutions to these problems:

1. the reconfigurability allows the synthesis of well dimensioned ALUs for the application. In the case of small ALUs, the unit is merely replicated in the component, with regard to the hardware resources. For this application, the global operative unit is implemented with a 16-bit word per node pushed in the FPGA by the processor. Hence, for one node, 4 cells are updated during a processor read/write cycle. This limitation is due to the current capacity of the FPGA.

2. the ring topology of the FPGAs enables data exchanges on the logic layer level (i.e., without overhead on the primary network). The problem of edges between slices is solved with a special configuration for the so-called *margin node*.

After a transformation towards the classical expression of a cellular automaton (figure 6 shows some particle motion laws), the specification of the model is compiled with the CCEL tools. These development and synthesis tools provide a specific FPGA configuration (for both

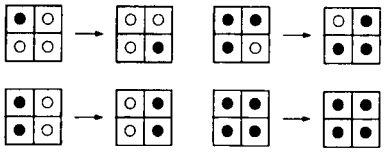
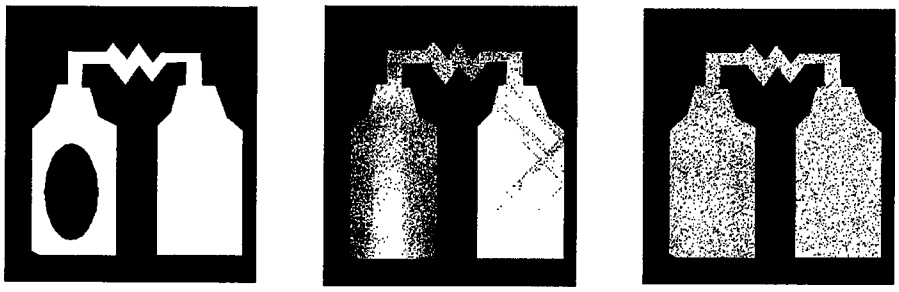


Figure 6: Some possible motions for 1 to 4 particles

compute and margin nodes) where a shift register contains the cell states and an array of combinatorial functions computes the transition.



(a) Starting state (b) After 500 iterations (c) After 2.500 iterations

Figure 7: Graphical displays of the gas model simulation

Figure 7 shows snapshots of the model application: at the beginning, the left bottle contains a set of particles. The evolution of the particles converges to a well balanced number in each bottle. The data space of this model consists of 350×370 cells. The performance for one simulated time-step of the model is about 60 ms, using an ArMen machine with 8 nodes.

3.2 Event-driven simulation kernel

The second example puts forward an implementation of global hardware operator able to compute and disseminate the minimum of distributed values. This operator has been used in an event-driven simulation kernel described in an earlier paper[18].

Now it is well-known in the *event-driven* simulation field that the knowledge of the global minimum of all logical clocks is important (either the exact value, or an approximation). This computation implies overhead on the primary network of the a conventional MIMD machine. Rather than using a control network, as the CM5 does, we take advantage of the reconfigurable technology for this objective.

The ring topology of the logic layer makes possible data exchanges between two neighbours. That is, it offers the possibility to execute, without any communication on the primary network, the operation on three values (the local one, and the ones from the two adjacent nodes). In the case FPGA resources are large enough, then we are able to replicate this operator to form a vertical pipeline. The second stage then computes the minimum on the three following values:

- the previously locally computed minimum,
- the previously computed minima from the adjacent nodes.

Thus, the minimum on 5 values is computed after 2 stages. By extension, after n stages, the minimum on $2n + 1$ values is computed. Figure 8 shows an example with 4 nodes.

One point worth being mentioned is that the global value is already broadcasted when the computation is over.

From the MIMD processors pointof view, the only work to do is to write its local value in a digit-serial manner in its associated FPGA. After each write, it reads back a resulting digit,

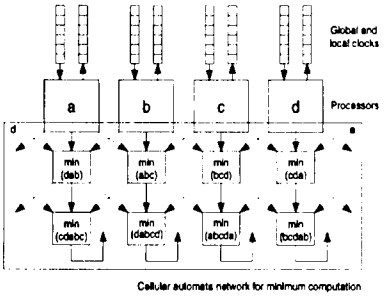


Figure 8: Digit-serial implementation of the global minimum computation and broadcast

just as in a pipelined architecture. The main problem with such an implementation is that it requires all the processors to cooperate in an SPMD execution. With the *phase algorithm* we described in [18], every processors have to synchronize at the end of each phase. So the SPMD computation is undertaken immediately after this synchronization.

4 Conclusion

The main benefits of using reconfigurable technology in the PDES domain are possibilities for:

1. synthesizing operators for model evaluation suited to the problem or the data size.
2. synthesizing specific coprocessors able to assess global or partial reduction over distributed values.
3. building a *simulation machine* efficient for larger application space than the previous ones, and combining the speed of hardware solutions with the flexibility of the programmable ones.

We have presented two machine-configurations. A fine-grained time-driven simulator with 8 nodes performs one step in 64 ms for a 350×370 cell model. A coarse-grained event-driven simulator computes each next event timestamp in 0.4 ms. Currently, a parallel simulator is being ported on the machine and will allow performance studies on hardware accelerators for the different asynchronous protocols. In particular, we have shown that it is possible to create a circuit able to evaluate minima of values belonging only to a subset of processors from which a node can receive messages. This possibility will be used to implement a kind of *deadlock detection and recovery* protocol.

The ArMen machine is used as an experimental machine to investigate the use of FPGA inside parallel machines and the influence of specific hardware resources on algorithms and performances. First experiments show that ArMen, as a versatile machine, can improve all types of parallel simulations, from fine-grained time-driven to coarse-grained event-driven.

References

[1] G. Nutt. Distributed simulation design alternatives. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 51–55, San Diego USA, January 1990.

[2] K. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transaction on Software Engeneering*, 5(5):440–452, September 1979.

[3] J. Peacock, E. Manning, and J. Wong. Distributed simulation using a network of processors. *Computer Network*, 3(1):44–56, February 1979.

[4] R. Fujimoto. Parallel discrete event simulation. *Transactions of the ACM*, 33(10):31–53, October 1990.

[5] J. M. Filloque. *Synchronisation répartie sur une machine parallèle à couche logique reconfigurable.* PhD thesis, Université de Rennes I, France, November 1992.

[6] T. Toffoli and N. Margolus. *Cellular automata machines.* MIT Press, 1987.

[7] K. Bouazza, J. Champeau, P. Ng, B. Pottier, and S. Rubini. Implementing cellular automata on the ArMen machine. In P. Quinton and Y. Robert, editor, *Proceedings of the Workshop on Algorithms and Parallel VLSI Achitectures II*, pages 317–322, Bonas, France, June 1991. Elsevier Science Publishers B.V.

[8] D. Jefferson and H. Sowizral. Fast concurrent simulation using the Time Warp mechanism. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 63–69, San Diégo, USA, January 1985. SCS.

[9] P. Reynolds. Efficient Framework for Parallel Simulations. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 167–174, Anaheim, USA, January 1991. SCS.

[10] XILINX. *The Programmable Logic Data Book.* Xilinx, 1994.

[11] B. Pottier. *ArMen, une machine parallèle intégrant un réseau de circuits logiques reconfigurables.* PhD thesis, Université de Rennes I, France, June 1991.

[12] P. Dhaussy, J.-M. Filloque, B. Pottier, and S. Rubini. Global control synthesis for an MIMD/FPGAmachine. In I. C. S. Press, editor, *IEEE Workshop on FPGAs for custom computing machines*, pages 51–58, Napa, California, April 1994.

[13] P. Dhaussy, J.-M. Filloque, B. Pottier, and S. Rubini. ArMen, an FPGA-based parallel architecture. In I. C. S. Press, editor, *Parallel System Fair, IPPS'94*, pages 43–49, Cancun, Mx, April 1994.

[14] J.-M. Filloque, E. Gautrin, and B. Pottier. Efficient global computation on a processor network with programmable logic. In *Proceedings of PARLE'91*, number 505 in LNCS, pages 55–63, Eindhoven, NL, June 1991. Springer-Verlag.

[15] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, (2):161–175, 1987.

[16] D. Désérable and J. Martinez. Using a cellular automata for the simulation of flow of granular material. In *Powders And Grains 93*, July 1993.

[17] T. Toffoli. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica 10D*, 1984.

[18] C. Beaumont, J.-M. Filloque, and B. Pottier. Efficient synchronous parallel discrete event simulation using the ArMen architecture. In A. Pave, editor, *Proceedings of the 1993 European Simulation Multiconference*, pages 611–615, Lyon (France), June 1993. SCS.

CONTROL OF CLOS REARRANGEABLE SWITCHING NETWORKS : ON THE NEIMAN ALGORITHM

I. Sakho

Ecole Nationale Supérieure des Mines de St-Etienne - Centre SIMADE
158, Cours Fauriel - 42023 St-Etienne Cédex - France
Tel : (33) 77 42 01 66 - Fax : (33) 77 42 00 00 - Email : sakho@emse.fr

Abstract : The Clos rearrangeable switching networks are interconnection networks used in applications as diverse as telephone exchanges and parallel computers. Among the large variety of the algorithms for the control of these networks, Neiman algorithm is representative of the most used. It consists in two phases whose the second one is in general too expensive.

This paper reports a modification of the first phase to lower the probability of the second one. When the interconnection matrix contains at most 2 non null elements by row and by column, the modification leads to a control algorithm similar to looping algorithm and then does not necessit a second phase.

Keywords : parallel computers, permutation networks, Clos rearrangeable switching networks, bipartite graphs edges coloring.

I . Introduction

In the use of Distributed Memory MIMD computers (DM MIMD), it is now well known that, among many others criteria, closer to the processes graph is the processors graph better will be the performance of an application.

DM MIMD computers with fixed interconnection network are not always well suitable to bring closer these two graphs. On contrary dynamic or reconfigurable interconnection networks, with their ability to bring neighbours two distant processes by connecting directly the processors on which they are running, allow to overcome this difficulty. To interconnect a very large number of processors, these networks consist in several crossbars organised in stages; one calls them multistage switching networks.

In the large variety of multistage switching networks, the Clos three stages rearrangeable networks [1] belong to the most attractive. Indeed they necessit a low cost of crosspoints, induce a low delay and, because of their rearrangeability, are able to perform any permutation that is to say any processors interconnection network. Many industrial realisations of reconfigurable parallel computers use this kind of switching networks [2], [3], [4], etc. The price to pay for the flexibility of these switching networks is the control: the computation of the command to execute to perform a given configuration.

Several control algorithms have been proposed in the literature among which Neiman algorithm[5]. It consists in two phases. The first is relatively simple. Second phase said completion phase, necessary if the first does not succeed is more complex and could be too expensive.

In [6] Tsao-Wu reports a modification of the first phase to lower the probability of the second one. Although interesting for large switching networks, this modification, does not induce a noticeable improvement of the cost of the second phase.

More recently Jajszczyk [7] proposed an algorithm which should not necessit a second phase. In [8] it is proved that this algorithm does not always provide information enough to solve the problem without a second phase.

In this paper we address the problem of the existence of an algorithm that does not necessit a completion phase. Section II presents the Neiman interpretation of the control of the Clos three stages rearrangeable switching networks. In section III we proceed to an analysis of the avoidance of a completion phase in the control of these networks; then we present a strategy to lower the probability of this completion phase.

II . Interpretation of Clos three stages switching network control

A Clos three stages rearrangeable switching network consists in m dxd input modules, d mxm intermediate modules and m dxd output modules. The modules are interconnected as follows: the k-th output (input) of the i-th (j-th) input (output) module is connected to the i-th (j-th) input (output) of the k-th intermediate module.

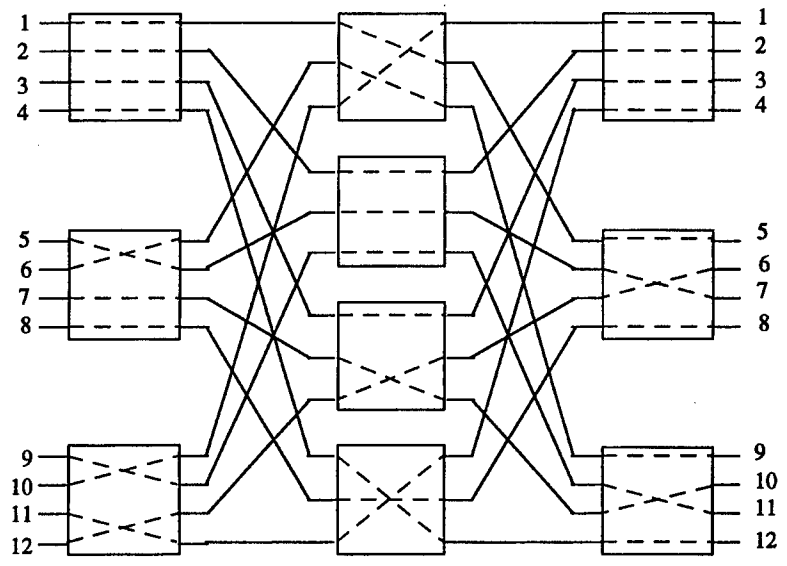


fig.1 : A Clos three stages rearrangeable switching network.
The dotted lines correspond to a non blocking setting performing the permutation
$\pi = (5, 2, 3, 12, 7, 9, 10, 11, 1, 4, 6)$.

Performing a processors interconnection network through a Clos three stages rearrangeable switching network consists in the calculus of the settings of the modules. The settings of the outermost modules can be easily inferred from the intermediate modules one. So we will focus only on this latter.

The problem concerns the assignment of the processors interconnections to the intermediate modules such as there is no blocking that is to say two interconnections from (to) the same input (output) module are not assigned to the same intermediate module (see fig.1).

Let π be a permutation of N=md elements representing a set of processors interconnections. π defines a matrix H_π or simply H called interconnection matrix whose each component $H_\pi(i, j)$ represents the number of the connections that π induces between the i-th input module and j-th output module. Three obvious properties of H_π are :

(1) $\forall(i, j), H_\pi(i, j) \geq 0$,

(2) $\Sigma_{1 \leq i \leq m} H_\pi(i, j) = \Sigma_{1 \leq j \leq m} H_\pi(i, j) = d$.

(3) If P is a permutation matrix extracted from H_π, then H_π-P also verifies properties (1), (2) and (3).

To compute the settings of the intermediate modules, Neiman algorithm decomposes H_π in permutation matrices whose each one corresponds to the settings of one intermediate module. To that purpose the algorithm he proposes consists first in marking the non null elements of H_π such as there is one by row and by column.

When it is carried out without an appropriate strategy, this marking can fail that is the number of the marked elements can be less than m (see the example on fig.2 from [8]). Then a completion phase is necessary. This one is generally based on the Berge theorem for characterising a maximum matching in bipartite graph [9] and then could be too expensive.

$$\begin{bmatrix} 0 & 0 & 0 & \textcircled{2} & 1 & 0 \\ \textcircled{1} & 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & \textcircled{1} & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 1 & 0 \end{bmatrix}$$

fig.2 : The marked elements are circled.
The 5-th row can not contain a marked element.

III . Characterising of the algorithms without completion phase

Let us consider the interconnection matrix H associated to a processors interconnection network. Suppose that k non-null elements of H have been already marked. By appropriate permutations of the rows and the columns, H can be rewritten:

$$\begin{bmatrix} H_1^{(k)} & H_2^{(k)} \\ H_3^{(k)} & H_4^{(k)} \end{bmatrix}$$

1. Determine all the rows i* of $H_4^{(k)}$ with the largest number of zeros.
2. For all the rows i*, determine the non null elements whose the column j* contains the largest number of zeros ;
3. Mark the element $H_4^{(k)}(i^*, j^*)$
4. Delete the row i* and the column j*.

Here is an application of the algorithm.

1	0	0	0	②	1	0
2	1	0	0	0	0	②
4	①	0	1	1	0	0
6	0	1	0	0	①	1
3	1	0	②	0	0	0
5	0	②	0	0	1	0

fig.3 : The marked elements are circled.
On the left is the order of the marking.

When we apply this strategy, one can get several non null elements to mark. In this case it is better to mark the largest one as this could allow to generate more than one permutation matrix at the same time.

Because $|\Gamma(I_2^{(k+1)})| = |I_2^{(k+1)}| + \varepsilon - 1$ with $\varepsilon \geq 0$can be chosen in keeping $|I_2^{(k+1)}|$ as small as possible while $|\Gamma(I_2^{(k+1)})|$ will be as large as possible.

Now let us evaluate the algorithm. Let x be the largest number of the non null elements which have been marked and $S(H_1^{(x)})$ be the sum of all the elements of $H_1^{(x)}$; we have $H_4^{(k)} = 0$ and $S(H_1^{(x)}) = \Sigma_{1\leq i\leq x}\Sigma_{1\leq j\leq x} H_1^{(x)}(i, j)$. One can also verify that $S(H_1^{(x)}) = (2x-m)d$.

Suppose that $H_4^{(x)}(i, i)$ is the i-th element to be marked. Let $r \in [x+1, m]$ be such as $H_3^{(k)}(r, .)$ is non null that is to say that the r-th row of $H_3^{(x)}$ is non null. Then the set $B = \{i \in [1, x] : H_3^{(k)}(i, .) = 0\}$ is not empty.

Indeed let r(i), $1 \leq i \leq n_r$, be the column numbers of the non null elements of $H_3^{(k)}(r, .)$. According to the marking strategy, for $p \in [r(n_r-1), r(n_r)]$, $H_4^{(p)}(p+1, .)$ must contain at least as much zeros as $H_4^{(p)}(r, .)$. Thus $H_4^{(r(n_r-1))}(i, j) = 0$ for $i \in [r(n_r-1)+1, r(n_r)]$ and $j \in [i+1, m]$; from where $B \neq \emptyset$ (see fig.4).

where:

- $H_1^{(k)}$ ($H_4^{(k)}$) consists in all the elements whose the row and the column contain (does not contain) a marked element and $H_1^{(k)}(i, i)$ is the i-th element to be marked,
- $H_2^{(k)}$ ($H_3^{(k)}$) consists in all the elements of H whose only the row (column) contains a marked element.

In the following $H_p^{(k)}(i, j)$ means that H(i, j) belongs to $H_p^{(k)}$; so we will talk about the i-th row and/or the j-th column of $H_p^{(k)}$.

It is obvious that a necessary and sufficient condition for a marking strategy to be without completion phase is that for any $k \in [0, m-1]$, $H_4^{(k)}$ contains a (m-k)x(m-k) permutation matrix.

More formally, let A be a set of rows of $H_4^{(k)}$, $\Gamma(A)$ be the set of the columns j of $H_4^{(k)}$ such as $H_4^{(k)}(i, j) \neq 0$ for $i \in A$. According to the Koenig-Hall theorem [9] a necessary and sufficient condition for $H_4^{(k)}$ to contain a (m-k)x(m-k) permutation matrix is

$$(4) \quad |\Gamma(A)| \geq |A| \; \forall A \; .$$

For any matrix H that verifies properties (1) and (2) of the previous section, $H_4^{(0)}$ verifies the relation (4). Identically, it is obvious, whatever the first non null element to be marked, that $H_4^{(1)}$ also verifies the relation (4).

But from now, choosing a non-null element to be marked must be carried out carefully. Thus, given $H_4^{(k)}$ that verifies the relation (4), we are faced to the problem of the choice of a non-null element such as $H_4^{(k+1)}$ also verifies the relation (4).

Let $H_4^{(k)}(i^*, j^*)$ be the non null element chosen. As each row (column) must contain one marked element, we can choose i^* arbitrarily. Furthermore because each non null element $H_4^{(k)}(i^*, j)$ defines a distinct $H_4^{(k+1)}$, it is obvious that searching for j according to the relation (4) could be too expensive.

IV . An efficient marking strategy

Given the cost of searching for j^*, in this section we will focus on the problem of the choice of $H_4^{(k)}(i^*, j^*)$ that instead minimises the cost of the completion phase if this one becomes necessary.

To that purpose we first choose i^* as the row which contains the largest number of zeros . Indeed in this way, the number of the columns candidate to be j^* is reduced as much as possible. Then let $I^{(k)}$ the set of the rows of $H_4^{(k)}$. $I^{(k+1)}$ can be written as $I_1^{(k+1)} \cup I_2^{(k+1)}$ where $I_1^{(k+1)}$ ($I_2^{(k+1)}$) is the set of the rows $i \in I^{(k)}$ such as $H_4^{(k)}(i, j^*) = 0$ ($H_4^{(k)}(i, j^*) \neq 0$).

For any $A \subseteq I^{(k)}$ the relation (4) is verified. However, when $A \cap I_2^{(k+1)} \neq \varnothing$ we have $|\Gamma(A)| \geq |A| - 1$; these are the only case for which relation (4) could not be verified. For such a set, the largest number of rows of $H_4^{(k)}$ which could not contain a marked element is $|A \cap I_2^{(k+1)}|$. Thus this number has to be minimised to increase the probability to mark more non null elements. That can be obtained by minimising $|I_2^{(k+1)}|$. This points to the following marking strategy :

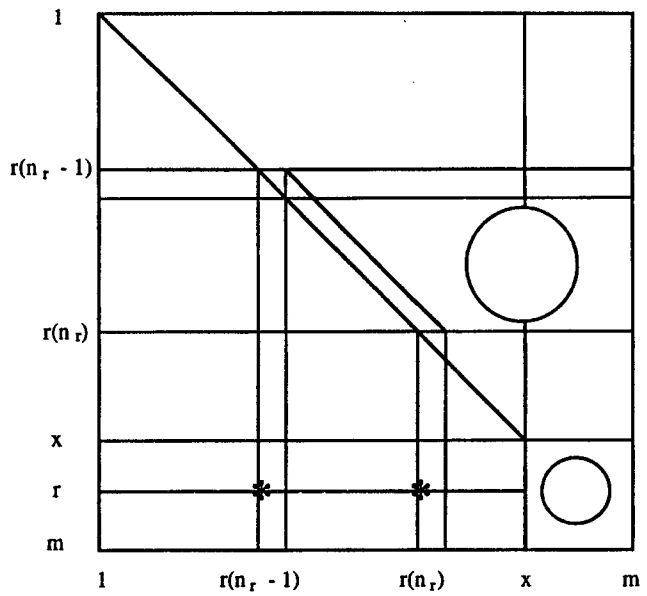


fig.4 : The * sign indicates the non null elements of the row r.

One verifies easily that $S(H_1^{(x)}) \geq |B|(d-1)+x$; as $S(H_1^{(x)}) = (2x-m)d$ then it follows that

$$x \geq ((m+|B|)d - |B|)/(2d-1).$$

In general, it is not obvious to evaluate $|B|$. However, it could be large enough to avoid a completion phase. As an illustrative example, let us suppose that by row and by column H contains at most p non null elements.

Proposition : If p = 2 then the marking strategy is without completion phase.

Proof : We don't restrict the generality by considering that H contains by row and by column two non null elements. Indeed the strategy will first mark all the non null elements with value d.

Now let us remark that when k non null elements have been marked, only one of the rows of $H_4^{(k)}$ contains the largest number of zeros. Let $r \in [x+1, m]$ be such as $H_3^{(k)}(r, .) \neq 0$ and r(1), r(2) the column numbers of the non null elements. After the r(1)-th step of the marking, only the row r contains the largest number of zeros. According to the marking strategy, $H_1^{(x)}(r(1)+1, r(1)+1)$ is necessarily $H_3^{(x)}(r, r(2))$. Thus $H_4^{(x)}(r, .) = 0$ instead (see fig 5); from where $S(H_1^{(x)}) = xd$. As $S(H_1^{(x)}) = (2x-m)d$, it follows that x = m and then no completion phase is needed.

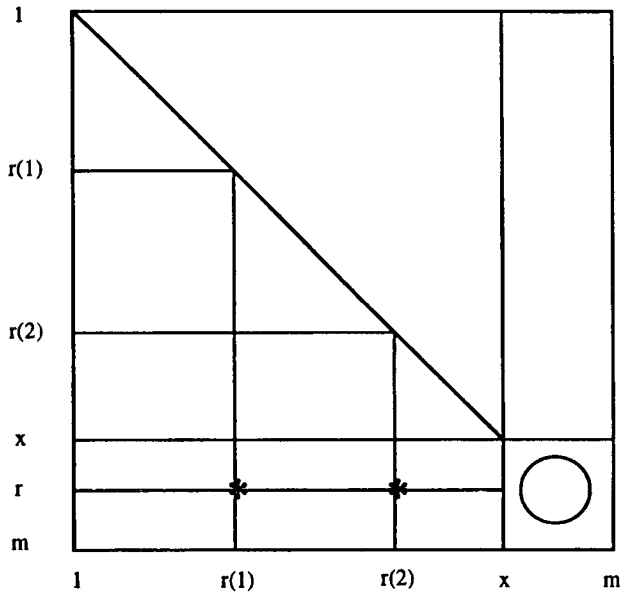


fig.5 : The * sign indicates the non null elements of the row r.

V . Concluding remarks

This paper reports a modification to the first phase of the Neiman algorithm for the control of Clos three stages rearrangeable switching networks. When the interconnection matrix contains at most 2 non null elements by row and by column, the induced algorithm works as the looping algorithm [10]. In the general case, the number of the marked elements depends on $|B|$, the number of the null rows induced in $H_2^{(x)}$. This one has to be evaluated more carefully; this will be the next step of our work. However we can make the following remarks.

At each step k, the largest number of zeros is deleted from the matrix $H_4^{(k-1)}$ and then in the same time the probability to find at the (k+1)-th step a non null element to mark is increased. Furthermore the algorithm, at each step, maintains as small as possible the rows will do not contain a marked element when it fails. This is guaranteed by choosing j* as the one which contains the largest number of zeros.

V. References

[1] C. Clos : "A study of non blocking switching networks", The Bell system technical journal, March 1953.

[2] J. Beetam, M. Denneau, D. Weigarten : The GF11 supercomputer", Proc. 12th annual International Symposium on computer architectures, IEEE 1985.

[3] V. P. Bhatkar : "Parallel Computing. An indian perspective", Proc. of CONPAR 90 - VAPP IV, LNCS 457, Springer-Verlag, H. Burkhart (Ed).

[4] P. Waille, T. Muntean : "Introduction à l'architecture des machines supernodes", La lettre du Transputer, n°7 1990.

[5] V. I. Neiman : "Structures et commandes optimales de reseaux sans blocage", Annales des Telecom, Juillet-Aout 1969.

[6] Tsao-Wu : "On the Neiman's algorithm for the control of rearrangeable switching networks", IEEE Trans. on Comp., Vol c-22, June 1974, pp 737-742.

[7] A. Jajszczyk : "A simple algorithm for the control of rearrangeable switching networks", IEEE Trans. COM-33, pp 169-171.

[8] J. Gordon, S. Sritkanthan : "Novel algorithm for Clos-type networks", Electronics lettres, Vol 26 n°21 October 1990, pp 1772-1774.

[9] C. Berge: "Graphes", Gauthier Villars, 1983.

[10] S. Andresen : "The looping algorithm extended to the 2^t rearrangeable switching networks", IEEE Trans. on Comp., Vol c-25, 1977, pp 1057-1063.

HIGHLY PARALLEL ASYNCHRONOUS COMPUTER SYSTEMS FOR THE LARGE DATA ARRAYS PROCESSING

Prof. Arkady B.Barsky
Moscow State Railway University,
15, Obraztsova Str., Moscow, 101475, RUSSIA
Tel: (095)284-24-04; fax: (095)281-13-40

Dr Valery V.Shilov
Scientific Computing Centre of the Russian Academy of Sciences,
32a, Leninsky Pr., Moscow, 117334, RUSSIA
Tel: (095)938-18-93; fax: (095)938-58-84; e-mail: user@comcp.msk.su

Summary.

Two multiprocessor computer system architectures are described which can potentially achieve highly parallel execution of programs, represented in traditional form. The first architecture combines some features of von Neumann and data flow computers, and the second one called locally asynchronous monoprogram is similar to sample program - multiple data computers. Numerous examples illustrate main organization and operation principles of proposed computers.

Keywords.

Computer architecture, parallel computations, multiprocessor architecture, data flow, asynchronous computations.

1. Introduction.

It is evident enough that modern requirements to computer's performance may be satisfied only by creating highly parallel computer systems that integrate thousands and even millions relatively simple processors. The exploiting of large number of processors for the joint data processing closely connected with such problems as computer topology and main memory structure. But any restrictions imposed on these characteristics lead to narrowing of effectively solving problem's class (the measure of efficiency is there the executive devices loading factor).

However, the design of the general purpose computer system with a large number of processing elements (PE's) which is not specialised in any class of problems may be connected to all appearance not only with the search of particular topologies and memory structures, but with return to the most general original conception of multiprocessor system with the shared memory. The presence of such memory solves the problem of interprocessor communication.

In parallel computer systems to decrease memory collisions influence, memory interleaving is widely employed. So the problem arises to communicate large number of PE's with the large number of memory modules. The well-known decisions realised in matrix commutators are too complicated. Thus to demonstrate the quality of highly parallel computer systems with the shared memory and to show the advantages of their programming (and it is the primary goal of the present paper) mean to attract designers' attention to some aspects of the commutation problem, especially to necessity analyse the possibility of fast simple uni- and bi-directional commutators' design.

Not less difficult than the commutator design is the problem of maximum loading of all PE's during the computation period, i.e. the efficient paralleling problem. It is well known that two different paralleling methods exist - by control and by data. The first method based upon an analysis of informative and logical graph of algorithm and con-

sists in the distribution of partially ordered program fragments between processors. This method may be used while OS program or task scheduling, but the number of processors may not exceed tens.

The second one supposes data elements' distribution between PE's for the following identical processing and is widespread in vector and matrix computer systems. Obviously, the design of highly parallel computer requires to use both methods (although in our opinion data paralleling is the main source of PE's maximum loading).

2. Data flow computer system.

But to use the advantages of data paralleling staying in the frameworks of traditional von Neumann architecture is a very difficult task. The main difficulty there consists in the discrepancy of the program static representation and the dynamics of its execution. The data flow computational model avoids this and some other problems arising while parallel computations. That explains the popularity of data flow conception, although too many questions are need the answer till now.

On the other hand, the simplicity and habitual character of traditional programming, together with the potential possibility of computers with the shared memory forced us not to reject the von Neumann model completely but to accept it partially (in particular, to keep program counter).

The architecture of proposed data flow computer presented in Fig.1.

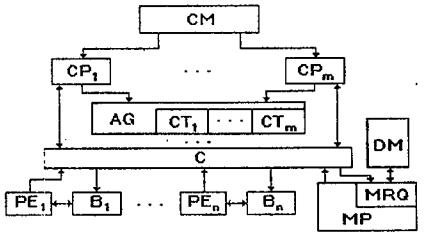


Fig.1. A data flow computer.

The main blocks of data flow computer are the following: m commutation processors (CP's); address generator (AG) which includes a set of correspondence tables (CT's); executive devices and their buffers, commutator (C) and data memory (DM) which is the main memory of the system. The set of executive devices consists of processing elements with attached buffers (B's) and memory processor (MP) which performs all data interchanges with DM. MP includes its internal buffer called memory requests queue (MRQ).

Let us examine the principles of data flow computer operation.

Every CP performs its own commutation program (let it call "program" because it is the only representative of the computation process) which corresponds to the task, process, program module and so on, realising so the coarse-grain parallelism. The program ought to be written to the command memory (CM) preliminary.

Program instructions may contain PE's mathematical addresses interpreting as special memory area addresses (in this area the repeated use of data is not allowed).

During the commutation process every mathematical address must be mapped onto a physical address of some B register. CP's and PE's operate independently with the exception of conditional jumps defining by the computation results.

While running instructions from the basic instruction set, CP (if it has found the mathematical address in the text of instruction) with the help of AG reserves one of B's or MRQ registers (cells) in the special order. (Mathematical address, physical address) pairs, formed so, are written to the CT of this CP.

Besides, CP writes opcodes and physical addresses of cells, where it must send operation result, into the mentioned register.

Every buffer' cell has the format as shown in Fig.2.

Operation code θ	Operand 1	Operand 2	Operand 3	Operand 4	Result	Destination address

Fig.2. Buffer' cell format.

(Four operands are used in the special operations as it will be shown below).

Commands forming in the MRQ are analogous to that. READ command must have the address of the cell in some buffer, where the data value must be sent. While forming WRITE command its address in MRQ is sent to the instruction that commutates the calculation of it.

Thus CP commutates executive devices dynamically for the joint algorithm realisation.

All executive devices, operating asynchronously, perform instructions from their buffers, which obtain in its text the full set of operands. The result is sent to destination address (in the text of some other instruction) forming so new ready to execution instructions.

Let choice the traditional structure for the commands of (commutation) program. It is three address instructions of the next shape:

$\theta\ M_1\ A_1\ M_2\ A_2\ M_3\ A_3$,

where M_1 is an address of modifier (index register) and the effective address A_{eff} $i=A_i+(M_i)$.

As an example we may examine the program that searches the maximum element of array $\{a_k\}$, k=0,...,p-1, by the pyramid method (array components are matches in pairs and then this process repeats over the previous matching results as it shown in Fig.3 for p=7).

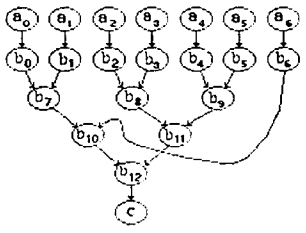


Fig.3. A computational scheme (searching of maximum array' element).

All of b_i, i=0,...12, from Fig.3 are the mathematical addresses of PE, which transforms into the physical addresses in the PE buffers during the commutation process.

Fig.4 shows the corresponding program realising the present scheme of computations.

N_k	Θ	M_1	A_1	M_2	A_2	M_3	A_3
0	LOAD	M1	$<<a_0>>$	M2	$< n >$	M4	
1	LOAD	M3	$<< c >>$	M5			
2	WRITE	M1					b_0
3	CYCLE	M2	17777				
4	WRITE	M1	00001			M4	b_1
5	MAX	M5	b_0	M5	b_1	M2	b_0
6	M+	M1	00001	M2	00001		
7	M+	M4	00001	M5	00002		
8	ENDC						
9	WRITE	M5	b_0			M3	

Fig.4. A program searching maximum array' element.

In Fig.4 opcode WRITE denotes the command, which must write (A_1) value to the DM cell defined by A_3 effective address; CYCLE and ENDC (END of Cycle) command organises the cycle with the number of iterations defined by A_1 effective address; MAX command realises the following operation:

$(A_3):=\max\{(A_1),(A_2)\}$; M+ is the vector command changing the contents of the corresponding modifiers; $<a_0>$ is the base address of array a; n is the quantity of elements in array a; <c> - the address of result. The elements of vector a are placed in the contiguous cells of the data memory.

The mathematical address of PE, b_0, is arbitrary chosen for ever program address (it is convenient to compare it to zero) and marked out by the corresponding tag; $b_1=b_0+1$.

It is obvious that by simple replacing opcode MAX in the string 5 (Fig.4) by PLUS, MULT, MIN or other opcodes, our program may be transformed to the program of summing up or multiplying vector elements, of searching minimum element in array, etc. accordingly.

It is necessary to remember, that in spite of the great formal resemblance of considered program and the usual program written for the some traditional computer, they have at least one but root difference. Our program is a program of commutation that arranges connections between computer executive devices only and separated from the computation process completely.

Let us mark, that the principal possibility exists (which limited only by the buffer's size) to commutate the algorithm beginning with the arbitrary point of its graph.

For some operations (arithmetical and logical) over two arrays the vector commands may be constructed. Modifiers' addresses used in such a command correspond to the base of array descriptors. Every array descriptor may consist of not more than three adjacent index registers.

Let in a command

$B\theta\ M_1\ A_1\ M_2\ A_2\ M_3\ A_3$

letter "B" indicates vector operation with opcode θ, $\{M_v, M_v+1, M_v+2\}$ is array V descriptor, where M_v contains base address, M_v+1 contains vector V augment and (M_v+2) is equal to the vector v size, v=1,2,3.

Having $\{a_k\}$ vector, we can form three arrays: $\{a_0, a_2,\ldots,a_{2k-4}\}$, $\{a_1, a_3,\ldots,a_{2k-3}\}$ and $\{a_k,\ldots,a_{k-2}\}$ on its base. Both the first and the second array descriptors consist only of one index register, M1 and M1+1 accordingly, because the second array base may be obtained using displacement and we don't need the number of array elements. The third

vector is described by descriptor {M2, M2+1, M2+2}. Fig.5 shows the same maximum element searching program modified by including vector command into it.

N_k	Θ	M_1	A_1	M_2	A_2	M_3	A_3
0	LOAD	M1	$<<a_0>>$	M2+2	$< n >$	M1+1	00002
1	LOAD	M2	M1	M3	$<< c >>$	M2+1	00001
2	M+	M2	M2+2	M2+2	17777		
3	BMAX	M1		M1	00001	M2	
4	M+	M2	M2+2				
5	WRITE	M2	17777			M3	

Fig.5. A program searching maximum array' element using vector operation.

The three initial instructions, 0-2, forms array descriptors and the result address. Instruction 4 commutates the main vector command. Instruction 5 commutates the result writing to DM. Note, that by insignificant changes both in the scheme and the program we may realise some other algorithms of convolution type, among them scalar product of vectors, integration and so on.

The possibility exists to commutate parallel execution of conditional alternative branches with the following choice of desirable result. It explains the presence of four operand fields in the PE buffer register (Fig.2).

To exclude conditional jumps from the computation of the arithmetical terms with conditions, the special five address COND command may be used. It needs two instructions in the program text for its placing and realises the following operator:

(A_3) := if (A_1) <operation> (A_2) then (A_4) else (A_5).

For example, Fig.6 presents the program computing the value of conditional statement:

A:=d* if a<0 then if x*y<5 then 0 else (a*x+b) else if a>5 then 0 else a*b.

N_k	Θ	A_1	A_2	A_3
0	MULT	x	y	(1,1)
1	MULT	a	x	(2,1)
2	MULT	a	b	(3,1)
3	PLUS	(2,1)	b	(4,1)
4	COND	<5>	a	(1,2)
		<0>	(3,1)	
5	DIV	(4,1)	c	(2,2)
6	COND	(1,1)	<5>	(3,2)
		<0>	(2,2)	
7	COND	a	<0>	(4,2)
		(3,2)	(1,2)	
8	MULT	d	(4,2)	A

Fig.6. A program computing the conditional statement.

Some parts of instructions omitted in the text of this program (for instance modifiers' fields so far as modifier's implementation is not necessary there).

The traditional character of programming for this computer admits to keep the succession of imperative programming languages and compilation techniques.

3. Highly parallel monoprogram computer system.

The design of computer systems with a large number of PE's is conditioned by the necessity of homogeneous data arrays processing. The predominance of such processing defines the possibility of monoprogramming that allows to perform copies of one program by every PE. Data elements are distributed automatically between PE's according to some discipline. For that, the executing branches of monoprogram may not be identical. The synchronization of data elements processing is expedient at the instruction level. So asynchronous at the whole PE's operation may be limited if need by synchronization. Let us call such structures locally asynchronous ones [1].

The structure of locally asynchronous monoprogram computer system presented in Fig.7. The program locates in command memory and is read by fragments into PE's local command memory buffers (CB). Every PE contains its own local memory (LM) module, in which stacks, modifiers, array descriptors and local data are stored. PE's connected with data memory (consisting of p modules S_i, i=0,...,p-1) via commutator (C). Synchronisation block (SYNCH) provides simultaneous PE's operation in the case of special instruction performing by any PE. Closed address's memory (CAD) block intended for computation synchronisation when the shareable data is used; data reading from marked cell is delayed until writing into it. Semaphore's memory (SEM) block stores synchronisation primitives (semaphores) and performs operations over them.

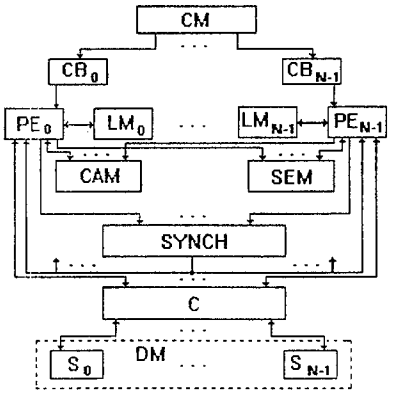


Fig.7. A locally asynchronous computer.

If computer system oriented to computational problems solving, the use of CAD is preferable because it is more delicate synchronisation tool realising the data flow principle. On the contrary, while running control, optimisation and artificial intelligence programs the use of semaphores is quite enough.

The important elements of considered computer system architecture and programming for it are array descriptors and the set of operations over them, as it will be shown below.

3.1. Vector operation of convolution type.

Let us compute the product of array $\{a_v\}$, v=0,...k-1, elements by the same "pyramid" method and let the number of processor elements N be equal 4.

The computation scheme shown in Fig.8 (for k=10).

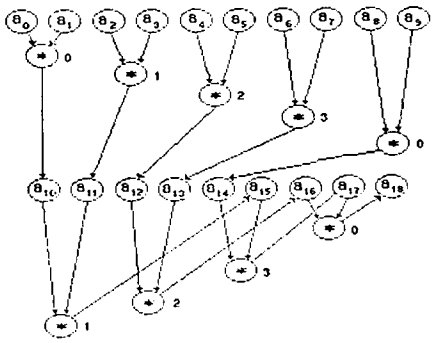


Fig.8. A computational scheme.

The nodes $a_{10},\ldots,a_{17}$ intruded, which correspond to the intermediate result's computation, and a_{18} node that corresponds to the out result computation. Every node marked out by the number of PE performing this operation. This number is not attaches to the node while computation planning (because the program is N and k invariant).

Every PE obtains the full set of array' descriptors (AD), consisting maximum of eight AD elements (if need be, some elements may be omitted). The AD on the whole and every AD element in particular is addressable; AD elements are places in the contiguous cells.

Let us form in LM_i, i=0,...,N-1, descriptor D_1 of $a=\{a_0, a_2, a_4,\ldots,a_{2k-4}\}$ array consisting of eight descriptor' elements; $D_1=\{D_{10},\ldots,D_{17}\}$. D_{10} contains the base address $<a_0>$ of array a; D_{11} contains the augment h, h=2; D_{12} is equal to k-1 (i.e. the number of array' a elements); D_{13} is the address of the last array' element, $<a_{2k-4}>$. D_{14} element serves for automatical augmentation while successive accesses to array a. Its source value is $<a_0>$; after j accesses to array a, D_{14} becomes equal to $<a_0>+jh$, where j may be the cycle index. Every successive access to array a performs the following operation: $(D_{14}):=(D_{14})+(D_{11})$.

The next group of descriptor' elements intended for automatical distribution of array elements between PE's. D_{15} contains address $<a_0>+ih$, where i is the number of given PE. It means that every PE forms its own D_{15} value, disposing of the address of some register that stores the PE number. So the initial access to "its" element of array a becomes possible.

D_{16} stores Nh value using for access to the next "its" array element. D_{17} contains the current value $<a_0>+ih+jNh=(D_{15})+j(D_{16})$, using for automatical augmentation while successive access to array descriptor (j=0, 1,...).

Then let us construct descriptor $D_2=\{D_{20},\ldots,D_{27}\}$ of $a'=\{a_1,\ldots,a_{2k-3}\}$ array. The difference between D_1 and D_2 determined by the values of the first and the last array elements' addresses; besides, the value of h is equal to 1.

Fig.9 shows the corresponding monoprogram written with the use of three address commands of proposed computer system' instruction set.

N_k	Θ	M_1	A_1	M_2	A_2	M_3	A_3
0	SYNCH						
1	CKADR	D_{15}		D_{25}			0007
2	CLADR	D_{27}					
3	MULT	D_{17}		D_{17}	0001	D_{27}	
4	CDWR	D_{27}		D_{23}		M	
5	CHADR	D_{17}		D_{27}			0007
6	FJUMP		0002				
7	EXIT						

Fig.9. A program multiplying array' elements.

Instruction 0 (SYNCH command) synchronises the system for the simultaneous performing of the next instruction. Every PE sends the signal to the SEM block, which returns the answer when and only when the signal from all PE's will be received. After receiving the answer PE's may continue their operation.

CheckK ADdRess command compares (D_{15}) address with (D_{13}) address and (D_{25}) address with (D_{23}) address. This command permits to do two compares simultaneously. If (D_{15})>(D_{13}) $\vee$ (D_{25})>(D_{23}), then the jump to instruction 10 (whose number written in the third address field) carries out. By the means of this command we verify the belonging of addresses which PE can "see" initially, to the set of array' elements' addresses. It allows to exclude automatically from the computation some PE's that are not provided by array' elements. If N is equal to 4, all PE's begin to perform the next instruction.

Instruction 2 (CLose ADdRess command) writes the value of descriptor' element D_{27}, i.e. (D_{27}), to the CAM. While the first execution, for PE0, (D_{27})=a_{10}, and for PE_3 - (D_{27})=a_{13}.

Instruction 3 defines the multiplication of two array' elements. The effective code (mult a_0 a_1 a_{10}) forms on PE_0, the effective code (mult a_2 a_3 a_{11}) forms on PE_1 and so on. After instruction performing and results writing to memory, $a_{10},\ldots,a_{13}$ addresses will be excluded from CAD.

It is obvious that if N>entier(k/2) than in the effective code of instruction 3 formed on $PE_{entier(k/2)},\ldots,PE_{N-1}$, previously closed by other PE's addresses are used. The attempt to perform such instruction (i.e. the attempt to read from the cell with the closed address) will be repeated until PE that had closed this address completes multiplication and writes the intermediate result to this cell.

Instruction 4 (CoNDitional Write) in case if (D_{27})=(D_{23}) reads the code from the cell which address stores in D_{27} (it is the out result address) and writes it to A_3 address.

Instruction 5 (CHange ADdRess) performs the following operations: (D_{17}):=(D_{17})+(D_{16}); (D_{27}):=(D_{27})+(D_{26}). If the values of newly computed addresses are greater then last array element address, i.e. (D_{13}) and (D_{23}) respectively, the control is transferred to instruction 7.

Instruction 6 is the Fast JUMP command that realises the jump for repeated performing of instruction 2.

The bounds violation while the repeated performing of instruction 5 in this example leads to termination of computations on processor elements PE_1-PE_3. P_0 performing instruction 4, will write the out result to address storing in M register. The third performing of instruction 5 on PE_0 also terminates computation process on it.

Instruction 7 (EXIT) performs return from this subprogram.

3.2. List processing.

In spite of the strictly sequential structure of lists, the parallel list processing is possible [2]. It bases on the sequential list element' processing reducing to the parallel array element' processing. This array is the "image" of the list to be processed. The list' image and the list itself are formed and processed simultaneously. Every list image element contains the name (number) of this one, data reference and the next list element reference.

While programming list processing monoprogram, this list' image is the base array. List elements are changed by means of data reference.

Let the array R be list image; each element of array R is the following record: (n, V, C, D). Here n is the name (number), V - data reference, C - the next list' element reference, and D is an auxiliary reference to another field of the next list' element.

Let the value S of some list' element be defined. The reference C_j exists in array $R=\{(n_j, V_j, C_j, D_j)\}$ that points at S. Knowing S, it is necessary to find list' element, corresponding to it, and to exclude it from the list.

As in the Section 3.1, array' R descriptor, D_R, may be constructed. D_R consists of maximum eight descriptor' elements $\{D_{R0},\ldots,D_{R7}\}$. The meaning of some elements will be explained below.

The joint processing elements' operation may be organized as follows.

Let all PE's "looks" at the beginning of the process at prescribed array' R elements. One of PE's will find the array element obtaining the reference to desirable list' element. This reference is replaced by the reference of the same name, storing in the excluding list' element. Array R element corresponding to excluding list' element is also excluded from array R. After that array' R descriptor is corrected.

As an example of list processing problem programming let us examine the program that searches and excludes some list' element. The program presented in Fig.10.

N_k	Θ	M_1	A_1	M_2	A_2	M_3	A_3
0	SYNCH						
1	NEQ	<i>					003
2	LOAD	K					
3	CKADR	D_{R7}					010
4	WRITE	D_{R7}	0002				M_1
5	WCODE	M_1	0001				M_2
6	NEQ		M_2		S		011
7	LOAD	K	001				
8	WRITE	M_1	0002			D_{R7}	0002
9	DEL	D_R		M_1			
10	EXIT						
11	NEQ	K					010
12	CHADR	D_{R7}					010
13	FJUMP		004				

Fig.10. A program searching and excluding list' element.

Instruction 0 synchronizes processing elements. Instruction 1 (Not EQual) is the conditional jump to the fourth string of the program, so only PE0, which number is equal to 0, will performs instruction 2 (the assignment of zero value to modifier K). This illustrates the possibility of parallel execution of monoprogram' branches by different PE's.

Instruction 3 defines the conditional jump to instruction 10, i.e. the exit from the program, if the current address stored in descriptor' element D_{R7} is greater than array R last element' address, stored in descriptor' element D_{R3}. If $(D_{R7})\geq(D_{R3})$ then the contents of current array R element' field C (i.e. the contents of $(D_{R7})+2$ cell) is written to M1 register (instruction 4).

Let the reference storing in the field C, points to the first address of array' R element. So (M1)+1 address contains the reference to this list element' value. Instruction 5 (Write CODE) writes this reference' value to register M2.

Instruction 6 compares the value of array' R element (it' address is the register M2 contents) with excluding element' value storing in S cell.

If (M2)=S then instruction 7 assigns modifier K the arbitrary non-zero value indicating so that the excluding list' element is found by some PE. In the case if (M2) is not equal to S, the control is transferred to instruction 11.

Instruction 8 performs the following operation: the value ((M1)+2) is assigned to $(D_{R7})+2$ element. It means that the value reference storing in array' R element that corresponds to excluding list' element, is assigned to the element that had pointed at the excluding element previously.

Instruction 9 (DELete). The element with address (M1) is excluded from array describing by D_R descriptor (after instruction 4 performing it is array' R element address corresponding to excluding list' element).

Instruction 10. Return from this subprogram.

Instruction 11 gets the control only if the reference in current array' R element don't point out excluding list' element. In this case PE checks the presence of sign indicating that some other PE already had found the desirable list' element and is ready to exclude it from the list. (If (k)≠0, then control is transferred to instruction 10; else, if (K)=0, PE must analyze the next array' R element that is "prescribed" to it).

Instruction 12 performs the readdressing as follows: $(D_{R7}):=(D_{R7})+(D_{R6})$, where $(D_{R6})=Nh$. If bound violation occurs, i.e. the value of D_{R7} became greater than the value of D_{R3}, then conditional jump to instruction 10 is to be performed; else beginning from instruction 13 PE will continue to analyze the next element of array R.

4. Conclusion.

Data flow computers, which have totally different architecture from those of the von Neumann computers, were expected for a long time to achieve highly concurrent execution of a program in a quite a natural way. But in our opinion the really high general purpose computer' performance may not be achieved within a framework of only one, though most attractive computational model. It needs the combining of different models features, applying at the different levels of computer organization. Two variants of possible decisions are described in this paper and it seems that the results achieved are hopeful enough to stimulate futher investigations.

References.

1. Barsky A.B., Shilov V.V. High-parallel monoprogram computer architecture. - "The 3rd St.Petersburg Int. Conf. "Regional Informatics-94", Abstracts of Reports", Part 1, SPb, 1994.

2. Hillis W., Steele G. Data Parallel Algorithms. Comm. of the ACM, v.29, N12, 1986, pp. 1170-1183.

Section V: Environments, Debugging and Monitoring

XDSM - an X11 based Virtual Distributed Shared Memory System.

A.D.S.Argile, A. Bargiela.

Department of Computing,
The Nottingham Trent University,
Burton St., Nottingham NG1 4BU, U.K,
arg@uk.ac.ntu.doc, andre@uk.ac.ntu.doc

Summary
An X11 based page-able shared memory system, permitting the implementation of a distributed water network monitoring and control software suite, is described in this paper. The system is based on the use of modular library and language specific interfaces, to access the underlying X11 communications platform supporting distributed shared memory (DSM). The system uses a server client relationship, with local computer node task managers, and uses the DSM for communication, configuration and coordination. An X11 based distributed mutual exclusion algorithm, based on the unconventional use of Lamport's bakery algorithm, is illustrated.

Keywords
X11, Distributed Shared Memory, XDSM, Bulk-synchronous parallel processing.

1. Introduction

When parallel tasks execute on a single processing node their data communication requirements can be provided by means of shared memory. This is because the shared memory paradigm gives the programmer a shared address space linking separate processes. It provides a logical view of data which abstracts out the coding requirements from the actual complexities of the physical data transfer.

Conversely, the conventional method of intertask communication via message passing, forces the programmer to be acutely aware of message source, destination, transmission protocol(s), and format. Thus message passing based communication can become quite complex in dynamically evolving software systems. This is especially true if there is no software layer translating the logical communication requests into physical hardware specific commands [9]. The rationale for shared memory has been stated explicitly by [12]:

> *"The shared memory system hides the remote communication mechanism from the processes and allows complex structures to be passed by reference, simplifying distributed application programming. Moreover, data in a distributed shared memory can persist beyond the lifetime of any single transient process."*

> *"The message passing models force programmers to be conscious of data movement between processes at all times, since processes must explicitly use communication primitives and channels or ports. Also since data in the data-passing model is passed between multiple address spaces, it is difficult to pass complex data structures."*

However, while the shared memory model of intertask communication simplifies program design, it

is limited by the power of the (single) host CPU. Increasing the available processing power means either using more than one CPU, or upgrading the host processor.

Using more than one CPU can give greater computation throughput by the exploitation of parallelism. However, closely coupled multi-CPU systems suffer from problems of scalability related to the complexity of the communication hardware required by CPU to CPU, or other hard wired connection topologies. Thus more general communication strategies based on networking software are desirable. The complexity of such distributed computing systems stems from the potential heterogeneity of CPU's, and heterogeneity of network communication protocols.

Distributed shared memory (DSM) is a concept of shared memory applied to loosely coupled systems, where true shared memory cannot be supported [10]. It attempts to combine the programming advantages of using shared memory for intertask operations, with the advantages offered by having tasks able to run on specialized hosts. Put more pragmatically [14]:

> *"Heterogeneous distributed shared memory (HDSM) is useful for distributed and parallel applications to exploit resources available on multiple types of hosts at the same time".*

However, inherent problems associated with loosely coupled systems - architectural and communication heterogeneity, and complex considerations of data consistency [5][6][7], have been responsible for the lack of widely accepted implementations of distributed shared memory systems.

This paper presents a candidate implementation of distributed shared memory (DSM), which is designed for a network of heterogeneous computing nodes. The system described here was developed having in mind a particular class of industrial process control applications requiring extensive computational power, but involving only moderate interprocess communication. In particular, it has been used to implement a large software suite for real time water network monitoring and control, as explained in section 5.

The proposed DSM system is based on the use of the X11 Windows graphics standard. This offers the advantage of both portability and an integrated graphics environment for the development of graphics user interfaces.

2. X11 basics

X11 Windows is a network transparent, vendor independent graphics oriented operating environment [8]. It consists of two parts: an I/O server which controls the graphics hardware (display screen, keyboard and graphics pointing devices), and client application programs which require access to the visual display and pointer device, and/or keyboard. Clients use the display and other devices by sending message requests to the server and, when necessary, they receive replies back from it. The server also provides system and display information to clients, thus acting as a database.

This client/server relationship, which is based on an underlying message passing system, corresponds to the central server algorithm [12]. This is where global data is held by a central database task. This task, the server, controls client access to the global data by supplying data on request to clients, and it updates the global data when sent new data by clients. Thus global data and client data may be structured in totally different ways. Furthermore, in the algorithm, global data does not migrate to other servers.

Figure 1 illustrates the concept of the X11 client server arrangement. An X11 server provides high level access to display and keyboard, and globally available client and system data. Clients access the I/O devices and data by sending requests to the server via a suitable network communication protocol. Clients that are local to the X11 server normally use a local communications protocol.

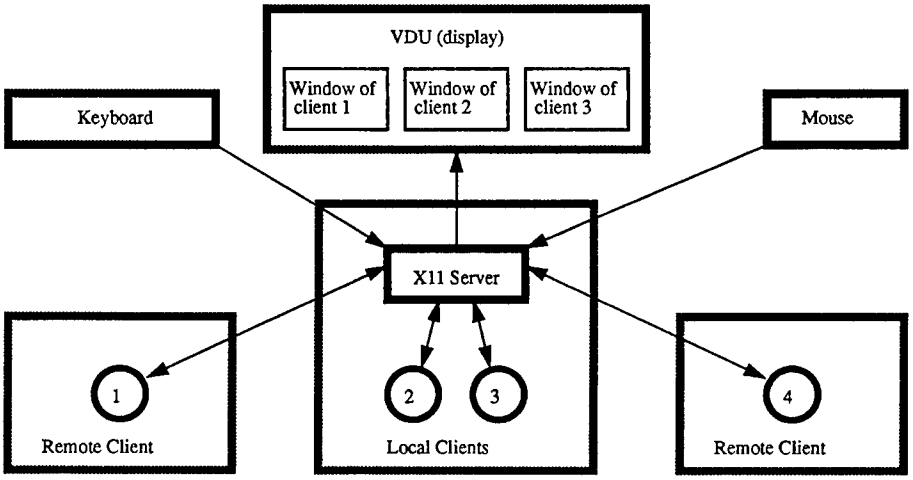


Figure 1. X11 client server arrangement

Access to the display and keyboard, together with the organization of the display, is provided by the windows structure. Clients are notified of any change to the windows by the server generating appropriate events. It is the responsibility of the client to monitor the event queue and to act accordingly. The communication of data between clients and a server is effected by means of properties, which are data structures associated with corresponding windows. The following gives a summary of the key concepts underlying the X11 Windows system:

- Windows

Windows are normally rectangular regions of the screen serving as output for client graphics and input from the keyboard. Windows are selected using a graphics pointer device, and they provide applications (clients) with a method of organizing the display screen. Each display screen consists of a single root window covering the entire screen. This serves as parent for overlying child windows. These child windows may also have child windows, and be overlying or underlying other windows. When a client connects (registers) with a server, it is assigned its own 'private' window within the server's display. The client may then create more child windows. This relationship between parent and child window creates a hierarchy of windows.

- Events

X11 clients frequently need to be aware of what is happening to any windows created for them, or about other windows they may have an interest in. Clients have two ways of obtaining information about the window environment. For immediate needs they can explicitly request specific window information from the server. For continual change notification they instruct the server to asynchronously send status change information. This status information is contained

in event data structures, which are queued on an input event queue.

Various window event structures exist to provide status information about window size and stacking changes, window property changes, pointer position and motion, client messages, etc. In addition, timer and I/O event facilities are defined by the standard X11 toolkit. X11 clients are in effect event driven, because they are expected to continually monitor incoming window and I/O events, and then react to them.

- Properties

Properties are data structures maintained by the X11 server and referenced by display and window id. If the window is destroyed the data will be destroyed by the server. The properties are structured as 1, 2, or 4 byte arrays permitting the storage of text strings and integers (which can be mapped to ASCII floating point values). Properties are referenced by host display, window, and property id. Thus a property stored in any window may be accessed by any client connected to the window's server. So properties represent globally sharable data.

Properties can be owned exclusively by a client task; this can provide the basis for an access locking protocol. A client wishing to gain privileged access to the property determines if the property is unowned, and then sets the ownership of the property to the window of its choice.

The client-server communication in X11 is based on a message passing system which is accessed via a Remote Procedure Call (RPC) interface. This means that the underlying detail of physical message routing is hidden from the user [4][10]. The mechanism used to communicate graphics function requests between client applications and the graphics server can be adapted to communicate appropriately structured, shareable properties. It is this mechanism which is utilized to support the facility of the distributed shared memory.

3. X11 based Distributed Shared Memory (XDSM)

A fundamental development for the distributed shared memory system is the definition of shared data areas and their mapping onto the physical memory resources. In our system, the local memory of the client is treated as addressed data blocks to be stored in global, shared X11 properties. This principal characteristic of our system is reflected in its name: the X11-based Distributed Shared Memory system - XDSM.

XDSM is constructed on the basis of library modules. Various modules exist for creating GUI interfaces, user language code interfacing, and GUI message generation. Figure 2 illustrates the logical information flows between the X11 server and main XDSM modules. At the center of an X11 client there is an endless loop checking the input event queue for incoming information. Events are extracted and processed by XDSM library code on the basis of whether they may be intended for the X11 GUI toolkit, or are control area property change events required by the control module. Reception of relevant property change events causes the configuration and control module to instruct the data access module to copy XDSM control data to a local copy of the control area.

- Shared data access module.

The most important module for the XDSM user is the module dealing with shared data access. This operates the various XDSM access protocols needed when reading and writing shared data, and transfers data directly between local memory, accessible to application code, and the XDSM maintained by the X11 server.

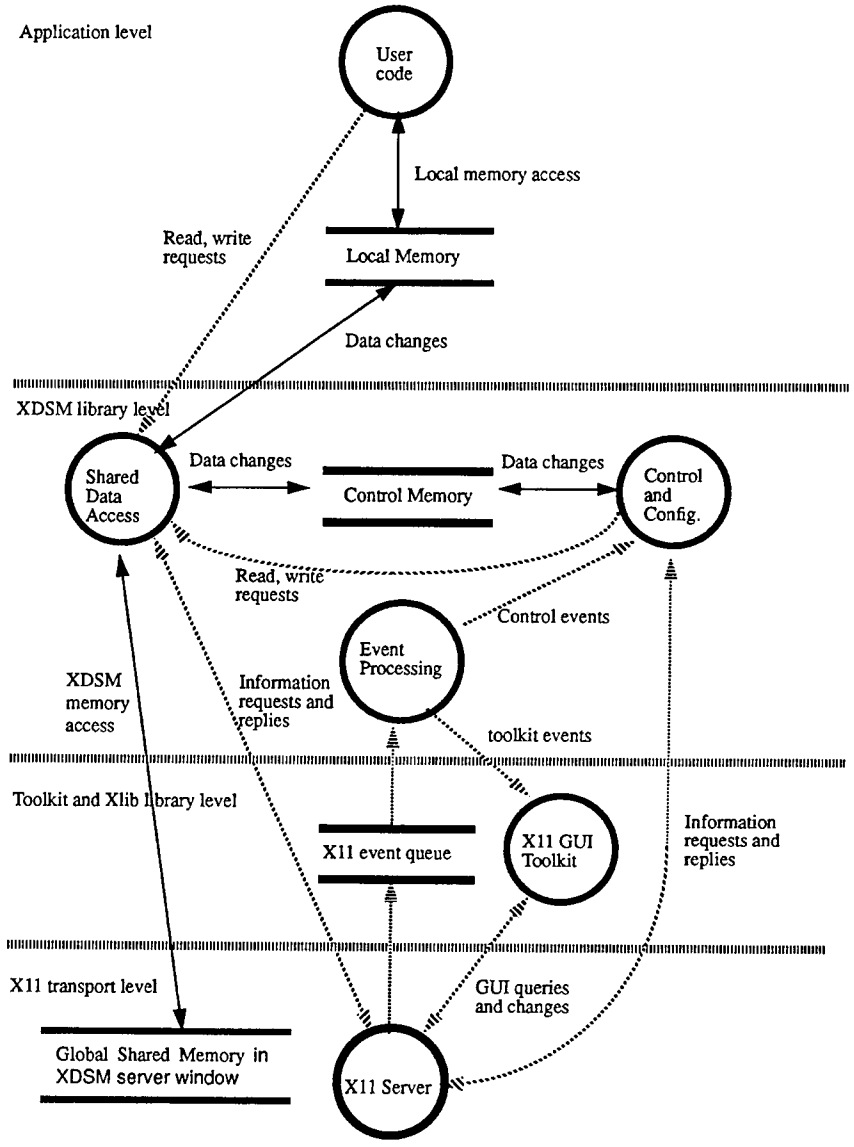


Figure 2. X11 and XDSM event processing

By using a property access locking protocol (explained on in section 4), X11 is used to supply all required memory access operations - lock, free, read shared area(s) to local memory(s), and save local area(s) to shared area(s) - in both access blocking and non-blocking forms.

• Event processing module.
The event processing module reads the queue of events sent by the server, and directs appropriate events to the X11 toolkit and to the XDSM control module. The X11 server, user code and the XDSM library tasks, execute in a pseudo-parallel manner. The event processing module implements in effect the scheduling of these tasks.

While the event processing loop is an integral part of any X11 application, for the purpose of the distributed shared memory system it is seen as an implementation detail. For this reason, the event processing module is embedded within the XDSM system and is transparent to the distributed applications.

• Control and configuration module.
The control and configuration module is responsible for maintaining the integrity of the distributed processing system. The module maintains a control data area which contains task pathnames, status and command information. The maintenance of the control data in XDSM systems is linked to X11 event processing and is hidden from the application programmer.

To prevent unnecessary re-reading of the control data, the XDSM system uses the facility by which X11 is instructed to inform an application of any modification to window properties. Changes to the control area are reported in the input event queue, and extracted from it by the control module. This passive approach to monitoring data consistency avoids the round trip delays incurred when requesting information from the X11 server - the message passing involved in a client→server→client information request requires a minimum delay of 2 task re-schedulings.

Regarding notification of property changes, there is evidence that occasionally events do not arrive, or can be seriously delayed. Therefore, using this passive communication protocol to support more complex XDSM page operations, corresponding to read, write, and delete faults, and dynamic data merging to give continual local/remote XDSM data consistency, is problematic. Implementing continual consistency checking and updating by data merging (see section 4 on Paging) would involve direct event processing by the data access module.

4. XDSM Implementation Issues

• XDSM configuration and start-up.
To provide for XDSM start-up and global control, an XDSM supervisor task was created. This provides a convenient window for XDSM data, and can be used to monitor the ownership status of each data area, periodically save the XDSM to disk, and provide various start-up and control options. However, its most important function is to start-up the XDSM system. To do this it reads a local configuration file which provides:

- A unique XDSM identifying name for the task suite to use.
- XDSM host names, and the operating system in use.
- The names of the local hosts together with the names of the local host task managers, and

which tasks are to run.

The supervisor (XDSM server) remotely executes each local host manager (this is operating system dependent, and may have to be done manually). The XDSM display and identifying name are supplied as command line parameters. After locating the XDSM control area and obtaining the client pathnames from it, the clients are activated on their associated hosts. The clients then put their status information in the control area. Should a client fail, the loss of its window can be detected by the manager so that it can be re-executed.

• Control area facilities.
To deal with task control and XDSM configuration, an XDSM control module and memory area has been provided (shown in figure 2). This contains lists of hosts, pathnames, status and command data for managers and clients. The command information relates to general commands issued at user instigation by the supervisor - start, stop, exit. The status information relates to a client's executing task status. By default, clients are marked as absent. When the client task is initiated it is marked as loading, rather than running. This is because task execution may fail, but the initiating manger will not be informed by the operating system. Also, there is an undeterminable delay before the client can locate and update the XDSM control area. Therefore, between the absent and the executing state of a task, there exists an intermediate state of loading. Once a client has connected with the XDSM, it marks itself as either running or waiting, as appropriate.

• Mutual exclusion during XDSM data access.
When requiring sole access to shared data, a client must determine if the property is unowned. When it is, it must set ownership of the property to its own window in the display where the required XDSM data is located. However, because X11 is asynchronous, with variable, buffered message delays, it is possible for applications in contention over data ownership to incorrectly gain multiple ownership.

An attempt to provide mutual exclusion by implementing a server grabbing protocol, which involves a client gaining exclusive access to the X11 server, is also deemed to be unsatisfactory as it can lead to the starvation of individual clients. The current version of XDSM implements a separate mutual exclusion protocol which is based on the use of a standard non-distributed mutual-exclusion algorithm. The property based protocol is a variant of Lamport's Bakery Algorithm [2] [11]. It uses property data to provide an abstraction of flat memory data structures (queue position numbers). The protocol causes the client at the head of the queue to wait in the queue until the data is free. It is implemented by the following stages:

1. Assign queuing number (highest existing number+1) and enter the queue.
2. Wait until there are no smaller numbers in the queue.
3. Wait until data is free.
4. Lock data.
5. Delete queuing number - exit queue.

The queuing system inherent in the Lamport Algorithm resolves this potential liveness problem for the client tasks. This is because after a process enters the queue, it will eventually move to the head of the queue, whereupon it can gain exclusive access to shared memory data.

• XDSM consistency and access efficiency.
Properties are also used to provide memory consistency checking. Each data property is associated with the 'last accessed' property. Each process that accesses the data sets the property ownership to its window. Whenever the area is to be read, this is checked to see if the data in it has been modified by another application. By making it unnecessary to re-read unchanged data, the XDSM system increases the efficiency of the shared data access.

• Data paging.
Because of restrictions in the way X11 properties can be used, modifying one item of shared data requires that all the data be rewritten. This implies that if the size of the properties are big, there is a large communication overhead associated with the update. XDSM overcomes the problem by automatically segmenting the data into smaller 'pages'. Storing data as pages has the advantage of reducing the amount of data to be read in and out, and by using the 'last accessed' property mentioned above, unnecessary reading can be avoided entirely. If the task holds a shadow copy of the data originally read from the XDSM, this can be compared with local memory data, and only modified pages saved back to shared memory. Merging data changes made both in local memory and XDSM, followed by updating the XDSM, is also possible.

5. Application
Using the XDSM, the following functionality can be provided for an application suite:

1. Definition of shared data areas - with user monitoring and control of the shared data provided by an interactive GUI graphics user interface.
2. Shared or exclusive access to global data.
3. Task coordination and control, including automatic task start-up and local task control.
4. Distributed error handling and recovery - achieved by maintaining shadow copies of the shared data, and by error handlers.

XDSM has been used to facilitate the distributed implementation of a realistic decision support system for the monitoring and control of water distribution networks. The structure of the application software suite is shown in figure 3. The suite comprises network and telemetry simulators, state estimators and an operator interface, configured to provide a classical feedback control loop. Other modules, which are scheduled for future incorporation into the suite, are concerned with optimal control and telemetry confidence limit analysis.

The original application suite was implemented as a set of concurrent tasks communicating by conventional shared memory. The parallel program implementation of this system is a natural extension of the original single processor multitasking implementation [3] [1]. XDSM's main addition to this system, though not involving any change to the original subsystem concept, is the XDSM supervisor task, together with local host task managers.

Using a network of 4 Sun SC Sparcstations running the simulation, telemetry, estimation, and operator interface modules respectively, the cycle time achieved for a 65 node network was approximately 10 secs. This is at least an order of magnitude better than would be expected in real life. Projecting the results for larger networks, it is expected that the communications overhead will, at worst, increase linearly with network size, so the communication to computation time ratio will actually decrease.

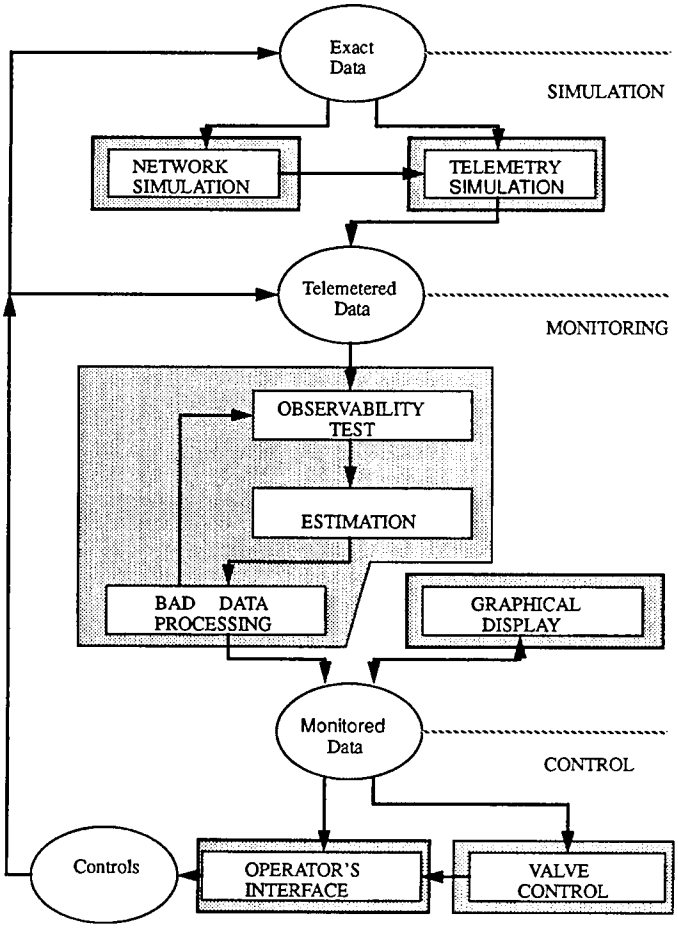


Figure 3. Water network monitoring suite

6. Conclusions

Our work to date has resulted in the development of an original software product, an X11 based Distributed Shared Memory (XDSM) system, intended for applications in scalable heterogeneous distributed computing environments.

The XDSM system has been successfully used to provide an implementation framework for a telemetry system concerned with the monitoring and control of water distribution networks. The performance of the current implementation of the XDSM indicates its applicability to such a class of industrial process control applications.

As major performance differences relating to various mutual exclusion algorithms have been noted, the next stage of our research will be concerned with performance evaluation. This will be done in conjunction with the development of an orthogonal programming meta-language harness in-order to support the implementation of the Bulk Synchronous Parallel (BSP) model of parallel processing [13].

7. References

[1] Argile.A. & Bargiela.A., Using X11 windows to provide shared task-memory in distributed systems, in Integrated Computer Applications in Water Supply, Volume 1, Coulbeck.B.(ed.), Research Studies Press, 1993.

[2] Ben-Ari.M., Principles of Concurrent and Distributed Programming, Prentice Hall, 1990.

[3] Bargiela.A. & Al-Dabass.D., A Simulated real-time environment for verification of advanced water network control algorithms, Systems Science 14.3. 1988.

[4] Coulouris.G.F. & Dollimore.J., Distributed Systems, Concepts and Design, Addison-Wesley, 1988.

[5] Heddaya.A., & Sinha.H,
An implementation of MERMERA: A Shared Memory System that Mixes Coherence with Non-Coherence, BU-CS-92-013, 1992.

[6] Heddaya.A., & Sinha.H,
An overview of MERMERA: A System and Formalism for Non-coherent Distributed Parallel memory, BU-CS-92-009, 1993. To be published in Proc. 26th Hawaii Int. Conf. Sys. Sci.

[7] Hutto.P.W. & Ahmand.M., Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories, IEEE 10th Int. Conf. Dist. Systems, 1990, 302-307.

[8] Jones.O., Introduction To The X Window System, Prentice Hall, 1989.

[9] Kranz.D., Johnson.K., Agarwal.A., Kubiatowicz.J., Beng-Hong.L., Integrating Message Passing and Shared Memory: Early Experience, SIGPLAN Notices, 1993 28(7), 54-63.

[10] Levelt.W.G, Kaashoek.F., Bal.H.E., Tanenbaum.A.S., A Comparison of Two Paradigms for Distributed Shared Memory, Software-Practice and Experience, 1992, 22(11), 985-1010.

[11] Raynal.M., Algorithms for Mutual Exclusion, North Oxford Academic, 1986.

[12] Stumm.M. & Zhou.S., Algorithms Implementing Distributed Shared Memory, COMPUTER, 23, 5, 1990.

[13] Valiant.L.G., A Bridging Model for Parallel Computation, Comm.ACM, 1990,33.8, 103-111.

[14] Zhou.S, Stumm.M. & Wortman.D., Heterogeneous Shared Memory, IEEE Trans. on Parallel and Distributed Systems, Vol. 3, 5, 1992.

CCSP -A Formal System for Distributed Program Debugging

BETH ARROWSMITH and BRUCE McMILLIN[1]
Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401 USA
e-mail: ff@cs.umr.edu

Keywords

Parallel Programming, Distributed Programming, Formal Methods, Run-Time Satisfaction, Executable Assertions

1 Abstract

One of the major problems with programming in a parallel/distributed environment is the difficulty in debugging the programs due to their complex interactions. One can have results that appear random, but when given a complete knowledge of the specific run-time behavior, the results are foregone. Unfortunately, this complete knowledge is not generally attainable in a distributed system. In order to develop a system for debugging distributed programs and, for the more general case, ensuring their correctness at run-time, we built a distributed execution environment based on Hoare's CSP [5] which allows for the execution and evaluation of embedded assertions within the CSP program.

2 Introduction

In sequential programming, debugging consumes the most time[4] and is frequently the most difficult part of a program's creation. Now that distributed and parallel programs are commonplace, particularly among scientific computing and control applications, debugging has become even more difficult. The debugging of a program consists of three phases: fault detection, fault location, and fault repair [2]. The complexity of each of these three stages is greatly magnified when applied to distributed, parallel programs. For example, in order to repair a fault, consideration must be given to such possible side effects as deadlocks and data races, irrelevant concepts in a purely sequential program. Fault location is also more difficult. Any given fault could be caused by an interaction of multiple processes rather than the malfunction of a single process. The program's natural nondeterminism complicates fault detection. Differing scheduling schemes can cause successive runs of the program with the same input to result in differing outputs.

[1]Supported in part by the National Science Foundation under Grant Numbers MSS-9216479 and CDA-9222827, and, in part, from the Air Force Office of Scientific Research under contract number F49620-92-J-0546 and F49620-93-1-0409, and, in part by a grant from the University of Missouri Research Board.

Based on Hoare's theoretical language CSP[5], Ada is relatively popular for debugging research. This research has produced debugging methods for Ada programs to help with fault location and detection. However, these methods are mainly for pre-production debugging and require deterministic program runs[1] [11]. Moreover, these methods can only ensure that programs do not "break" on previously tested execution schedules.

However, debugging only attempts to ensure a program's correctness after the program is written. This is difficult to check. Verification is required while coding the program. Formal methods offer such verification. If a programmer could perform a complete and correct program-proof or verification using formal methods, debugging would be unnecessary. Unfortunately, no programmer is perfect. A program is only as correct as its worst error. This includes the verification of the program. Therefore, a combination of formal methods and debugging is necessary to achieve confidence in both the correctness of a program-proof and the reliability of the program, itself [4]. ANNA, an offshoot language of Ada, contains facilities for executable assertions, but is not written for the distributed environment. Nor is ANNA considered sufficient for specification or error discovery and recovery[9]. Ergo, these requirements must be met by other means, in this instance by returning to Ada's multitasking roots, CSP.

3 CCSP Syntax

CSP, having a rich background in formal methods theory, particularly in the use of mathematical and logical assertions[7] [8], has proven to be a powerful tool for reasoning about program structure[14]. In order to take advantage of the power of CSP, a parser was created. This parser, called CCSP, creates a C process from a CSP process and allows CCSP to be run on any BSD UNIX network.

3.1 Communication

CSP differs significantly from many computer languages. For example, a considerable portion of its grammar concerns communication. In fact, the only means a process has to affect another process is through interprocess communication. No shared memory or variables are permitted. For instance, if a traffic light process needed a car process to know that the light was green, a message would be sent from the traffic light process to the car process as follows.

Example 3.1 *Process traffic_light::*
[*car* ! *light_type(green)*]

In the above instruction, "car" is the name of the car process, "green" is the message, and "!" indicates a send operation. In order to receive this message, the car process would use the following receive statement:

Example 3.2 *Process car::*
[*color light;*
traffic_light ? *light_type(light)*]

In the receive instruction, "traffic_light" is the sending process, "light"is a variable in which the color of the light is received, and "?" indicates a receiving operation.

In contrast to CSP, since CCSP is a real language, communication is not this simple. At the beginning of a CCSP program, a process name is explicitly paired with an address, allowing a

process to be referenced by its process name. The message, however, is a bit more complex. In order for a message to be sent or received, the size of the message must be known in addition to the type and contents of the message. So the traffic light becomes:

Example 3.3 *Process traffic_light::*

```
[
mach1@car!light_type,green,light_size
]
```

Process car::

```
[ color light;
mach2@traffic_light?light_type,light,light_size
]
```

In the new receive instruction, "`light_type`" is the type of message, "`light_size`" is the size of the message, "mach1" and "mach2" are the two machine names given as numerical Internet addresses or names, and "`car`" and "`traffic_light`" are the process numbers (as in 2001). In this example, the car process waits for the "`traffic_light`" process to send a message, and the "`traffic_light`" process waits for the car process to receive its message.

4 Assertions in CCSP

Logical satisfaction of assertions within a formal methods approach to debugging is embedded in CCSP as much as communication. Logical satisfaction is implemented by checking mathematically stated preconditions and postconditions (assertions) for correctness at runtime. For example, suppose that five philosophers share a two room apartment. One room is a dining room and one room is a thinking room. Alas, the philosophers are poor. Their dining room has a table with five chairs and five plates, but they only possess five forks: one to the left and one to the right of every plate. (Forks are shared.) In order to eat the spaghetti each philosopher needs two forks. If a philosopher cannot get two forks, she will starve. Each philosopher picks up the fork to her left first and then the fork to her right. Should she be unable to get the fork immediately, she can get the fork as soon as another philosopher releases it. As long as one philosopher can eat, deadlock will not occur. The eating philosopher will eventually put down both forks allowing others to eat. However, if all the philosophers are at the table and none can eat, then deadlock has occurred, and the philosophers will starve. The most famous solution and the solution implemented is to only permit four philosophers in the eating room at one time. Thus, at all times, this solution allows at least one philosopher to eat, preventing deadlock. The philosopher and fork processes are straightforward and are depicted in Example 4.1 for Philosopher 1 and Fork 1 and the room process in Example 5.1.

Example 4.1

Philosopher 1

```
*[
Thinking;
assert(I and value(in1)==false);
room!enter,&come_in,4;
assert(I and value(in1)==true);
printf("Phili is in the room\n");
fork1!pickup,nullbuf,0;
assert(I and value(in1)==true);
fork2!pickup,nullbuf,0;
assert(I and value(in1)==true);
Eating;
fork1!putdown,nullbuf,0;
assert(I and value(in1)==true);
fork2!putdown,nullbuf,0;
room!eggsit,&go_out,4;
assert(I and value(in1)==false);
] ]
```

Fork 1

```
*[
*[ /* Philosopher to the right */
  []if (Phil1?pickup,nullbuf) ->
      printf("Phil1 picks up Forki");
      assert(I and value(in1)==true);
      Phili?putdown,nullbuf;
      printf("Phil1 puts down Fork1");
      assert(I and value(in1)==true);

  /* Philosopher to the left */
  []if (Phil2?pickup,nullbuf) ->
      printf("Phil2 picks up Fork1");
      assert(I and value(in2)==true);
      Phil2?putdown,nullbuf;
      printf("Phil2 puts down Fork1");
      assert(I and value(in2)==true);
] ]
```

To prevent deadlock, certain preconditions must be met. In particular, an invariant is needed to ensure that all of the philosophers are not in the dining room simultaneously.

Example 4.2 `#define I (not(value(in0) and value(in1) and value(in2) and value(in3) and value(in4)) and value(occupancy_copy)<=4)`

This invariant contains six global values: "in0" to "in4" and "occupancy_copy". The "ini"'s store the same numbered philosophers location: the dining room, 1, or the thinking room, 0. "occupancy_copy" contains the number of philosophers in the dining room. All dining philosopher processes as well as the fork and room processes depend on this invariant, I.

Additionally, a philosopher must verify subsequent to picking up or putting down a fork that she is in the eating room and that I, the invariant, is true. For Phil0, this assertion would be:

Example 4.3 `assert(I and value(in0) == TRUE)`

Assertions allow the specifications to be written into the program, improving the debugging of the code as well as pointing out weaknesses in a specification, both useful qualities for formal methods.

Global Variables In the philosopher example, the global variables "in0" to "in4" and "occupancy_copy" were used. However, CCSP is based on CSP, a completely distributed programming language whose design does not even permit memory to be shared. In order to allow global-like variables, each process must possess a local copy of the "global" variables. These variables can be set within a process using the equal statement. For instance, in the

room process (Example 5.1), "occupancy_copy" is set to zero by "equal(occupancy_copy, occupancy);". To retrieve this value, just use "value(occupancy_copy);". In addition, the "global" variable could be set during communication, either explicitly via receiving a communication into an equal function, or implicitly.

In an implicit "global" variable exchange, process A sends to process B (and vice versa) every global variable changed since the last communication with process B.

Example 4.4

```
                Process Room            Process Phil4
Before communication:
                Time: 23                Time:  20
in0             value: 1  time: 21      value:  0   time: 16
in1             value: 1  time: 17      value:  1   time: 17
in2             value: 1  time: 19      value:  1   time: 19
in3             value: 0  time: 23      value:  1   time: 18
in4             value: 0  time: 20      value:  0   time: 20
occupancy_copy  value: 3  time: 23      value:  4   time: 19

Communication:
(Phil4?enter,equal(in4,x))              room!enter,&come_in,4;
After communication:
                Time: 24                Time:  24
in0             value: 1  time: 21      value:  1   time: 21
in1             value: 1  time: 17      value:  1   time: 17
in2             value: 1  time: 19      value:  1   time: 19
in3             value: 0  time: 23      value:  0   time: 23
in4             value: 1  time: 24      value:  1   time: 24
occupancy_copy  value: 3  time: 23      value:  3   time: 23
```

Each communication in CSP is augmented in CCSP with two functions, Psi and Phi [8] which prepare copies of a processes' global auxiliary variables for communication and unions these variables into a process' local state, respectively. Thus, above, prior to a communication, Room and Phil4 prepare their copies of the global variables for transmission using the Psi function.

Formal Basis for Global Variable Maintenance (Psi and Phi) The Psi function ensures that all copies of the global variables sent are the most recent information and that only those variables changed since the last communication with the other process will be sent. In order to determine which "global" variables are most recent, it is necessary to timestamp them. However, there is no global clock for truly distributed processes. Agreement on time is difficult, but since only relative, not absolute, times are needed, Lamport clocks are sufficient[6]. Each process begins with a local Lamport clock set to 0. At each communication, the Lamport clock is changed to the maximal value from the two processes and then incremented by one[8]. This is demonstrated by the trace of Room and Phil4. The clock in Phil4 jumps from 20 to 24, the original time of the Room process plus 1, to make 24. The Room clock, however, merely increments by one, already having the maximal time in that communication. Additionally,

Psi must prepare to send all copies of global auxiliary variables it has heard so far from other processes, along with their timestamps.

Upon receiving Room's copy of the "global" variables, Phil4 stores them and then transmits its own copies of the variables to Room. After the global variables are sent, Phil4 updates its current back copies of the global variables based on Room's information using the Phi function[8].

The Phi function ensures that only the most recent copies of the global variables, whether from Phil4 or Room, are kept. For example, in Phil4, global variable in0, which has a value of 0, assigned at time 16, is changed to room's more current information to value 1, assigned at time 21. This spread of "new" information ensures that the logical assertions from a CSP specification are preserved in CCSP's run time environment. [2]

5 Debugging with CCSP

Traditional debuggers are nearly incapable of debugging a multiple process program. However, by using CCSP's `assert()` statement, debugging a distributed multiple process program, such as the Dining Philosophers, is relatively simple. A successful Dining Philosopher's solution is an endless round of philosophers thinking, picking up the forks, and eating.

Example 5.1

```
                                                    Phil4 is thinking
Phil0 is in the room
Fork0 From Phil0
          Phil1 is in the room
          Fork1 From Phil1
                     Phil2 is in the room
                     Fork2 From Phil2
                                Phil3 is in the room
                                Fork3 From Phil3
                                Fork4 From Phil3
                                Phil3 is eating
                                Fork3 To Phil3
                                Fork4 To Phil3
                                Phil3 is thinking
                     Fork3 From Phil2
                                              Phil4 is in room
```

A philosopher and a fork are straightforward processes to write. The philosopher, as can be seen in example 4.1, has no conditionally-reached states. One command merely follows another, sequentially. The only assertion necessary to ensure the absence of deadlock is the assertion discussed in examples 4.3 and 4.4.

A fork process is nearly as straightforward. In order to have a correctly functioning fork, two conditional statements are required, one to see if the philosopher on the right wants it (using a conditional, nonblocking receive statement) and one to see if the philosopher on the left wants it. Also, to ensure that only philosophers in the room can pick up or put down a

[2] "occupancy_copy" is 3 rather than 4 because the room process has not incremented "occupancy_copy" yet. The increment occurs after the communication.

fork, an additional assertion requiring the philosopher requesting the fork to be in the dining room is needed. Finally, in order to ensure that deadlock does not occur, each fork process has the same invariant as the philosopher process.

The most difficult process type, as is seen in Example 5.4 to write correctly is the room process.

Example 5.2 (Room)

```
*[ assert (I);
*[ []if ((Phili?enter,equal(ini,x)) AND occupancy < 4) ->
        assert(I and value(ini)==true and value(occupancy_copy) < 4);
        occupancy := occupancy + 1;
        equal(occupancy_copy,occupancy);
        assert(I and value(ini)==true and value(occupancy_copy) <= 4);
   [] if (Phili?eggsit,equal(ini,x)) ->
        assert(I and value(ini)==false  and value(occupancy_copy) <= 4);
        occupancy := occupancy - 1;
        equal(occupancy_copy,occupancy);
        assert(I and value(ini) == false  and value(occupancy_copy) < 4);
/*The other philosophers are symmetrical.*/
] ]
```

The room monitors the philosopher processes. It is due to the room that deadlock does or does not occur. The room process needs to ensure three things:

1. the invariant, presented in the philosopher and the fork processes, holds,
2. the relevant philosopher is in the room, and
3. the number of philosophers is less than or equal to four.

In order to maintain these assertions, the room process has two conditional statements for each philosopher: one to exit and one to enter. To exit correctly, the philosopher sends a message to the room that she intends to leave. The global variable representing this philosopher (ini, where i is the philosopher number), is changed to reflect this fact. The more difficult problem occurs when the philosopher enters, for at this point in time, two criteria must be met

1. the philosopher is ready to enter and
2. fewer than four philosophers are present.

It is necessary to make sure that fewer than four philosophers are in the room before adding another philosopher, as deadlock occurs if all five philosophers are in the room. For a philosopher i, this code looks like:

Example 5.3 `if ((Phili?enter,equal(ini, x)) AND (occupancy < 4)) ->`

However, this code fails spectacularly. In fact, it often does not even last one round of philosophers eating.

Example 5.4 (Output of Faulty program)

```
Phil0 is thinking
          Phil1 is thinking
                    Phil2 is thinking
                              Phil3 is thinking
                                        Phil4 is thinking
Phil0 is in the room
          Phil1 is in the room
                    Phil2 is in the room
                              Phil3 is in the room
Assertion failed: file "room.c", line 47
Assertion failed: file "Phil4.c", line 53
```

Tracing the failure in the C programs, the first assertion which failed, the assertion on line 47, of room.c which is: "assert(I);", where I is as previously defined. The next failed assertion is on line 53 in the Phil4.c program and is as follows:

```
assert(I&&value(g[0],in4) == true);
```

This assertion merely states that the fourth philosopher is in the dining room and that less than five philosophers are present in the dining room. A careful examination of the invariant indicates that all five philosophers were in the room; the room's occupancy was more than four. The program's output indicates that four philosophers were in the room when philosopher 4 tried to enter the room. Somehow, philosopher 4 entered the room when she should not have. In order to enter the room, the program requires philosopher 4 to query for permission to enter the room. Then the algorithm checks if the number of occupants is less than four. The above philosopher code only requires permission from the room to enter. Ergo, the above guard statement is out of order and the order should be changed to:

Example 5.5 `if ((occupancy < 4) AND (Phili?enter, equal(ini, x))) ->`

When this modified code is tried, it works perfectly. An eternal round of philosophers thinking and eating occurs.

5.1 CCSP on a Non-Trivial Problem

Flexible enough to be used for research programming, CCSP is currently used in the development of a dynamic system for machine identification in a network during a startup program called DYMATAC, (DYnamic MAchine TAble Configuration). Currently the normal procedure is for a system administrator to create a machine table before the network is started. This table contains the necessary information, particularly the machine names and addresses of machines available for communication. The DYMATAC implementation is nearly completed, despite with only two design errors found during the coding process as a result of using a formal development approach.[10]

6 Extension of CCSP to Evaluate Temporal Logic Specifications

CCSP has also been expanded to operationally evaluate specifications written in Interval Temporal Logic [12]. Operational evaluation of temporal specifications, not unlike axiomatic specifications, consists of two parts:

1. collection of events and
2. evaluation of specifications on these events.

Evaluation of axiomatic assertions involving global auxiliary variables can be viewed as a special case of the general problem of collecting event *histories* in which the collection of global auxiliary variable copies is actually the collection of an event history of size 1. Collection of longer event histories requires that the events be timestamped by a vector clock scheme [3] such that causally-related events can be ordered into a collected *equivalence class* of histories.

Once these histories are collected, interval formulae and responsiveness assertions which denote timing behavior of the system can be operationally evaluated. For example, let p, q, α be interval formulae. The interval formula $[p, q]\alpha$ asserts that α must be true within the interval $[p, q]$. Operationally, this requires a decision procedure more powerful than *assert*. Ergo, the recently developed Π [13] evaluates an interval formula against a collected event history which returns true or false if the assertion is satisfied by the collected history.

7 Conclusion

CCSP has proven invaluable for the creation of correct distributed programs. Not only does it provide an environment compatible with distributed programming, but with the rapid dissemination of global variables, the accuracy of the program run-time predictions can be determined. With CCSP, not only the code of the distributed program has been corrected, but also some of the handwritten verification proofs of programs, much more readily than could have otherwise been expected. Moreover, CCSP has proven flexible enough for expansion into temporal logic and powerful enough to handle non-trivial programs like DYMATAC.

References

[1] R.N. Brindle, A.F. Taylor and D.F. Martin. A debugger for Ada tasking. *IEEE Transactions on Software Engineering*, 15:293 304, 1989.

[2] M. B. Feldman and M. L. Moran. Validating a demonstration tools for graphics-assisted debugging of ada concurrent programs. *IEEE Transactions on Software Engineering*, 15:305 313, 1989.

[3] J. Fidge. Timestamps in message passing systems that preserves the partial ordering. *Proceedings of the Tenth International Conference of Software Engineering*, pages 182 187, 1993.

[4] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[5] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[6] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[7] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[8] H. Lutfiyya, M. Schollmeyer, and B. McMillin. Formal generation of executable assertions for application-oriented fault tolerance. *UMR Department of Computer Science Technical Report Number CSC 92-15*, 1992.

[9] S. Sankar and M. Mandal. Concurrent runtime monitoring of formal specified programs. *Computer*, 26(3):32–41, 1993.

[10] A. Su. Dynamic machine table configuration algorithm in distributed systems. Master's thesis, University of MO - Rolla, Department of Computer Science, 1994.

[11] R.H. Tai, K.C. Carver and E.E. Obaid. Deterministic execution debugging of concurrent ada programs. *IEEE The Thirteenth Annual International Computer Software and Applications Conference (COMPSAC89)*, pages 102–109, 1989.

[12] G. Tsai, M. Insall, and McMillin B. Constructing an interval temporal logic for real-time systems. Technical Report 93-25, University of MO - Rolla, Department of Computer Science, 1993.

[13] G. Tsai, M. Insall, and McMillin B. A run-time decision procedure for responsive computing. Technical Report 93-29, University of MO - Rolla, Department of Computer Science, 1993.

[14] Zhiwei Xu. Structured principles for developing parallel computing programs. *IEEE International Conference on Systems, Man, and Cybernetics*, pages 655–660, 1991.

A Friendly Data Flow Programming Environment

Miguel Menasche[1] Denis Maciel Maia[2]

[1] Professor at the Departamento de Engenharia Elétrica
Pontifícia Universidade Católica do Rio - PUC/Rio
Rua Marquês de S. Vicente 225, CEP 22453-900
Rio de Janeiro, Brazil.
E-Mail: Bitnet - menasche@brlncc.bitnet ; Internet - menasche@ele.puc-rio.br

[2] Student of Computer Engineering at PUC-Rio, Brazil.
E-Mail: Internet - caec@gsc.ele.puc-rio.br

Summary

This work describes the **dfEnv** project that is being held at the Electrical Department of PUC/Rio - Brazil. The **dfEnv** project is a friendly data flow programming environment consisting of a data flow processor card, a data flow assembler, a graphics editor, a simulator and debugger and a data flow high level language. In a short-run planning, the aim of this project is to gain familiarity with a different processing philosophy that highly exploits parallelism, and in the long run, to study its behavior in real situations in the image processing field.

Keywords Data Flow, programming environments, graphics editors, simulators, debuggers.

1. Introduction

The continuous search for better performances in computing systems paved the way to three distinct and self completed dimensions: smaller switching times due to technical developments in electronic components; new internal organization of components in order to optimize information flows and alternative models of computations that highly exploits parallelism.

Among the alternative models of architecture proposed, one particularly strongly departs from the conventional models, based on Babbage proposals by the last century. The data flow model proposed by Jack Dennis in the 70's, compels to a different way of thinking. In this model, logical and arithmetic operations will no longer be executed following a previous order specified by the programmer (control flow), but instead following a non deterministic order defined by the available data (data flow). As soon as data are generated, they enable the execution of all operations dependent on them. Execution of these operations may be simultaneous or in any order depending only on the available processing units free at that moment. Therefore, this type of machine does no longer needs a register pointing to the instruction that will be executed (the program counter). The program concept itself must also be reviewed. In a data flow machine a program is no longer provided under the form of an instruction list that will be executed sequentially, but instead, under the form of an oriented graph where the relations of dependencies or precedences in the execution of instructions must be explicit. Two operations that are not specified in a precedence relation may be executed in parallel or in any order. The following graph illustrates a data flow program used to compute **X**, one of the solutions for a second degree equation.

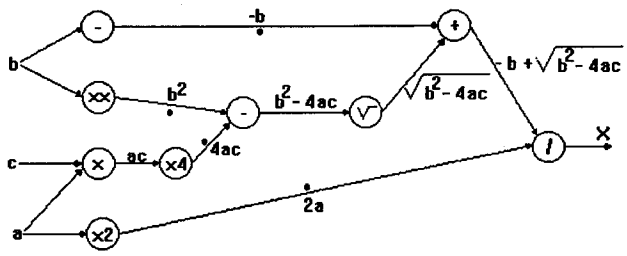


Figure 1 A Data Flow Graph

Available data in a precise instant of time are represented by small dots in the proximity of their corresponding arcs, called **tokens**. Thus, the figure above shows not only the flow graph representing the precedence relations to obtain **X**, but also the instant at which the values **-b**, $\mathbf{b^2}$, **4ac** and **2a** had already been calculated. An operation can only be started when all arcs that arrive in the nodes that correspond to this operation contain their respective **tokens**.

Until few years ago, prototypes of data flow machines were found only at research centers, such as the MIT or the Manchester University. It was only in 1984 that a commercial product, the NEC μPd 7281 microprocessor was issued.

Although this first commercial product dates from 1984, enabling the low cost construction of data flow machines, its use has been far short of the raised expectations, probably due to the difficulties of adopting such a new computational paradigm.

Therefore, having in mind to increase the familiarity with this new programming philosophy, the Electrical Engineering Department of PUC/Rio started the project of a friendly data flow programming environment (**dfEnv**) that comprises a processor card (**dfP**) based on NEC's microprocessor μPd 7281 to be used in a PC compatible host computer, a data flow assembler (**dfA**), a graphics editor (**dfG**) for data flow graphs, a simulator and debugger (**dfSim**) and a high level data flow language (**dfL**). [1]

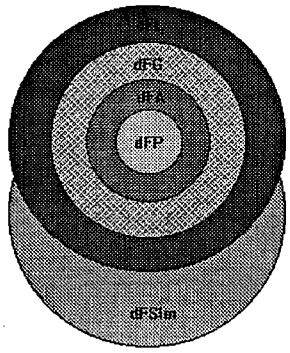


Figure 2 The dfEnv Environment

The principal objective of this project is to obtain a data flow platform where it would be possible to gain experience on the development of data flow programs in order to compare parallel and sequential algorithms in the image processing fields.

2. The Processor Card dfP

Since the **dfP** should be the physical platform above which the **dfSim** would be constructed, it was first decided to attach it to a host computer in order to avoid designing a whole data flow computer with operating system, I/O and everything else that would demand us more design and construction time, and potentially could bring us supplementary problems unrelated to the parallelism itself. NEC's microprocessor, μPd 7281, foreseeing its utilization jointly with a host CPU, strongly contributed for this decision. The great popularity of the PC-compatible microcomputer at our university labs, led to its choice as the host computer adopted.

In order to better understand this processor card and the proposed environment, an explanation of the processor μPd 7281 follows in the sequel.

2.a. The processor μPd 7281 [2]

The processor μPd 7281, also denominated **Image Pipelined Processor**, is a VLSI NMOS device, embedded into a 40 pins chip, fed by a single 5 V power supply and clocked at 10 MHz. It was projected to be utilized jointly with a host CPU and to be connected with up to 14 other processors in cascade. Each one of the cascaded processors must receive an identification or module number provided throughout its input bus, immediately after initial RESET signal; the host CPU identification is always 0.

Each μPd 7281 has one circular pipeline, where tokens circulate, and a single processing unit. The following figure shows a block diagram of a μPd 7281 microprocessor:

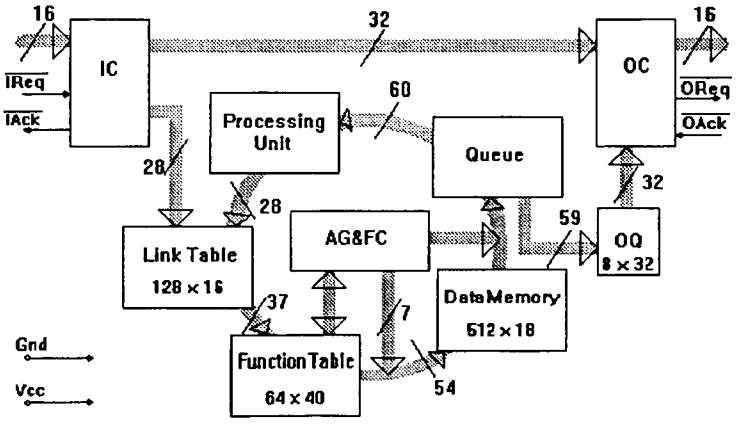


Figure 3 Block Diagram of a μPD7281

32 bits tokens arrive to a μPd 7281, through its 16 bit input bus, then are converted by the input controller **IC** and sent to the internal pipeline or to the output controller **OC**, according to their destination: the current processor or another processor, respectively.

The block diagram of processor μPd 7281 makes visible the circular pipeline composed by five blocks: three of them constituted by memory devices; among these, two are under the format of a table (The **Link Table** and **Function Table**), and are useful to internally store the flow graph of a program which will be performed by the μPd 7281. The **Link Table** is responsible for storing the arcs of the flow graph and the **Function Table** for storing the nodes of the same graph. The third memory device is the **Data Memory**, which is responsible for storing constants and temporary data that corresponds to available tokens waiting for the generation of their companion tokens necessaries to the execution of an operation. When all the necessary tokens to the execution of an operation are available, all these informations are packed into a single token and stored in the **Queue** which is a queue of tokens waiting for a processing unit. Tokens in the **Queue** are sent either to the processing unit of the same processor or to the output queue **OQ** in order to be directed to other processors or to the output through the output controller **OC**. The last element of the circular pipeline is the **Processing Unit**, which is responsible for the execution of operations and for the generation of a result token that will be sent back again to the **Link Table** in order to continue the computation.

Throughout the pipeline, tokens are transformed, packed and unpacked, assuming different forms and sizes according to the stage in which they had arrived. Due to the pipeline, up to five instructions can be internally processed in parallel at a rate of up to 5 MIPS.

In 1990, a successor of the μPd 7281, the μPd 72181, pin to pin compatible with it, but with the double of its speed and performance appeared.

2.b. The dfP card design

A first requirement established for the project of the **dfP** was that the card should be modular, allowing the addition of more processors if necessary, using the inherent capability of a μPd 7281 to be connected with others in cascade. Thus, the card disposes of four 40-pins sockets, enabling the insertion of up to 4 μPd 7281.

The concept of tokens outside a μPd 7281 is unknown, as well as the concept of memory addresses inside a μPd 7281. Thus, it is necessary to have some interface to perform these conversions. An integrated circuit, called **Memory Address Generator and Interface Controller - MAGIC, the μPd 9305**, was therefore provided by NEC to be in charge of this task.

The first prototype of the processor card **dfP** contains 1 μPd 7281 expandable up to 4 processors, a μPd 9305 and a private static RAM memory of 128K x 16 bits, as shown by the following block diagram.

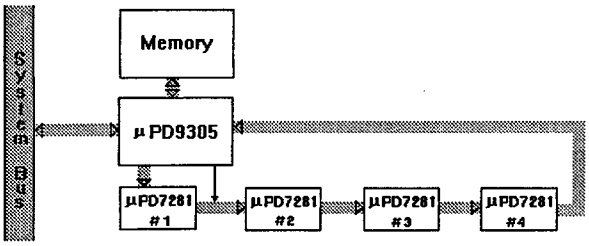


Figure 4 Block Diagram of the **dfP**

The μPd 9305 is responsible for loading, into memory, files with the programs generated by the host computer. Afterwards, the μPd 9305 transforms these programs into a queue of tokens and sends them, for execution, to the processors connected in cascade.

It is possible to increase the maximum performance of this card by inserting more processors in cascade.

Depending on the applications however, bottlenecks may appear, mainly in the communication with the bus system. Nevertheless, for the proposed goals, the aforementioned scheme is useful enough for a prototype.

3. The Assembler dfA

The first step to a programming environment is the availability of an assembler that could translate mnemonic code, generated by a programmer, into machine language.

Despite the knowledge that NEC had already developed an Assembler for their processor, it was somewhat difficult to obtain it in the Brazilian market. Thus, we decided to construct our own version. This assembler was built in Turbo Pascal, following the specifications provided by NEC, whenever possible.

An assembler is only a direct converter of code that translates keywords of a text file into their respective binary machine code. Furthermore, an assembler must also solve a series of address references used by several program variables. Keywords used for programming a data flow machine are those necessary to the description of a flow graph: their nodes and arcs.

The development of an assembler for a data flow machine in general, and for the μPd 7281 in particular, presents specific details not found in the case of traditional machines. In a data flow machine, for instance, the assembler needs to store, under some form, the flow graph. In the μPd 7281, this is done by initialization of the internal tables **Link Table** and **Function Table**.

Changes produced to the **Link Table** and to the **Function Table**, in particular their initializations, must be done through μPd 7281 instructions. This means that the execution of an initialization program must always occur previously to the execution of the program itself that is being assembled. Consequently, the assembler **dfA** besides to performing the code conversion from assembly to the machine language, must also be the responsible for the automatic generation of this initialization program, in machine language too.

A data flow program (flow graph) must also be distributed between all the available processors since the **dfP** card may have several of them. The **dfA**, therefore, has also the task of loading, into the respective processor's internal tables, the specific pieces of the flow graph that must be executed by each one.

Among the supplementary characteristics presented in the **dfA** assembler there is also the optional possibility of a text file generation containing a map of the contents and free space of the existent tables in the three internal memories, (**Link Table, Function Table** and **Data Memory**), as well as a map of the used variables.

4. The Graphics Editor dfG

The low level programming in a data flow computer consists therefore in the conception of a flow graph that must be translated and understood by the machine in order to determine the several allowed possibilities of parallelism.

In the case of the μPd 7281, this flow graph is originally manually converted to a text description of nodes and arcs that constitute the assembly language. Only then the assembler **dfA** converts this text description to a binary sequence corresponding to the machine language program.

This process can be automated. The recent advances on graphic devices for PC like computers allow graphics interfaces for programs, that formerly would be very slow, to become feasible. The idea is that the user could sketch his data flow program by directly drawing, on the computer screen, the flow graph for its problem. This could be accomplished by means of a graphics editor of flow graphs. Lately the programmer could convert automatically its flow graph to a text file in the assembly language ready to be assembled by **dfA**.

In the case of the μPd 7281, this editor must have an additional complexity, because the nodes of the flow graph are not uniform as they appear to be in the first figure of this paper, but can assume several formats depending on the respective instructions.

The project of this graphic editor was achieved in **C**, making use of a graphics interface developed at **PUC-Rio, Brazil**, called **INTGRAF**, which uses the international graphic pattern **GKS**. As a supplementary advantage, with little adaptations, it can be run in **SUN** machines as well.

The following figure shows a **dfG** screen, and illustrates some of the possible types of nodes used.

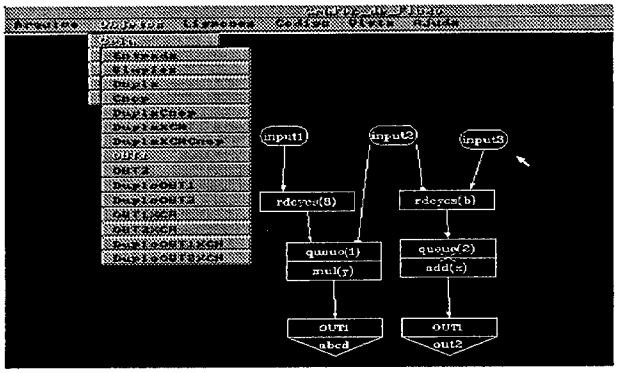


Figure 5 A Screen from the **dfG**

5. Simulator/Debugger dfSim

From the previously exposed, it can be noted the extreme importance of a simulator/debugger which could give support to this programming environment. The simulator/debugger must be considered as a support to the rest of the **dfEnv** components. Therefore, based on this need, it is proposed that the **dfSim** should have the following characteristics:

- must be an auxiliary mechanism to understand the data flow philosophy;
- must be a tool for the data flow developing and debugging programs process;
- must be able to make performance analysis and the result verifications of data flow programs even in computers that do not contain the **dfP** card.

There are five differnt kinds of requirements for this simulator/debugger: principle requirements; data flow requirements; parallelism requirements; μPd 7281 requirements and general requirements, as shown in the sequel..

5.1. Principle Requirements

To attain the aforementioned objectives, the following principle requirements were defined:

1. the user must use **dfSim** in his most intuitive manner;
2. **dfSim** must also accommodate a strong didactic aspect in order to effectively become a helpful tool for the comprehension of the data flow philosophy.

To fulfill such requirements it was determined that all the graphics capabilities available to the standard PC compatibles world should be used at its maximum. Consequently the MS-Windows environment was a good and natural choice for the implementation of **dfSim** since it provides a very friendly and intuitive interface.

5.2. Requirements Specific to Parallelism

For MIMD parallel machines, where multiple and simultaneous data streams may be generated, a simulator must be able to deliver different views of each one of them in an independent way. Fortunatelly, in the MS-Windows platform there is a built-in mechanism that makes natural the visualization of these streams independently of each other; it is called the **Multiple Document Interface**, **MDI** for short.

In the **MDI** interface there is a desktop manager that can display multiple documents independently. Each document can be minimized, maximized, sized or positioned on screen, and displayed side by side with other documents.

In the **dfEnv** environment, multiple processors may be present in the **dfP** card, generating each one a different stream of data. Data in each microprocessor may be viewed under different points of views. Nevertheless, all views of a single microprocessor data must be encapsulated and constitute a single document to the desktop manager. Therefeore, by displaying multiple documents in the screen, the simulator will present to the user a simultaneous view of several processors.

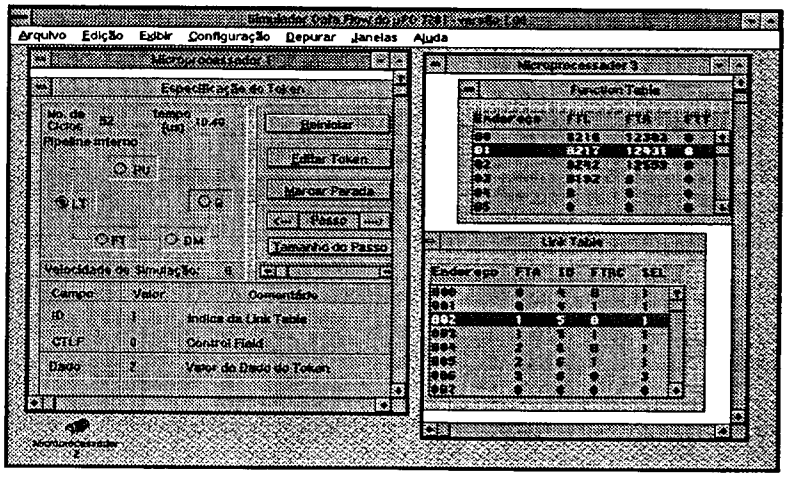


Figure 6 A Screen from the dfSim displaying data from 3 microprocessors

5.3. Requirements Specific to data flow machines

There are several simulators and debuggers for control flow machines available in the market. Since such kind of machines contains a register (the program counter) that points to the next instruction to be executed, these tools are usually manually or automatically, stepped to machine states that deeply depend on the instruction just concluded. The content of this register is, in some way, a manifestation of the current state of these machines. Data flow machines do not possess a similar kind of structure. On the contrary, in data flow machines there is a different sort of concept that helps the execution of many parallel operations at a time, called a **token**. Unlike the program counter that is a physical structure (hardware), a token has not a physical existence. Tokens in a data flow architecture go around throughout many different blocks of the machine, changing their size and/or contents. A data flow simulator and debugger must be capable of representing tokens, as well as watching and interpreting its contents. These are essential requirements, and thus were adopted for this simulator and debugger.

Another characteristic of a data flow architecture is that in order to enable the circulation of several tokens at the same time, practically all data flow machines are strongly dependent on pipelines. Each stage of the pipeline can hold a different token in the same instant of time. Thus it is convenient to a simulator to be able to show all these diferent tokens at different stages simultaneously.

The data flow architecture also imposes its particularities on the debugger. The break point concept, that in control flow machines are also strongly related to the program counter, in a data flow machine must be considered differently. A ***break point condition*** for a data flow machine must be expressed as a relation between internal data using one of the following operators: $=, \neq, >, <, \geq, \leq$. A ***break point*** is thus a logical clause between different ***break point conditions*** on several machine stages at the same time, using one of the following logical operators: $\wedge, \vee, \neg$.

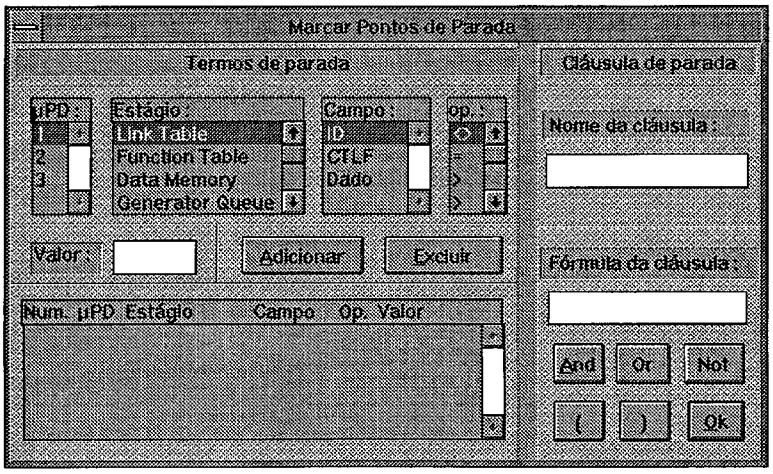


Figure 7 A Window for the specification of a Break Point

5.4. Requirements Specific to the μPd 7281/72181 microprocessor

The internal circular pipeline of the **μPd 7281/72181** is composed of 5 stages, as mentioned before, three of them being memory devices. Tokens may circulate by all the 5 stages simultaneously. Being able to watch the contents of the memory devices, as well as the tokens' form and contents at any stage of the circular pipeline is crucial to someone trying to debug a **7281/72181** program. Moreover, it is very important to make the user task easier, by providing an automatic interpretation of tables' and tokens' contents.

A nice feature is to display a block diagram of the circular pipeline, in order to allow the user to see the token's content inside each block, by simply clicking over it.

5.5. Generic Requirements

Two modes of simulation were required: A dynamic or automatic one, where the user do not interacts with the simulator, except for providing the program to be simulated and for reading the expected computation time and results delivered by the program; a static or step by step method, where the user interaction is much intensive. In the dynamic mode, the user may chose, continuously and at any moment, the simulation speed in order to visually follow the tokens' and data transformations. The simulation speed may be set by a slide rule in the range of zero to SpeedMax. When the simulation speed is set to zero, it is equivalent to chose the static method of simulation. In the static mode, the user may chose the step size and the direction of simulation (forward or backward).

6. High Level Language dfL

In a data flow programming environment it would be very important to dispose also of a high level language.

The existence of some high level languages, developed for specific data flow machines, is well known. The intention is to include, in the future, in this environment, a high level language that will enable the expression of several programming characteristics taking advantage from the data flow philosophy and from the μPd 7281. However, the initial steps for the study and development of this language have not been initiated.

7. Final Comments

Since we are still working on this programming environment, it is not possible to state final conclusions about its performance and operating features. However, since all modules, with the exception of the predicted high level language, are in a very advanced stage, it is possible to make some pertinent comments about important features for a data flow environment.

In the development of all modules of this environment, a great effort was dedicated to provide it with friendly attributes. Consequently, besides modules commonly present on several other programming environments, such as an assembler, a high level language, or a simulator/debugger, a specific data flow module, the graphics editor - **dfG,** was also proposed. It is expected that the **dfG** will allow a data flow programmer to mostly concentrate his or her time to problems that are intended to be solved, rather than to the particularities of the assembler or the microprocessor chip used; the assembler program being generated automatically by the **dfG**.

Even in the simulator/debugger, **dfSim**, specific features such as data flow break points and tokens were emphasized, making it a very interesting tool for the desired process of familiarity increase with a different way of parallel programming.

After the conclusion of this project, a powerfull data flow platform will be available over which it will be possible to perform comparative studies between sequential and data flow oriented algorithms applied to the image processing field.

Acknowledgements

The authors of this paper would like to manifest their gratitude to:

- **COBRA** - Computadores do Brasil, for the supply of the chips used to implement the **dfP**.
- **CNPq** and **FAPERJ** - for the research grants that made possible the continuation of this work.
- Every participant in this project: Eliseu M. Chaves Filho, Átila L. F. Xavier, Alexandre Nigri, Marcus V. Í. Ferreira.

References

[1] Um Ambiente De Programação "Data Flow".
Miguel Menasche, Denis Maciel Maia.
V Simpósio Brasileiro de Arquitetura de Computadores - Processadores de Alto Desempenho, (In Portuguese), Florianópolis - Brazil, september 1993, pp. 275-289.

[2] Manuals from NEC about the **μPd 7281**, **μPd 9305** and **μPd 72181** 1986, 1990.

A MESSAGES DENSITY MONITORING STRATEGY FOR DISTRIBUTED MEMORY PARALLEL SYSTEMS

Luís Paulo Santos[†] **Alan Chalmers**[‡] **Alberto Proença**[†]

[‡]Dept. of Computer Science, University of Bristol, Bristol, U.K.

[†]Dep. Informática, Universidade do Minho, 4719 Braga Codex, PORTUGAL

Tel: +351 53 604479 Fax: +351 53 612954

e-mail: psantos@di.uminho.pt

Summary

Complex applications in distributed memory parallel systems often follows a demand driven approach with domain decomposition. An uniform data distribution among the local memories at the processing elements, may require frequent remote data access. To keep the processors busy while data is remotely fetched, concurrent application processes are assigned to each transputer based processing element. Adding more concurrent application processes in a large scale parallel system may degrade performance, due to the traffic increase with data requests and data block replies. A conditional broadcast mechanism is implemented during data requests, to limit this flow of messages. Monitoring strategies are proposed to further reduce the messages density, and a parameterized model to measure and evaluate global execution times is presented.

Simulation data running the model with up to 35 transputers show that monitoring can reduce the performance degradation when more local concurrence is added. However, if too much data replication is present, the simulation data also show that the supply of communication services at each node still imposes a burden, requiring complementary monitoring strategies to allow removal of redundant reply messages.

1 Introduction

Complex science and engineering applications typically have very high computational demands and deal with large amounts of data. To reduce computation times, parallel processing in a distributed memory system is often considered one of the most promising alternatives. However, certain applications require far more data than can be accommodated at each local memory.

When a processing element (PE) requires data that is not available locally it must be fetched from other PEs. The completion of a task may thus be delayed until data requests are satisfied. In massively parallel systems this delay can be significant. To prevent a severe degradation of

the overall system performance the PEs must be kept busy. To ensure the PEs are busy most of the time with productive work they can be assigned several concurrent application processes sharing the same code. While one process is suspended awaiting for a remote data item, other processes may still be able to proceed, reducing processor idle time.

This approach is in line with the Bulk Synchronous Parallel model proposed by Valliant [1]. This model suggests that adding more application processes at each processor allows communications delays to be effectively hidden, by overlapping computation with communication.

Recent results have shown that increasing the number of processes per processor produces a performance improvement until a certain level is achieved [2, 3]. The overheads of having additional processes above this value are greater than the benefits gained, and the time to solve the problem increases again. The number of application processes a processor can efficiently support seems to be dependent on the messages density within the system. As more processes are launched, more data requests are generated, which may increase the messages latency due to the overload of the communications network (both the channel bandwidth and the support of communication services). This performance degradation is reduced if data locality can be applied during the initial data distribution, such as in non-uniform memory access approaches (NUMA). The work being presented here assumes data locality is unknown beforehand.

Efficient large scale parallel processing under these conditions requires the number of data requests generated by each processor to be kept at an optimum level. If the location of the required data items is unknown then the requests have to be broadcasted; if several copies of the same data item lie distributed amongst the processors, one request may generate several replies, increasing the messages density even further.

This paper proposes a distributed strategy for monitoring the messages density within a distributed memory parallel system. Each processor keeps track of the communication delays with its contiguous neighbours; when this time exceeds a certain average value, the data requests are not sent to those neighbours. The requests are buffered for each of these neighbours waiting for a timeout value; if the data request is satisfied by another neighbour while pending at the source node, it will not be sent.

In a high messages density system, communication overloads can now be localized and the traffic be reduced. Requests may never be sent to some of the neighbours since they were satisfied along other paths within the interconnection network..

2 Characterization of the experimental model

The results achieved with the present work were generated using a network of 35 T805 transputers, running at 30 MHz, with 4 MBytes of RAM each, from Parsytec. The developed system software is based on the demand-driven approach, using domain decomposition. A System Controller processor (SC) performs the system I/O, distributes the original set of data among the PEs, supplies tasks to the requesting PEs and collects the results. The SC processor also acts as an intermediate node on the communications network.

To provide the underlying software structure to implement a demand-driven distributed memory system, several activities need to be undertaken by each PE. These include (figure 1) a router

process (R) which handles the communication of messages that originate at, are addressed to, or pass through the PE (these services are provided by the Real Time System Manager, RTSM, supplied by Parsytec [4]), a data manager process (DM) which manages the data requirements of the application processes (a modified version of the DM presented in [3]), a task manager process (TM) which handles all task requests (which follows the model proposed in [5]) and one or more application processes (APs) which perform the desired algorithm on the data items. An application process controller (APC) is further introduced to provide the interface between the data manager and the aplication processes.

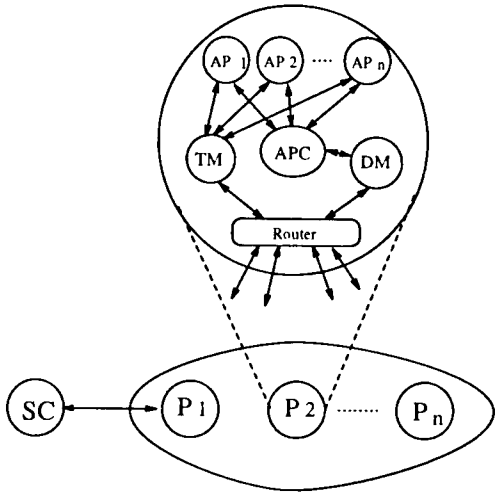


Figure 1: Processing element structure

The solution of a problem on a multiprocessor system using domain decomposition entails applying the same algorithm to different data items in parallel. In this model each AP performs a task by applying an algorithm to one data item to produce a result. Other data items may also be required for the completion of the task. The kind of problems being addressed requires far more data than can be accommodated at each processor's local memory. Therefore, some of these data items may not be available locally and must be fetched from other PEs. Since the address of the nearest PE with the requested data item is unknown beforehand, the request is broadcasted into the network.

Each data item in this model is a read-only fixed-size block of data. The various blocks of data are randomly distributed amongst the processors at start up.

The DM at each PE keeps a list of the data blocks that are available at its local memory; if any of these data blocks is required by the application a request is issued to this process. All the DMs that receive such request check if they can satisfy it. If they can, and if the data block is requested by a local AP, the DM sends back the data block; if the request comes from another processor, the DM directly sends the desired data block to the requesting processor. If the DM can not satisfy the request, it either broadcasts a request to all contiguous neighbours, if the request comes from a local AP, or forwards the received broadcast to a pre-defined set of

neighbours, if the request came from another processor.

When the DM receives a reply to a request it sends the data to the appropriate AP and keeps a copy on a local circular buffer with a pre-defined size (this size gives a measurement of the degree of data replication within the system). As several copies of the same data may exist distributed amongst the processors, one request may result in several identical replies. Only the first one is necessary and all the others must be ignored.

Several application processes at each PE reduces the impact of communication delays by overlapping computation with communication. However, increasing the APs above a certain value may introduce significant overheads which degrades the performance. These overheads may be produced by two factors: process switching and increase of message traffic. Processes have a very fast hardware context switching on transputers, so this factor may be disregarded. Increase to the messages density in the system is due to each processor launching more data requests and overloading the communications network. This increase on the number of messages has three major consequences that impair the system peformance:

- more CPU time is dedicated to message routing, when the CPU processing time is shared with the message routing (as it happens in T800 based networks);
- more CPU time is consumed supplying external data requests with local data blocks;
- longer average message delays to cross intermediate nodes, due to more frequently engaged links.

As more processors are added to the system these negative effects have a greater impact, since more messages are travelling and the average distances that messages have to travel is larger (figure 2). Although the performance gain obtained with multiple concurrent APs increases with the number of processors, the results obtained so far with a net of up to 35 processors show that the optimum number of APs each PE can support is dependent on the messages density within the system and decreases with the number of processors. These results suggested that the use of an appropriate message monitoring strategy could lower the messages density to allow a significant number of concurrent APs per PE on large systems.

A series of experiments were conducted to evaluate how different parameters could affect global execution times, when running a general purpose complex application. Several interconnection topologies were considered - ternary-tree, 2-D mesh and the AMP [5] - but the relative behaviour of them was similar, in spite of significant differences in execution times. The 2-D mesh topology was adopted for its good average path length and expandability, also allowing alternative routes between nodes.

A parameterized running model was then built which lets the user specify for each run:

- the number of APs per processor;
- the number of tasks to simulate the global size of the application;
- the number of data items distributed throughout the system;
- the size of the circular buffer on each DM.

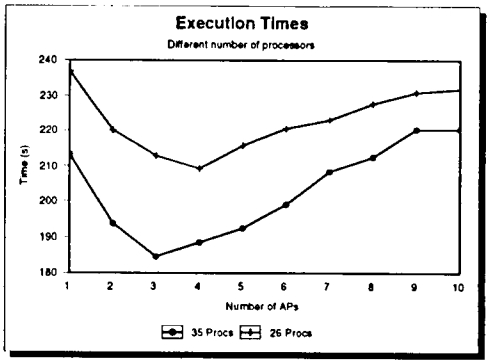


Figure 2: Execution times for different numbers of processors

Each task consists on a pre-defined set of interleaved computational operations (a given number of integer and floating point multiplications) with data fetches. The user may also specify:

- the size of the set of interleaved computational and data fetches operations,
- the size of the computational task, i.e., the number of integer and floating point operations.

The results presented here for discussion were generated using 2-D meshes of 26 and 35 transputers, with the following fixed parameters: 1500 tasks, 20 data items per processor, 200 data fetches per task and 1000 integer and 1000 floating point multiplications between data requests.

Conditional broadcast

When the location of the data items within the system is unknown, data requests need to reach a processor which contains the desired data. Within a non-disjoint interconnection network, more than one route may exist from every PE to every other PE. The set of routes which give the shortest distance between a PE and every other PE is called the spanning tree for that source PE.

When performing a global broadcast, a message from the source processor is sent to all its neighbours. In a transputer network, each contiguous neighbour receives the message to broadcast and forwards it to all PEs in the same spanning tree, until all of them have been reached. This requires that every PE has a table with all relevant information about the spanning trees.

Since the broadcasted messages are data requests, one way to reduce the traffic of messages is to delay a forward at each PE, until its DM acknowledges that the requested data block is not present at that PE. If the data item is there, the broadcast is not further propagated through that spanning tree branch.

This implementation of *conditional broadcast* reduces not only the traffic of messages with data requests, but also the number of returning copies of the requested data item. The higher up it

is this non-propagating processor in the source PE's spanning tree, the more effective will be this non-propagation of data requests.

Reducing redundant messages

Each DM has a circular buffer with copies of the latest data block it received. This data block's replication has a random pattern within the system. While this feature helps decreasing the average distance at which data requests are satisfied, it also generates more messages, containing the same data item, being returned from different processors. All but one of these messages are redundant. Results obtained so far show that increasing the size of the buffer at the DM (and consequently increasing the degree of data replication in the system) degrades the performance for a large number of processors (figure 3).

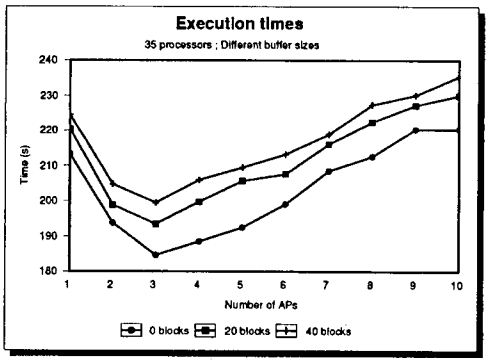


Figure 3: Different buffer sizes

If messages density needs to be reduced, then the redundant messages must be removed as soon as possible, ensuring that at least one desired copy of the data item does indeed reach the destination DM. Several techniques for reducing redundant messages are described in [5]. Some of these rely on intermediate nodes to inspect passing-through messages. The work performed so far only considers the conditional broadcast mechanism to reduce redundant messages.

3 Monitoring strategies and results

To keep the messages density at an appropriate level to achieve optimum efficiency, it may be necessary to reduce the number of messages, and eventually to delay the transmission of messages whenever the network is identified as overloaded. A monitoring strategy which detects the slower throughput channels between processors lets the individual PEs decide whether or not a message should be delayed.

Data requests are the main cause for the heavy traffic being generated, because they are broadcasted and can cause several identical replies. The messages density monitoring can be used to

decide whether a data request should be launched immediately on the net, or if the DM should delay it until the messages density fall below a given level.

To identify the slower channels that may cause the bottleneck, one possible monitoring strategy evaluates the reply time to a data request. By continually updating the average time to satisfy a data request, each PE can identify the path of the longer replies, to delay further requests until they are satisfied in acceptable times or a given timeout is ellapsed. As the time a message takes to arrive to its destination is proportional to the number of hops, the distance between the source and the destination processors (taken from the spanning tree) must be considered when computing the average value.

This strategy, based on the time a message takes to go through the whole path, makes a serious misjudgement, since it does not pin-point the bottleneck: it assumes the whole network is busy in that direction.

Another way to identify a slow channel is to check the communication time between the source PE and its contiguous neighbours. Every time a PE receives a data request, it checks if the source PE for that request is its contiguous neighbour; if so, a very short acknowledge message is sent back. The source PE uses this acknowledge message to compute the time taken since the request was issued until the acknowledgement was received. If this time is longer than a specified value, future data requests are not immediately sent to this neighbour. They are queued waiting for a timeout. Once the timeout ellapses, communication with this neighbour is resumed and the suspended data requests are sent to it. This is done separately for each neighbour.

This *close-neighbour-delay* strategy only considers the communication load of each processor's immediate neighbours, and only cancels communications with those processors that might be overloaded. If a request is satisfied from another channel before the timeout occurs, requests waiting on the queues are removed, and not sent to those branches of the spanning tree. Therefore, this strategy also reduces the number of messages launched in the system.

The experimental results given below consider a timeout obtained by multiplicating a given constant by the average time needed to send the data request and receive the acknowledge message (figures 4 and 5). The chosen timeout period depends on the number of processors and network topology, and it must be large enough to allow data requests to reach a significative number of processors and for replies to be received.

The results obtained also show that monitoring produces better performance values when a certain degree of data replication is allowed and the timeout period is significantly larger than the average value.

Implementation of a monitoring strategy in a system with no data replication produces worst results. The monitoring overheads do not compensate the delay of data requests, since the probability of a request being removed from a waiting queue is very low (the number of messages is not significantly reduced). And if there are no copies of the data items, only one processor will be able to satisfy the request; if the request to that processor's branch of the spanning tree is delayed, this will only increase the necessary time to satisfy the request.

When monitoring is active, the overheads due to receiving several identical replies to the same data request are lower than the benefits of not sending the messages to all the branches of

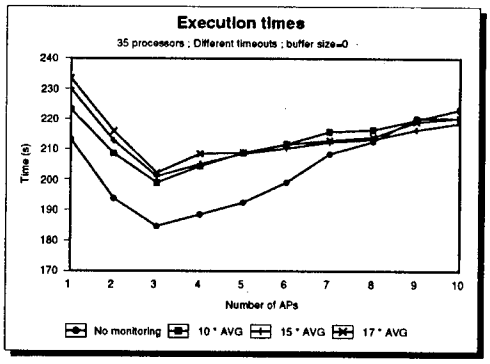


Figure 4: Different timeouts without buffering

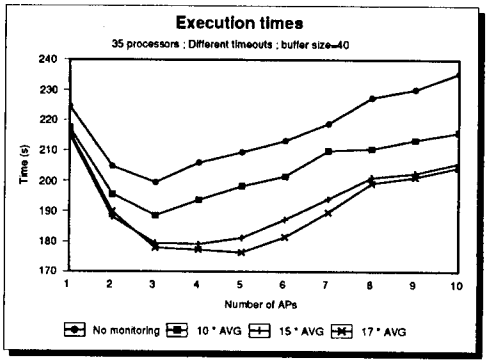


Figure 5: Different timeouts with buffering

the spanning tree. This suggests that the use of redundant messages removal techniques at intermediate nodes may improve the obtained results.

4 Concluding remarks

In a demand driven approach with domain decomposition, processors can be kept busy by overlapping communication with concurrent application processing. However, increasing the number of concurrent application processes causes an increase to the messages density due to remote data accesses, overloading the communication network and impairing system performance. The messages density can be controlled if the flow of messages is monitored.

Remote data accesses in an uniform data distribution system may cause a broadcast of requests, and redundant replies when data is replicated. Conditional broadcast mechanisms reduce this traffic, and it can be further reduced by monitoring the requests and introducing delays on the

request's transmission.

The selected monitoring strategy analyses the *close-neighbour-delay* instead of the overall reply time to a request. This approach produces better results, specially when data is replicated on the network. The obtained results with different timeout values suggest that this improvement may be caused by the arrival of replies from the branches of the spanning tree that were not delayed, while the request was pending at the source node. Applying this strategy to the intermediate nodes may not improve the performance, if the replies are directly sent to the requesting node.

Further improvements can yet be achieved if higher priority is assigned to the DM (together with the router process). This approach could reduce messages delays, speeding up execution times.

A monitor strategy that supports the delay of data request helps to reduce the messages traffic, since some data requests may be cancelled. However, several redundant replies may still circulate in the network, due to the requests placed on the other branches. Monitoring should be extended to the income replies at the intermediate nodes, to allow both the removal of redundant replies, and the use of the *close-neighbour-delay* monitoring strategy at these nodes.

Further simulations are still required to evaluate the behaviour of a much larger multiprocessing system. Although these results can be crucial to assess the monitoring impact on massive parallel systems, present results show that the proposed monitoring strategies are promising.

References

[1] Leslie G. Valliant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103-111, August 1990.

[2] Alan Chalmers, David Stuttard, and Derek Paddon. Data Management for Parallel Raytracing of Complex Images. In S.P.Mudur, editor, *4th SIAM Conference on Parallel Processing for Scientific Computing*, Bombay, February 1993.

[3] Luís Paulo Santos, Alan Chalmers, and Alberto Proença. A Data Management Strategy for Increased Parallel Processing Efficiency. *Computing Systems in Engineering*, 1994. To be published.

[4] Andreas Maassen, Armin Joachimsmeyer, and Bernhard Piasta. *RTSM - Technical Documentation.* Parsytec Industriesysteme GmbH, 0.95 edition, February 1993.

[5] Alan Chalmers. *A Minimum Path System for Parallel Processing.* PhD thesis, Department of Computer Science, University of Bristol, April 1991.

Graphical Construction of Parallel Programs

G. R. Ribeiro Justo
Centre for Parallel Computing, University of Westminster, London, W1M 8JS
e-mail: justog@wmin.ac.uk,

Abstract

Parallel programming is not difficult, as the programs build up their complex behaviours in a similar way to the real world (i.e through the simple interaction of independent and simple entities). The parallel system engineer needs, however, a systematic method to decomposing the networks into independent ones or composing existing processes to form new networks. In this paper, we introduce a technique for the graphical construction of hierarchical networks (or configurations) of processes. The technique focuses on the concept of templates *which define reusable patterns of communication and synchronisation for processes. We introduce a set of graphical rules based on the equivalence between processes, more specifically* templates, *and networks (configurations) of* templates. *The rules can be used to decompose networks of processes by substituting a single process for an equivalent sub-network of processes, or to abstract a sub-network of processes as a single process in order to simplify complex networks.*

Keywords: Parallel Programming, Graphical Design, Program Transformation

1. Introduction

There is a general consensus that software systems should be constructed in a hierarchical modular way based upon the composition of software components. The underlying model commonly used to describe and implement parallel systems as a network of (hierarchical) communicating processes satisfies that requirement. This suggests that the software architectural description (network or, as we usually call it, "configuration" of processes) is essential for all phases of the development of parallel software from specification, which can be seen as a configuration of component sub-specifications, to evolution, as changes to a system configuration can be carried out dynamically during the system execution by removing or including new processes.

To simplify the construction of configurations, graphical notations have been widely used [1, 2, 3, 4]. The graphical representation of a parallel program configuration is usually simple. It includes icons (usually boxes or circles) to denote processes, lines to denote communication or synchronisation paths between processes and arrows to represent the flow of messages between the processes.

Semantics for configurations of processes has also been proposed [1, 5]. The semantics is usually defined in terms of the structure of the configuration (structural semantics [5, 6]) that gives information about nodes (processes), ports and links. However, we also need information about the states and external events the configuration can perform which are defined by a behavioural semantics [5, 7, 8].

Several attempts have been made to combine the structural and behavioural aspects of a configuration in the same graphical representation. A simple way of doing this is to include behavioural expressions for the links of each process. Another approach is to ap-

ply the concept of timing thread [9] that specifies the ordering of events of the processes' ports (interface). These notations, however, usually complicate the diagram.

One problem encountered is that, in general, there is no technique for semantic-preserving transformation or verification of the behavioural properties of the configuration using the graphical notation directly. This is usually done on the textual (behavioural) specification of the configuration, for example, using the formal description of the configuration written in a CCS [8] or CSP [7] specification of the configuration.

In this paper, we show that the concept of *template* [10, 11] can be used to define rules that allow us to transform (abstract or decompose) graphically configurations of processes. The rules relate *templates* and configurations of *templates* that are behavioural equivalent. Thus, we can transform a configuration by substituting a configuration of *templates* for an equivalent *template* (abstraction) or by substituting a single *template* for an equivalent configuration of *templates* (decomposition).

These rules are important during the development of parallel programs. Abstraction can be applied, for example, to reduce the number of processes of a configuration. Deadlock-freedom properties can, therefore, be checked more easily. Also, during the allocation of the processes to the target architecture, we usually need to alter the number of processes – sometimes there are too many processes, and sometimes there are too few. This means that configurations have to be serialised into single processes (abstraction), or processes have to be parallelised into configurations of processes (decomposition).

In this paper, we will concentrate on two kinds of *templates*, namely, IO-SEQ and IO-PAR, that we have been using in many applications, and the graphical rules associated with them. The remainder of the paper is set out as follows. Section 2 reviews the concept of *template* (focusing on the IO-SEQ and IO-PAR *templates*) and its graphical representation. In Section 3, the rules are introduced. Section 4 covers examples of the use of the rules in the construction of configurations. Finally, Section 5 presents comments and directions for future work.

2. The Concept Of template

A *template*, basically, defines a type of synchronisation or communication pattern that a process can perform. We can identify many of these patterns that define the behaviours of general processes. For instance, a process server generally behaves as follows. It receives requests from several client processes, selects one of them, performs some activity for the chosen client and at the end sends back the result. We have studied several types of *templates* but for brevity we will concentrate on only two of them.

The first *template* is called IO-PAR and defines a process that sends and receives messages in parallel. More formally, if we assume the alphabet of an IO-PAR process P to be αP, then the behaviour of P is defined by the following traces [7]:

$$\begin{aligned} traces(P) &= INTERLEAVES*(\alpha P) \\ &\cup\ (INTERLEAVES(\alpha P) \wedge traces(P)) \end{aligned} \qquad (1)$$

where the function *INTERLEAVES* computes permutations of events and the function *INTERLEAVES** applies *INTERLEAVES* to the powerset of the process' alphabet. The

definition above determines that the traces of P are described by all interleavings of events in the alphabet.

The second *template* is called IO-SEQ and corresponds to a process that receives a set of messages in parallel, performs some computation and sends another set of messages. In this case, the traces of an IO-SEQ process Q with alphabet $\alpha Q = in(\alpha Q) \cup out(\alpha Q)$ are characterised by the sequences of two interleavings — one for the receives and another for the sends:

$$\begin{aligned} traces(Q) \;=\; & INTERLEAVES*(in(\alpha Q)) \\ \cup\; & (INTERLEAVES(in(\alpha Q)) \wedge INTERLEAVES*(out(\alpha Q))) \\ \cup\; & ((INTERLEAVES(in(\alpha Q)) \wedge INTERLEAVES(out(\alpha Q))) \\ & \wedge traces(Q)) \end{aligned} \tag{2}$$

Every *template* is given a unique graphical notation. In the case of IO-PAR and IO-SEQ *templates*, a box stands for an IO-PAR *template* and a box divided into two segments an IO-SEQ *template*. Each segment corresponds to the receives and sends respectively. Other types of representations can be seen in [11, 12].

The *templates* can be composed in parallel to form configurations. In a configuration (graph), edges symbolise internal links between pairs of processes. In an edge, the source node is the process that sends messages and the target node is the process that receives the messages. If an edge is linked to only one process it means that the process can communicate with the external environment. The names of the edges – i.e. the names of the events generated by the processes are, for simplification, only named when necessary.

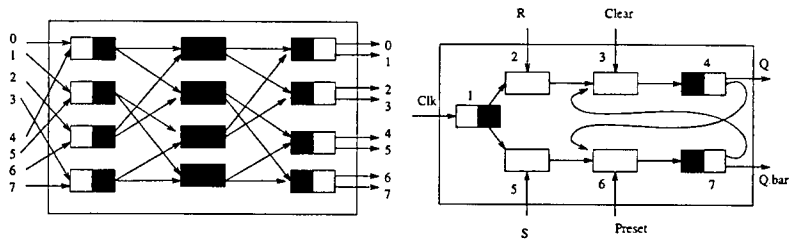
It is worth considering an important property of the configuration graphs of IO-PAR and IO-SEQ *templates*. As the behaviour of the *templates* specifies all sends always occur in parallel, and all receives also occur in parallel, when several links exist between two *templates* in the same direction, only one edge needs to be represented in the graph because communications can occur at the same time. This means that the configuration graphs do not contain "parallel edges" between *template* vertices. Also, to simplify the application of the graphical rules, half of the box representing an IO-SEQ *template* is sometimes shaded. A box with the left side shaded describes a process that does not receive messages from the environment. Similarly, if right half of the box is shaded the process does not send messages to the environment.

Figure 1(a) shows a configuration of IO-SEQ *templates* that computes an 8-point Fast Fourier Transform (FFT). Every *template* corresponds to a **butterfly** process that executes a two-point FFT.

Figure 1(b) illustrates a configuration of IO-PAR and IO-SEQ *templates* that simulates a simple flipflop device. The IO-SEQ *templates* 1, 4 and 7 refer to wiring (splitting) processes. Processes 2 and 5 are IO-PAR *templates* that refer to **and** gates. Processes 3 and 6 are IO-PAR *templates* that refer to **nor** gates. The **and** and **nor** IO-PAR gates must send null values initially because the IO-PAR behaviour specifies that sends and receives occur in parallel. More details on how these circuits work can be seen in [13].

3. Equivalence Rules for Templates

Our technique is based on rules that define the equivalence between *templates* and con-



(a) (b)

Figure 1: Configurations of IO-PAR and IO-SEQ templates. (a) FTT (b) Flip-flop.

figuration of *templates*. Since IO-PAR and IO-SEQ *templates* have particular properties, we have defined a special kind of equivalence that we call *p-equivalence* [10]. We use the concept of *p-traces* (*precedence traces*) that defines the possible ordering (or precedence) of the events in the alphabet of a process derived from its traces. The main idea is to determinate how processes interact with IO-PAR and IO-SEQ *templates*, simplifying the proof of properties of configurations of *templates*. Instead of comparing traces of processes, we compare *p-traces* or the more simple initial *p-traces*. In this case, two processes that have the same (initial) *p-traces* are *p-equivalent*. For example, an IO-PAR *template* P with alphabet αP has the initial *p-traces*:

$$p\text{-}traces(P) = \{s \mid s \leq u \text{ and } u \in INTERLEAVES(\alpha P)\} \quad (3)$$

So, every process that has the same initial *p-traces* of P is *p-equivalent* to P. Similarly, an IO-SEQ *template* has the initial *p-traces*:

$$\begin{aligned} p\text{-}traces(P) &= \{t' \mid t' < u \text{ and } u \in INTERLEAVES(in(\alpha P))\} \\ &\cup \{t' \wedge t_2 \mid t' \in INTERLEAVES(in(\alpha P)) \text{ and} \\ &\quad t_2 \leq v \text{ for } v \in INTERLEAVES(out(\alpha P))\} \end{aligned} \quad (4)$$

3.1 Equivalence Rules for IO-PAR templates

In Figure 2(a) we present equivalence rules that relate configurations of IO-PAR and IO-SEQ *templates* with IO-PAR *templates*. The strategy we used to define the complete set of rules was to systematically generate every directed graph with three nodes including IO-PAR and IO-SEQ nodes. We then proved by induction that we can apply the rules to graphs of any size. The induction cannot start with graphs of only two nodes because of the fork and join configurations shown in **Rule 4** and **Rule 5**.

The rules are usually simple and easy to understand. More complex rules can be defined from the basic rules. **Rule 1** says that a configuration of two IO-PAR *templates* that communicate is *p-equivalent* to an IO-PAR *template*. The dotted lines denote edges that may exist but are not a condition for application of the rule. This means that **Rule 1** can be applied to any kind of configuration of two IO-PAR *templates*. The *p-equivalent* IO-PAR *template* sends (receives) messages to (from) the environment like both *templates*.

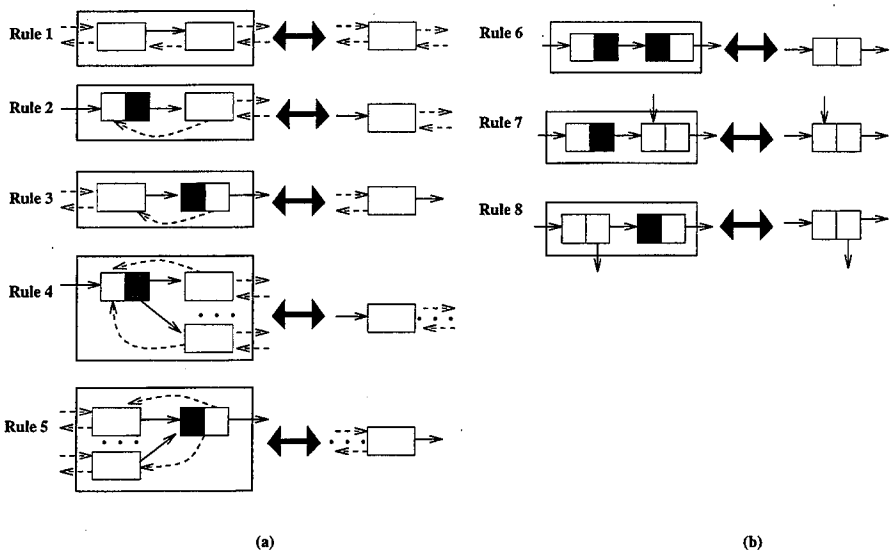


Figure 2: Graphical equivalence rules for configurations of IO-PAR and IO-SEQ templates.

In **Rule 2**, the IO-SEQ *template* must only send messages to the IO-PAR *template* but it receives messages from the environment and can also receive messages from the IO-PAR *template*. The *p-equivalent* IO-PAR *template* receives messages from the environment like the IO-SEQ *template*, and also sends (receives) messages to (from) the environment like the IO-PAR *template*. **Rule 3** is the reverse of **Rule 2**, as the IO-SEQ *template* must only receive messages from the IO-PAR *template*.

Rule 4 is a special rule. It corresponds to a kind of "fork" configuration where the IO-SEQ *template* connects a set of IO-PAR *templates*. In the diagram, the "..." determines that there can be an arbitrary number of IO-PAR *templates*. Note also that the IO-PAR *templates* do not communicate. If they did, we could have applied **Rule 1**, and then **Rule 2** to prove that the configuration was *p-equivalent* to an IO-PAR *template*. The *p-equivalent* IO-PAR *template* in **Rule 4** receives messages from the environment as the IO-SEQ *template* and also sends (receives) to (from) the environment like every IO-PAR *template* in the configuration. The last rule, **Rule 5**, is the reverse of **Rule 4**.

3.2 Equivalence Rules For IO-SEQ templates

Unlike an IO-PAR *template* where the events in the alphabet may occur in any order, an IO-SEQ *template* imposes an ordering in the execution of the events in its alphabet such that the receives always occur before the sends. Therefore, any configuration that includes at least one IO-PAR *template* cannot be equivalent to an IO-SEQ *template*.

Figure 2(b) introduces the basic equivalence rules for configurations of only IO-SEQ *templates*. **Rule 6** specifies a configuration of two IO-SEQ *templates* where the first pro-

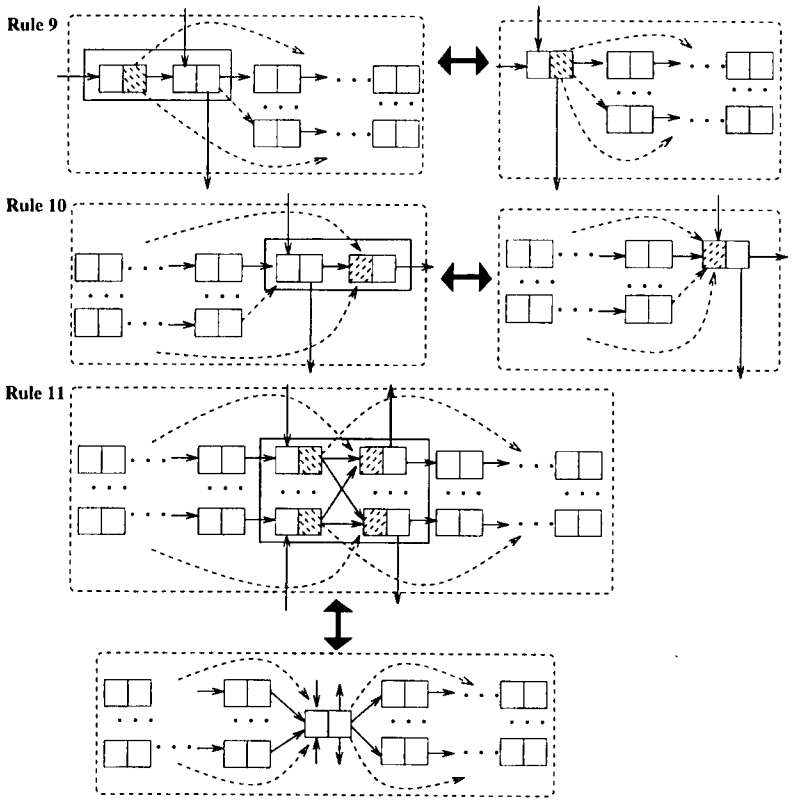


Figure 3: Context-dependent equivalence rules for configurations of IO-SEQ templates.

cess only receives messages from the environment, and only sends messages to the second process. On the other hand, the second process only receives messages from the first process and only sends messages to the environment. The *p-equivalent* IO-SEQ *template* receives messages from the environment like the first process, and sends messages like the second process.

In **Rule 7**, the first process is similar to **Rule 6**. However, the second process does not have to receive messages only from the first process, it can also receive messages from the environment. In this case, the *p-equivalent* IO-SEQ *template* receives messages from the environment like both processes. **Rule 7** is the reverse of this rule.

Another group of equivalence rules for configurations of IO-SEQ *templates* is presented in Figure 3. These rules are only valid in a special context (environment). The context is described as a configuration of *templates*. In the graphical rule, it corresponds to the configuration outside the solid line box delimited by the dotted rounded-corner box.

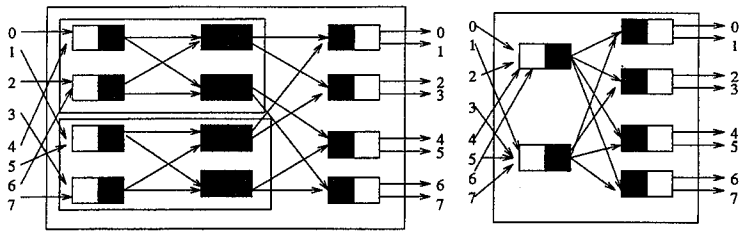


Figure 4: Configurations *p-equivalent* to the FFT configuration.

To explain the use of the context in the rules, assume a configuration similar to that defined in **Rule 7** where the first process also sends messages to the environment. This means that this process may send messages to the environment before the second process has received its messages from the environment. Hence, the configuration does not satisfy the behaviour of an IO-SEQ *template* that only sends messages (to the environment) after it has received all its messages.

Rule 9, however, requires that the first process can only send messages to processes in the environment that are part of a sequence of IO-SEQ *templates* receiving messages from the second process in the configuration. This condition guarantees that the processes in the sequence cannot send messages until they have received the messages from the second process in the configuration. Observe also that the dotted lines inside the IO-SEQ *template* indicate that it may only send (receive) messages to (from) that particular environment – that is, the sends (receives) are internal if we consider the configuration together with the sequences of processes in the environment. Similar conditions are required by **Rule 10** which is the reverse of **Rule 9**.

Rule 11 is the more complex rule. It is a generalisation of **Rule 6**. It requires that all processes that send messages to the environment can only do so after receiving the messages from all processes that receive messages from the environment. This guarantees that the sends only occur after all receives. The context is similar to the combination of the ones of **Rule 9** and **Rule 10**.

4. Using the Equivalence Rules

4.1 Simplifying Configurations

One of the main applications of the rules is in the simplification of complex configurations. For example, the configuration in Figure 1(a) can be proved to be *p-equivalent* to a single IO-SEQ *template*. The first step of the proof is to rearrange the processes as Figure 4(a) shows. It then becomes clear that we can apply **Rule 11** to each of the two sub-configurations. The result is the *p-equivalent* configuration in Figure 4(b). This configuration satisfies **Rule 11**, therefore it is *p-equivalent* to a single IO-SEQ *template*.

Figure 5 illustrates configurations *p-equivalent* to the flipflop in Figure 1(b). In Figure 5(a), each one of the two sub-configurations can be substituted by *p-equivalent*

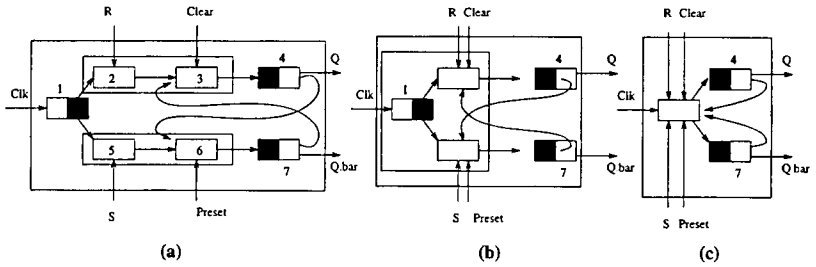


Figure 5: Configurations *p-equivalent* to the flipflop configuration.

IO-PAR *templates* by applying **Rule 1**. Therefore, the configuration in Figure 5(b) is *p-equivalent* to the flipflop configuration in Figure 5(a). Also, by **Rule 4**, the sub-configuration in Figure 5(b) is *p-equivalent* to an IO-PAR *template*. This shows that the configuration in Figure 5(c) is also *p-equivalent* to the flipflop configuration.

4.2 Decomposing Configurations

As we said, the rules can also be applied to the decomposition (refinement) of a configuration. This is illustrated by a Hospital Monitoring System. The main components of the system are: doctor, nurse, pharmacist and patient. The doctor receives information from the patient about his/her condition, and reports from the nurse about the patient's progress. He/She then sends instructions to the nurse about the the treatment to be applied to the patient. The nurse monitors the patient, and requests the pharmacist to send the prescribed medicine.

We assume that the system starts when the nurse requests data about the new patient, informs the doctor he/she has started, and asks the pharmacist to get ready. However, the doctor, the patient or the pharmacist may arrive before the nurse. In this case, we associate an IO-PAR *template* with the *Nurse* process to indicate that possibility. The patient waits for treatment from the nurse, and sends information about his/her state to the doctor and the nurse. So, the *Patient* process behaves like an IO-SEQ *template*. The doctor receives information from the nurse and the patient, and asks the nurse to apply the treatment. In this case, the *Doctor* process also behaves like an IO-SEQ *template*. Finally, the pharmacist only receives requests from the nurse and sends back the medicine. This corresponds to a simple IO-SEQ *template* behaviour. The graphical solution to the Hospital System using IO-SEQ and IO-PAR *templates* is presented in Figure 6(a).

The decomposition strategy depends on the particular design. In general, we try to increase the strength of each module (in our case processes), and reduce the coupling among modules.

Figures 6(b) and 6(c) show two possible decompositions for the *Nurse* process (module). In Figure 6(b), the decision was to decompose the *Nurse* process' function into three sub-functions performed by different processes: *Collect Data*, *Store and Compute* and *Generate Reports and Requests*. Applying **Rule 1** to the *Collect Data* and *Store and Compute* processes, we prove that they are *p-equivalent* to an IO-PAR *template*. We can

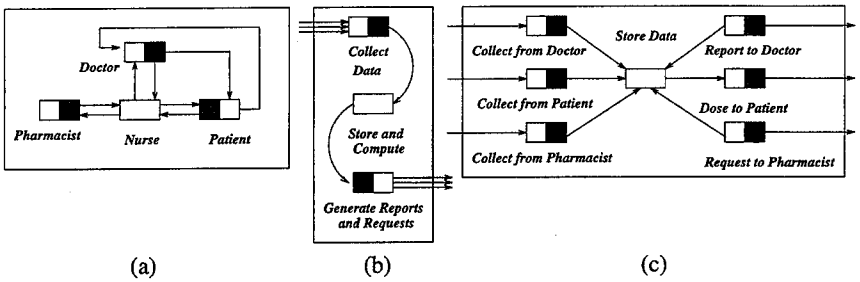


Figure 6: Hospital configuration.

then apply **Rule 2** to the (simplified) *p-equivalent* configuration and process *Generate Reports and Requests* to show that it is *p-equivalent* to the *Nurse* process, as expected.

Another decomposition of the *Nurse* process is presented in Figure 6(c). Here, the data collect function was decomposed assuming that different consistency checks are required. Similarly, as data is generated in different forms the generate reports and requests function was also decomposed. To prove that the new configuration is *p-equivalent* to the *Nurse* process, **Rule 1** can be applied successively to each collect data process (*Collect from Doctor*, *Collect from Patient* and *Collect from Pharmacist*) and process *Store Data* to prove that they are *p-equivalent* to an IO-PAR *template*. Next, **Rule 2** is applied successively to this IO-PAR *template* and the *Report to Doctor*, *Dose to Patient* and *Request to Pharmacist* processes, proving also that they are *p-equivalent* to an IO-PAR *template*.

5. Conclusions

In this paper, we introduced a set of graphical rules that relate configurations of *templates* to *templates*. These rules are very useful in the hierarchical development of configurations of processes because they allow us to refine a *template* into sub-configurations creating a lower level configuration, or abstract a configuration as a single process creating a higher level configuration.

An important property of our technique is to provide a compositional proof of deadlock-freedom — that is, if we prove that a high level configuration is deadlock-free, by applying the rules we can guarantee that the decomposed (lower) level is deadlock-free, since the rules preserve the behaviour of the configuration. Also, if we prove that a low level configuration is deadlock-free and *p-equivalent* to a *template*, when we reuse that process in a higher level configuration we do not have to check its proof of deadlock-freedom. In general compositional proof of deadlock-freedom is not easy because when we compose deadlock-free processes together they can interfere with each other [14].

At the moment we are investigating new rules for other kinds of *templates*. We are also developing a graphical environment that allows us to build and analyse configurations of *templates*, and at the same time derive parallel programs from the configurations of *templates*.

6 Acknowledgments

I am very grateful to Peter Welch who has given many suggestions to improve this work. Special thanks also to my colleagues at the Computing Laboratory at the University of Kent, England, with whom I have shared some of theses ideas.

References

[1] R. Milner. Flowgraphs and flow algebras. *Journal of ACM*, 26(4):794–818, October 1979.

[2] K. G. Shin D. Peng. Modeling of concurrent task execution in a distributed system for real-time control. *IEEE Transactions on Computers*, C-36(4):500–516, April 1987.

[3] J. Kramer, J. Magee, and K. Ng. Graphical configuration programming: The structural description, construction and evolution of software systems using graphics. *IEEE Computer*, 22(10):53–65, October 1989.

[4] T. Bolognesi. A graphical composition theorem for network of LOTOS processes. In *10th Int. Conf. on Distributed Computer Systems*, pages 88–95, Paris, France, May 28 – June 1 1990. IEEE.

[5] J. Parrow. Structural and behavioural equivalences of networks. *ICALP 90*, 1990.

[6] J. Parrow. The expressive power of simple parallelism. In *PARLE'89*, pages 389–405. Springer-Verlag, 1989.

[7] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[8] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[9] R. J. A. Buhr and R. S. Casselman. Architectures with pictures. In A. Paepcke, editor, *OOPSLA'92: Conf. on Object-Oriented Programming Systems, Languages and Applications.* ACM Press, October 1992.

[10] G. R. R. Justo. *Configuration-oriented Development of Parallel Programs.* PhD thesis, University of Kent at Canterbury, England, 1993.

[11] G. R. R. Justo and P. R. F. Cunha. Deadlock-free configuration programming. In *2nd Int. Workshop on Configurable Distributed Systems, SEI, Carnegie Mellow University.* IEEE Computer Society, March 1994.

[12] P. Welch, G. R. R. Justo, and C. Willcock. High-level paradigms for deadlock-free high-performance systems. In R. Grebe, J. Hektor, S. C. Hilton, M. R. Jane, and P. H. Welch, editors, *Transputer Applications and Systems '93*, volume 1, pages 981–1004. IOS Press, September 1993.

[13] P. H. Welch. Emulating digital logic using transputer networks (very high parallelism = simplicity = performance). In *Parallel Architecture and languages Europe*, volume 1, pages 357–373. Springer-Verlag, June 1987. LNCS 258.

[14] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof Theories For Network of Processes and Their Relationship.* Springer-Verlag, 1987.

Ada for multicomputers

Bénédicte Herrmann, Guy-René Perrin
LIB, Université de Franche-Comté, 25030 Besançon Cedex, France
E-mail: herrmann@comte.univ-fcomte.fr

Abstract : To facilitate the use of multicomputers, we aim at providing a standard Ada environment wich takes benefit of multicomputers. We have studied the way a distributed Ada Run Time can be defined on a distributed operating system in order to hide distribution to programers. A prototype, called DART, has been realized: its implementation lies on the Chorus/MiX operating system wich provides a support for distributed applications.
Keywords : Ada, Distributed Operating System, Multicomputer, Parallel programming.

1 Introduction

Programming languages are a key to take benefit of a computer. Current programming languages proposed on multicomputers are generally classical languages to which communication primitives are added. Distribution is not really integrated in these languages: users must manage themselves the distribution of their application by cutting it in several programs. Moreover these programs communicate by using specific primitives. However, some languages integrate distribution as a facility: unfortunately they are not widely implemented in distributed environments. For instance, Ada [1] provides a well suited support for multicomputer through the notion of *tasks* which are concurrent execution entities and the notion of *rendez-vous*. Thus, to facilitate the use of multicomputers, we aim at providing a standard Ada environment which takes benefit of multicomputers. The implementation of our distributed Ada Run Time prototype lies on the Chorus/MiX distributed operating system [2] which was implemented on multicomputers [3] [4]. As the design of Chorus/MiX provides a support for distributed applications, it it also well suited for the implementation of a distributed Run Time.
In this paper, we present this Run Time prototype. In a first section, we briefly present research projects which aim at executing Ada programs on distributed architectures. Then, we describe the distributed Run Time. The last section gives an evaluation of this study.

2 Distributed Ada research projects

Several implementations of ADA have been realized for distributed architectures. Among these implementations we can distinguish between shared memory implementations - where a global instance of the Run Time manages the execution of Ada applications - and distributed memory implementations - where several instances of the Run Time cooperate to manage Ada applications on several nodes. We focus on projects realized on distributed architectures with no global memory.

2.1 Ada applications composed by several programs

To execute an Ada application in a distributed environment, [5] [6] consider this application as several programs which do not share data. These programs communicate explicitely by

communication primitives: these primitives are not integrated in the language. In particular, programmers do not use rendez-vous or remote procedure calls. Communication primitives used in this case are specific packages which are added as language libraries and are compiled with the distributed Ada application. Using this environment, three main issues must be addressed by the programmer :

- he must design his application in a distributed way. The application must be developed as several programs - not as a unique program as in standard Ada - and the mapping of these programs must be taken into account in the design of the application,
- Ada applications are not standard because programs can not communicate by rendez-vous. This implies that applications are not portable to different environments.
- Ada type checking can not be used for the whole application but only for each program.

As a conclusion, we can say that this solution is not really adapted to parallel programming because it does not integrate distribution in the Ada language itself.

2.2 Ada applications composed by a simple program

Another way to write distributed Ada applications is to consider an application as a single program. This allows to solve the issue of type checking in the same way as on centralized architectures.

Ada program scattered in entities

In this approach [7] [8] [9] [10], an Ada progam is scattered in independant entities, called *virtual nodes* (tasks, packages or subprograms), which can be distributed. Programmers do not design their application as several programs but as several entities which do not share memory. Communication between entities is done by rendez-vous or remote procedure calls. Thus, by designing an Ada application as a unique program, type checking is applied to the whole application.

Generally, programmers must cut themselves their programs and give informations to describe the mapping of the entities on the nodes. Some projects also provide tools to easily generate these informations (graphical tools, description language, etc.). These projects do not address the issues of hidding distribution at the programmer level. Moreover, the applications are not truely standard Ada applications because they need mapping informations to be run.

Ada program with hidden distribution

This last approach provides the level we aim to provide on a multicomputer : programmers design their program in the same way to execute it on a multicomputer as on a centralized architecture. In the case of [11] [12] [13], distributed Ada programs execution is done whithout programmer's help. Restrictions are generally imposed on the Ada language, mainly on variables shared between tasks. Moreover the implementations of these Run Times are rather complex because they do not benefit from the support of a true distributed operating system. In our project, we wanted to address the three issues previously evoked and to remove all restrictions to the implementation of full Ada. This was possible by using the services provided by the distributed operating system Chorus/MiX.

3 DART : a Distributed Ada Run Time

A Run Time is the interface between the code generated by the compiler and the operating system. The standard Ada Run Time was designed to be executed on a mono-processor computer. It has been implemented in a centralized way : a unique instance of the Run Time supports the execution of the tasks and a multi-tasked program is executed in pseudo- parallelism on a processor. Thus the data structures describing the tasks can be accessed at any time and from any task during an execution.
This Run Time can be adapted to a shared memory multi-processor without main changes in the implementation and in the data structures management by taking benefit of the shared memory. However the Run Time can not be used on a multicomputer without adapting the distribution of the data and of the functions management.
Our aim was to execute standard Ada programs on multicomputers in a distributed way. We have modified the Run Time to adapt it to multicomputers by using the services of a distributed operating system. We are concerned with the issues generated in the execution of an application. We can distinguish three main classes of issues :

- global management of the execution,
- implementation of Ada features as tasks, rendez-vous, etc.
- implementation of Ada functions as tasks management, call of an entry point, etc.

These issues have been addressed by using the services of the Chorus/MiX operating system : remote execution to preserve the dynamic aspects of an Ada execution, threads or lightweight processes to support tasks, transparent inter process communication to implement communication facilities and distributed shared memory to provide shared variables.

3.1 Global management of the execution

In Chorus/MiX, the unit of distribution, in term of execution and data, is the Unix process. Thus, to be distributed, an Ada program is implemented as several processes which are executed on different nodes. We call an *Ada process*, a process which participates to the execution of an Ada program. To preserve the dynamic aspects of an Ada execution, an Ada program is executed as a process which can be duplicated during its execution. This implies that the Run Time is responsable of the Ada processes management.

3.1.1 Creation of Ada processes

At the begining of the execution, an Ada process is created to support the execution of the first tasks. Then son processes are created, possibly on remote nodes, to distribute the tasks. An Ada process is created by duplication of its father using the Chorus/MiX services for remote execution. Thus, this creation is transparent to distribution. A process is created in the same way to be executed locally or on a remote site: we assume that the load balance, in term of processes, is done by the load management services of the system.
The maximum count of tasks executed in a process is limited to n. When task $n+1$ must be created a new process is created if all the son processes are full: task $n+1$ is executed in this process. Ada processes must then maintain the count of tasks they execute and the count of tasks executed in their son processes.

3.1.2 Identification of an Ada process

To be localized on the network, a process is identified by a *process identifier*. Actually, this process identifier is the identifier of its Run Time communication port, used to exchange messages between Ada processes. The communication ports are entities provided by the Chorus/MiX system: they are queues of messages used in inter process communication. These ports are identified by a Chorus unique identifier managed by the kernel.

3.1.3 Data structures of an Ada process

The notion of Ada process introduces new data structures in the Run Times. A *process descriptor* is composed by a process identifier and by the count of tasks executed in the concerned process. To manage the distribution, each Run Time uses the following informations:

- the parent process descriptor,
- the local process descriptor,
- a list of child process descriptors : this list is used to manage creation and termination issues,
- a Chorus *mutex* to preserve the data coherency : the data of the distributed Run Time can be accessed concurrentely by the threads of the process, indeed. Using the mutual exclusion variable guarantees the coherency of these data.

3.1.4 Communication between Ada processes

The Run Times do not share their data : each Run Time manages its own data. To manage the distributed execution of the whole Ada application, Ada processes must exchange informations on the state of their part of the application. Using Chorus distributed shared memory would be too expensive because the granularity of these data is small. Thus, Ada Run Times use the inter process communication to exchange informations on the application. To implement this communication each Ada process owns a port, called *Run Time port* and a thread, called *Run Time thread.*

3.1.5 Completion of an Ada process

An Ada process must be completed when its execution is terminated. This implies that all tasks of the process and all son processes are terminated. Actually, a process can be completed when all its tasks are completed. A task which has finished its execution, is terminated if all its child tasks are terminated. Moreover, the mother of a task A is executed in the same process than task A or in its father process because of our tree organisation. So, when the last task of an Ada process is terminated, this process does not own any child task nor any active child process.
When a task is completed in an Ada process, the father process must be informed. If this task is the last one of the child process, the father process orders the child to complete.

3.2 Main Ada features

In this part we present our implementation of the main Ada features, using the services of Chorus/MiX.

3.2.1 Tasks

The execution unit in Ada is the task. In the distributed Run Time, *a* task is mapped on *one* Chorus/MiX thread (lightweight process). By using the multi-threaded process management of the micro-kernel, we take benefit of its scheduling. In this case, tasks execution merges thread execution. Moreover, the thread concept is well suited to implement tasks because it provides a small execution context and fast context switching possibilities between two threads.

Task creation
When a task is created in the local Ada process, the Run time creates a Chorus/MiX thread using the Chorus services. This thread will execute the task code by jumping to the starting address of the function.
As the Chorus semantics does not allow to creat theads from one process to another, remote tasks creation can not rely on the system services. Thus, the local Run Time sends a creating request to the target Run Time, using the Chorus IPC. When it receives the request, the Run Time thread of the receiving process executes a local task creation.

Task identification
Task identifiers are used to name tasks. An identifier must be unique for an Ada application to avoid ambiguities, for instance during entry calls. On the other hand, tasks can be executed in different processes which can be localized on different sites. Tasks identifiers are also useful information to access the corresponding task.
A task identifier is composed by the Ada process identifier and by a local identifier which provides the access in the process.

Data structures
The centralized Run Time manages a list of the whole task descriptors for an Ada application. In our implementation, each distributed Run Time manages its own list of task descriptors, dedicated to tasks which are executing in the local Ada process.
To implement tasks management a task decriptor is attached to each task. This descriptor is composed of :

- a task state : active, waiting on rendez-vous, terminated, etc,
- a synchronisation semaphore used to wait for a resource,
- a parent task identifier,
- a list of entry points descriptors : each descriptor contains the state of this entry point and the list of descriptors of the tasks waiting on the concerned entry point.

In the centralized Run Time, all of tasks descriptors are chained. In the distributed Run Times, this is replaced by a list of task identifiers which provides access to tasks descriptors.

End of task
An task ends its execution when all of its child tasks have ended their execution.
To run such an algorithm:

- each task maintains informations to access its child tasks, to know when it ends,
- each task maintains informations to access its parent task to inform it of its end.

3.2.2 Rendez-vous

Implementation

Local rendez-vous are managed in the same way as in standard Run Times. This management is extended to the distributed case by using Chorus IPC between Run Time ports. In each Ada process, the Run Time owns data describing each task and its rendez-vous.

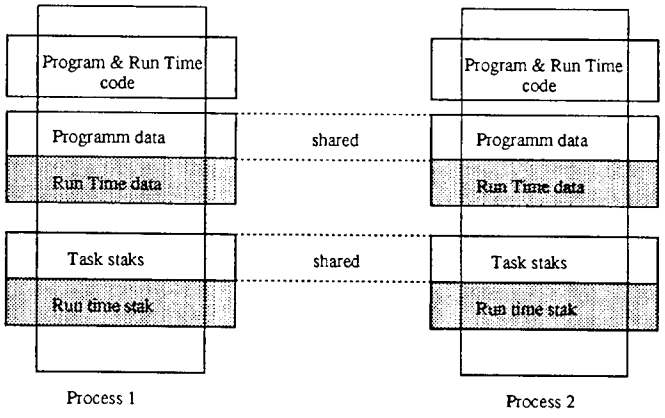


Figure 1: Sharing data between processes

Assuming that a task A calls an entry of a task B, when A asks for the rendez-vous it will be blocked on a semaphore. Then the Run Time thread looks for task B to know if task B is a remote task. If task B is a local task, the entry call is processed in the same way as in the standard Run Time. If task B is remotely executed, the local Run Time sends a message to the remote Run Time to request the access of the entry call of task B for task A. This is done by sending the message on the remote Run Time port. At the reception of an entry call request, a Run Time thread executes the dedicated processing on data attached to task B. This processing is the same than the one realized on a local rendez-vous. Then the modifications on task A data are done by exchanging messages with Run Time A.

Parameter exchange in a rendez-vous

To respect the Ada semantics, entry parameters are not exchanged in the same way than exit parameters.

Entry parameters:

in the local case, entry parameters are copied in the parameters of the called task when processing the *accept* instruction. In the remote case, entry parameters are copied, by the caller Run Time in the entry call message. Then, they are recopied from the message to the called task parameters.

Exit parameters:

Ada semantics distinguishes between two cases, according to the parameters size.

1. *Exit parameters having a size smaller or equal to the integer size :*
   these parameters are exchanged by value. The parameters of the called task are

Table 1: The task creation

```
taskCreate(newId, taskType, parentId)
begin
  if (non complete process)
    localTaskCreate(newId, taskType, parentId)
  else
    remoteTaskCreate(newId, taskType, parentId)
  endif
end
```

```
runTimeCode()
begin
  while TRUE
    waiting for a request
    ...
    if (request == taskCreate)
      localTaskCreate(newId, taskType, parentId)
    endif
  endwhile
fend
```

copied in the parameters of the caller task at the the end of the rendez-vous. In the local case, this copy does not cause any trouble. In the remote case, the parameters are copied in the message, at the the end of the rendez-vous. Then, these parameters are copied from the message to the caller task parameters.

2. *Exit parameters having a size greater than the integer size :*
these parameters are exchanged by address. In the local case, at the *accept* instruction processing time, the called task gets directly the address of the parameters. In the remote case, an other distinction on the size of the parameters is introduced by the distributed Run Time : when the parameters size is greater than a given bound, distributed shared memory services are used to exchange the parameters else they are copied to the message.

The value of parameters is always copied in the message except for the great size parameters. When parameters are composed by addresses, their transmission is transparent to the Run Time : these parameters look like small size parameters.

3.2.3 Sharing data between tasks

The Ada semantics allows to share data between tasks of a same program. This supposes that Ada processes must be able to share data, in particular the data and the stacks of tasks. We did not want to modify the Ada compiler, so this compiler provides an executable file per Ada program. Thus, one of the issues encountered when developping DART was to distribute this executable file on the processes to execute it. This executable file is made of code, data of the program and data of the Run Time, and a stack for each task.
The distributed shared memory services of Chorus are used to share data between tasks (Figure

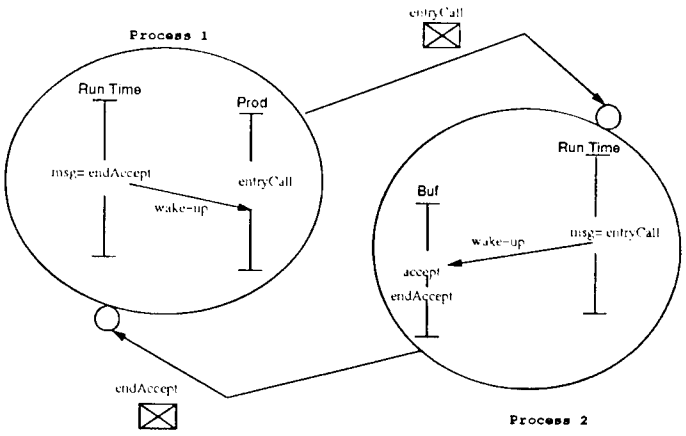


Figure 2: Remote rendez-vous

1). As it is too expensive to use the distributed shared memory services to share data between Run Time, the distributed Run Times uses the Chorus IPC.

3.3 Ada Run Time functions

3.3.1 General structure

At the initialization of an Ada process, a Run Time thread and a Run Time port are created. The Run Time thread executes the Run Time code : most of the time the thread is waiting for a request on the Run Time port. At the reception of a request, the request type is determinated and the corresponding Run Time function is called.
All the distributed Run Time functions are built in the same way:

- if the request can be locally processed, a function of the centralized Run Time is executed,
- else a request is sent to the corresponding remote Run Time, which will execute the function locally.

To illustrate the implementation of the Run Time functions, we present the implementation of a *task creation* and of a *remote rendez-vous*.

3.3.2 Task creation

Table 1 describes the *task creation* function of DART. When this function is called, if the local process is not full, the task is created locally with the *localTaskCreate* function. Else, the *remoteTaskCreate* function is executed. The *localTaskCreate* function creates a Chorus thread in the local process, this thread executes the task code.
The *remoteTaskCreate* function searches first an Ada process in which the new task can be executed : if all the son processes are full, a new process is created. When the destination process is selected, a task creation request is sent to its Run Time. At the request reception,

Table 2: Rendez-vous simulation

Ada Program	Simulation code
`task body buffer is`	bodyBuffer()
`begin`	begin
`  accept readbuf(a:out integer) do`	**accept**(readbuf, a);
`  ...`	...
`  end readbuf;`	**acceptEnd**(a)
`end buffer;`	end
`task body producer is`	
`begin`	
`  buf.readbuf(b);`	**entryCall**(buf, readbuf, b)
`end producer;`	
`prod :  producer;`	**taskCreate**(prod, producer, P)
`buf :  buffer;`	**taskCreate**(buf, buffer, P)
`begin`	**taskActivate**(buf, prod)
`  ...`	
`end;`	

the remote Run Time executes the *localTaskCreate* function.

3.3.3 Remote rendez-vous

Figure 2 presents a sample example of processing for a remote rendez-vous in DART. A *Prod* task calls the *readbuf* entry point of a *Buf* task. We assume that *Prod* and *Buf* are executed in two different processes.
At the entry call, *Prod* fixes that *Buf* is remote. An entry call request is sent to the Run Time port managing *Buf*. When the Run Time thread receives this request, it wakes up the *Buf* task, if it is waiting. Then, this task executes the rendez-vous. At the end of the rendez-vous, *Buf* task notices that *Prod* is remote. It sends a wake-up request to the Run Time managing *Prod* and continues its execution. When the remote Run Time receives the request, it wakes up *Prod* task.

4 Conclusion

DART has been developped on an iPSC/2 multicomputer running Chorus/Mix. This Run Time prototype has been written in C++ language. To test the distributed Run Time, we have simulated the code produced by an Ada compiler by using a program in C language. This simulation code calls DART functions. In table 2, we give an example of a simulation code which implements a sample rendez-vous between two tasks.
In the distributed Ada Run Time prototype, we have implemented the multi-task part of Ada: tasks creation, tasks activation, end of tasks, rendez-vous and select wait. This implementation does not assume any limits to the standard Ada language.
We did not test the distributed Run Time model with a real Ada compiler but with a C simulation. However, this is enough to study the adequation of the Chorus/MiX services to such a project. Thus, with such an implementation, Ada programmers could write their

programs in the same way to be executed on centralized architecture than on multicomputers.

References

[1] Departement of Defense U.S, United States. *Reference manual for the* ADA *programming language*, 1983.

[2] Francois Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or "distributing UNIX brings it back to its original virtues". In *Proc. of Workshop on Experiences with Building Distributed (and Multiprocessor) System"*, pages 153-174, Ft. Lauderdale, Oct. 5-6 1989. USENIX.

[3] Bénédicte Herrmann and Laurent Philippe. UNIX on a multicomputer: the benefits of the CHORUS architecture. In *Transputer's 92*, pages 142–157, Arc et Senans, FR, May 20-22, 1992. IOS Press.

[4] Bénédicte Herrmann and Laurent Philippe. Building a distributed UNIX for multicomputer. In *PACTA'92*, pages 663-671 (Part1), Barcelonna, SP, September, 1992. IOS Press.

[5] Marc Guillemont. CHORUS: a support for distributed and reconfiguration ADA software. In *Communication networks and distributed operating systems within the Space Environment*, pages 263-268, Noordwijk, 24-26 october 1989. ESA workshop.

[6] B.J Dobbing and I.C Caldwell. A pragmatic approach to distributing ADA for transputers. In *Distributed Ada: developments and experiences*, pages 200–221, Southampton, England, 11-12 december 1989. Cambridge University Press.

[7] A D Hutcheon and A J Wellings. The york distributed ADA project. In *Distributed Ada: developments and experiences*, pages 67–104, Southampton, England, 11-12 december 1989. Cambridge University Press.

[8] Rakesh Jha and Greg Eisenhauer. Honeywell distributed ADA: approach. In *Distributed Ada: developments and experiences*, pages 58–66, Southampton, England, 11-12 december 1989. Cambridge University Press.

[9] Colin Atkinson and Andrea Di Maio. From DIADEM to DRAGOON. In *Distributed Ada: developments and experiences*, pages 105–136, Southampton, England, 11-12 december 1989. Cambridge University Press.

[10] Richard A Volz Padmanabhan Krishnan and Ronald Therault. Distributed ADA: a case study. In *Distributed Ada: developments and experiences*, pages 15–57, Southampton, England, 11-12 december 1989. Cambridge University Press.

[11] Judy Bishop Stephen R Adams and David J Pritchard. Distributing concurrent ADA programs by source translation. *Software, Practice and Experience*, 17(12):859-884, december 87.

[12] Russel M Clapp and Trevor Mudge. ADA on hypercube. In *3rd conference on hypercube multiprocessors*, pages 399-408. ACM, january 88.

[13] Russell M Clapp and Trevor Mudge. Distributed ADA on loosely coupled multiprocessor. Technical report, Michigan University, An Arbor, Michigan, january 88.

Monitoring and Debugging in *RAMPA*

V.F.Alexakhin, D.M.Arapov, V.A.Krukov,
V.A.Ivanov, A.K.Petrenko, A.U.Vashakidze
Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya Sq, Moscow 125047, Russia
E-mail: petrenko@d23.keldysh.msk.su (Internet & Bitnet)

Abstract

The debugger of CASE RAMPA for parallel program development is described. Main part of the paper discusses advanced debugging facilities (abstruct events, query language, temporal operations).

Key words: parallel program monitoring and debugging, message passing model, virtual shared memory model.

1 Introduction

RAMPA[1, 2] is a program development system for distributed memory computers. Programming languages it supports uses different models of parallel computation. So in *RAMPA* there are *Norma* for difference methods of computations, *Fortran-DVM* (FDVM) for virtual shared memory model and *Fortran-GNS* (FGNS) for message-oriented model of computation. This languages are translated into standard Fortran-77 enhanced by calls to message-passing library. It also supports C language. *RAMPA* was designed for programmers, researchers, and students in parallel programming.

RAMPA simulates parallel environments under MS DOS, OS/2 and Unix. For mass computations one can use any parallel platform, supporting interprocessor communication. *RAMPA* supports three modes of debugging with running: on the parallel computer itself, on computer (mostly IBM PC clones) which emulates parallel execution and by pseudo-execution of the trace obtained as a byproduct of program execution on parallel computer. In any case only passive debugging is available.

Here we consider debugging facilities for FGNS and FDVM. These languages in distinguish to *Norma* are imperative (procedural) languages. Despite they have essential distinctions in model of parallel running (for short: parallelism model), these languages both are only two dialects of the Fortran. Note, debugging approaches and debugger facilities for Fortran and C are quite similar. The point makes possible to construct unified debugger for both languages. Therefore, we will discuss only Fortran programs debugging below for brevity. RAMPA debugger has simple structure. However it provides with broad set of facilities. Debugger uses trace information. Trace is collected by means of run-time support system. Part of trace processing is carried out in real-time. Nevertheless, main debugger facilities are focussed on program running history (below: history) processing post-mortem.

The debugger is used in the following two modes:

1. Program execution on a parallel computer or its simulator.

2. Program pseudo-execution on an instrumental computer by using the tracing information from the program execution on a parallel computer.

For both modes user may control state of the task and separated processes, statistics of interprocessing communication, browse and analyse history. Besides first mode allow to set breakpoints, watch variables and use other ordinary functions. This work is largely focussed on analysis facilitites.

The monitoring facilities can be treated as an integrated part of debugging tools since they enable programmers to gain a better understanding of his program.

In order to help user to reach better understanding of its parallel program behavior debugger must, on the one hand, support history analysis in terms of sources, and in terms more detailed level (for example, workload of processors), on the other hand. Note, here is some contradiction. So Fortran HPF program does not have any explicit process or synchronization point. However user should understand the location of the program code fragments and schedule scheme for their running. Consequently (implicitly!) processes exist, therefore debugger for low level analyzing and showing of program running must monitor and visualize these processes, must support binding between process state and sources, code location, must account load of processors and so on.

Hence there is a problem of hierarchical representation of any parallel program behavior or history. Top level represents behavior only by means of parallelism model entities of programming language used. Bottom level represents behavior basing on the multiprocessor system entities. This approach is in conformity with the task of unified debugger construction for support of the distinct parallel languages. The involved question will be discussed in the last section of the paper.

Besides ordinary facilities debugger provides following advanced ones:

- synchronized visualization of the trace and corresponding source code fragments ("pseudo-execution" or "pseudo-replay");
- reverse pseudo-replay;
- writing and processing of complex queries. They allow to search, for example, incorrect source code or trace fragments.

For *RAMPA* usage, user creates a lot of auxiliary debugging, testing and measurement material. In particular it consists of queries and templates (see below). It requires considerable effort. Therefore if it is not unreasonable to store this material and reuse it. This is a first question discussed below.

Nondeterministic behavior is one of parallel programs' attribute. One parallel program may generate a variety of serializations. An error of the parallel program may appear in one part of serializations and may be absent in other. For purpose of search of such errors RAMPA debugger builds causal graph on base of trace of some particular serialization. The graph represents some class of serializations which are in some sense equivalent to given. Let's call this class as valid in respect to given serialization. The facilities for valid class analysis are a second item of the work.

High level facilities for history analysis and visualization are strongly connected with parallelism model of parallel programming language used.

So we face with important question: is there any possibility of construction of a unified debugger for distinct parallel languages. It is a third question in the work.

2 Contents of the trace

The debugger facilities described below use tracing information. It is essential to know the trace contents to understand problems under discussed here. Parallel program have been run, trace are stored in one or more files. Each file is sequence of event records. Events may have not any time ordering in general case (although each event record has time stamp, timers of different processors may be dissynchronized). We must keep in mind the problem.

The event type set is based on FGNS run-time system notions. The ones are more low level compared to user level message passing and processor creation operations. For example to "send" operator corresponds two events: "pre-send" and "post-send". Any operator may have a duration. The duration's estimation is difference between post-send and pre-send time.

3 Advanced debugging facilities

There are many approaches to debug parallel programs discussed in the computer literacy [3]. Our debugger uses passive techniques. It does not have influence on the process it spies. When the process runs, the debugger just writes down the trace. Late, when the process ends, the trace could be analyzed. The set of traces' analysis tools is point of concern.

3.1 Events templates

We started from the abstract event approach [4, 5, 6]. It describes parallel program behavior as a partially ordered set of elementary events, such as starts of processes, message sending and receiving, variables' readings and writings. All of these events are atomic and in our model take no time. Elementary events serve as a base for more coarse events, we call them abstract. Rendezvous is an example of the abstract event. It can be coded as a sequence of atomic events:

pre-Send; pre-Receive; post-Receive; post-Send
or
pre-Receive; pre-Send; post-Receive; post-Send

The order of the elementary events belong the abstract one can vary. However there are restrictions imposed by causes and consequences. Say pre-Receive must be before post-Receive and pre-Send must be before post-Receive too. Abstract events can last some time. There may be many concurrent abstract events even in one sequential process.

We use some kind of regular expressions to describe abstract events. Elementary events and already defined abstract events could be used as terms. The terms could be combined using such operations as:

e1;e2 - event e1 ends before event e2 starts.

e1::e2 - events e1 and e2 can occur in any order or can execute concurrently.

e1 , e2 - one of events e1 or e2.

This approach is extended in RAMPA debugger by following:

- templates to describe events' types;
- temporal operators to describe possible timings;
- logical programming to describe trace invariant.

Event's templates describe events' types. Every event could be seen as an object. A template's field contains either a constant (for example, time or message receiver name) or an expression. Expression operands are either constants or reference to other templates' fields. Polymorhic dialog interface to edit templates is available, it adjusts matching rules according to the template type.

Example. Let us find all events, matching the receive any message from process called "Peter". First we will describe event type using a template. The template must contain all the restrictions we mention. The form looks like:

```
type: receive.....
name: .....
taskid: .....
time: .....
from: name = "Peter"
.....
```

3.2 Temporal operations

To describe time dependence we use temporal operators. There are two kinds of the operators. The first kind designed to use in context of one concrete serialization. The second describes order of events in class of serializations. There is a syntax difference between them. Here is an example of template

```
type: .....
name: .....
taskid: .....
time: after(self,exit(name="Bob") and
        before(self,new-task(name="John"))
```

This template describes all events, which took place between the process named "Bob" ended and process named "John" started. "Exit" and "new-task" are event's type's names, and "name" is the field identifier in both types of objects "exit" and "new-task".

3.3 Query language

When we designed the query language we looked for a compromise be-

tween power and flexibility of then language in one side and complexity restrictions in other. We had to choose between SQL, QBE variants and Prolog-like language. The most interesting facility of query language is the ability to write a recursive query. There are many variants of mixing logical programming and relational database query language. We choose Datalog [7] and develop an interpreter from its subset.

Example. Let's find all operators of starting process named "Mary" (many processes can share one name but use distinct taskids), which can occur between sending and receiving of a message addressed to all processes named "Mary". (The presence of such starting operator indicates the common error.) Let's formulate the query:

```
new-task(name="Mary",event=X),
    before(X, send-for-name(to="Mary",event=Y)),
    after(X,receive(from=Y))
```

Variable X is the query result. The system can visualize all the events X. They will be highlighted in the trace window. X can be used in other query also.

4 The ability of unified debugger development

It is obvious that different programming languages require different debugging tools, and conversely, debuggers with different compilers and run-time support systems have different interfaces. We try to design a unified debugger supporting two parallelism models used in FGNS and in FDVM. Let us begin with a brief consideration of these models.

These Fortran dialects represent two widespread approaches to parallel programming. The first one is based on explicit processes description and organization of communication between them by message passing. The second is based on representing algorithms in sequential form and on using additional instructions about data and computations distribution among processors. These instructions enable compiler to automatically translate a sequent program into parallel one, which solves some problems of applied software development for parallel systems. Parallelism models description used in FGNS and in FDVM can be found in [1, 2].

In spite of apparent difference between the approaches to these two dialects implementation they have common relational data base (synchronization and message passing techniques) and common problems (for example,

efficiency analysis, workload balance, valid serialization classes analysis, etc.). It allows to set a task of unified debugger development for debugging programs in FGNS and in FDVM.

Let us list objects and features which the user observes and manages working with message passing models as well as with DVM model. The main are the following:

- processors (current state, workload statistics);
- processes location on processors;
- causal links between processors, their realization in particular serialization and in a valid serialization class (see below);
- variables values;
- program sources (related to current time, processes and processors).

This list is possibly not exhaustive. However, it requires some comment.

At first sight some of the listed notions, for example, "processes" (as program units), are absent in FDVM. At the same time real debug demands and, in particular, performance debugging cause the necessity of such notion introduction. In practice the difference between parallel program representation in FGNS and in FDVM for programmer is that, in the first case, process corresponds to some program unit and a tree of called subroutines, and in the second case, process is a code fragment recognized by compiler automatically or by additional user instructions. Moreover, even in case of FGNS processes numeration (identification) system (taskid) is rather complicated, and when designing user interface we are forced to provide facilities for referring to desired processes not by name but other, indirect, means.

Thus, a unified interface of RAMPA debugger with compiler and run-time support system is made up of language dependent and independent parts. Language independent part covers facilities for accessing the objects listed above.

Language dependent part of FDVM covers facilities for:

- interrelational visualization of source code fragments and processes;
- relating processes to events in history;
- data location map ("variable-processor" map).

Summarizing this section of the paper, we think that for practical programming in the manner supposed by Fortran HPF it is desired to provide the user with facilities for explicit monitoring and control of parallel processes (what exactly is done in FDVM).

This work is supported in part by the grant of Russian Fundamental Researches Foundation, No. 93-012-628.

References

[1] V.A.Krukov, L.A.Pozdnjakov, I.B.Zadykhailo, RAMPA - CASE for portable parallel programs development.- Proceedings of the International Conference "Parallel Computing Technologies", Obninck,Russia, Aug 30-Sept 4, 1993.

[2] N.A.Konovalov, V.A.Krukov, S.N.Mihailov, A.A.Pogrebtsov. Fortran DVM – language for portable parallel programs development.- in the Proceedings.

[3] A.Petrenko. Parallel program monitoring and debugging.- Programmirovanie, No. 3, 1994. (English version are being published in Software and Programming Systems, N-Y).

[4] C. Kilpatrick, K.Schwan. ChaosMON - Application-Specific Monitoring and Display of Performance Information for Parallel Distributed Sysytems.- ACM Sigplan Notices, V.26, N. 12, December, 1991, pp. 57-67.

[5] J.Stone. A Graph Representation of Concurent Processes.- ACM Sigplan Notices, V.24, N. 1, January, 1989, pp. 226-235.

[6] P.Bates. Debugging Heterogeneas Distriduted Systems Using Event-Based Models of Behaviar.- ACM Sigplan Notices, V.24, N. 1, January, 1989, pp. 11-22.

[7] S.Ceri, G.Gottlob, L.Tanka. Logic Programming and Databases.- Springer-Verlag Berlin Niedelberg, 1990.

Monitoring Parallel Programs for Detecting Access Anomalies Occurred First

Yong-Kee Jun

Dept. of Computer Science
Gyeongsang National University
Chinju, 660-701, Korea

Kern Koh

Dept. of Computer Science & Statistics
Seoul National University
Seoul, 151-742, Korea

Summary

Detecting access anomalies, also called data races, is important for debugging shared-memory parallel programs, since the anomalies result in unintended nondeterministic executions of the programs. Previous monitoring techniques to detect access anomalies on the fly can not guarantee that an anomaly *occurred* first is an anomaly *detected* first. This paper presents *the first monitoring technique* to detect access anomalies occurred first in shared-memory parallel programs that have single-level parallelism and no inter-thread coordination. Detecting anomalies occurred first is important in debugging, since the removal of the anomalies occurred first may make the first detected anomalies disappear. Therefore, this technique makes on-the-fly anomaly detection more effective and practical in debugging a large class of shared-memory parallel programs.

Keywords: shared-memory parallel program, debugging, access anomaly or data race, execution monitoring, effectiveness

1 Introduction

One of the major kinds of bugs in shared-memory parallel programs is the instructions accessing a shared variable in a set of parallel threads, which include at least one write-access to the variable without coordinations. Such a kind of bugs, called *data race* or *access anomaly*, results in unintended nondeterministic executions of debugged programs, and then makes debugging parallel programs difficult. It is ineffective to use traditional breakpoints for detecting access anomalies in parallel programs, since the breakpoints can make the execution timing interfered and then may make erroneous behaviors disappear [1].

On-the-fly detection [2, 3] augments debugged program and monitors an execution of the program to detect actual access anomalies which have occurred during a particular execution. The approach can be a complement to *static analysis* [4], since it guarantees to isolate real anomalies from the potential anomalies reported by static analysis in a particular execution instance of the program. On-the-fly detection requires still less storage space than *post-mortem detection* [5] since much information can be discarded as an execution progresses, but is yet expensive in time and space to check accesses to all shared variables at run time. Although on-the-fly detection may not report as many anomalies as the post-mortem detection does, at least one anomaly is guaranteed to be reported on the fly for each variable which is accessed anomalously. The anomaly occurred first in a thread, however, may not be the first anomaly

detected in the thread. The anomalies occurred first is important in debugging, since the removal of the anomalies occurred first may make the first detected anomalies disappear. It is not effective to use previous on-the-fly detection technique repeatedly for detecting the anomalies occurred first in parallel programs, since the cost of monitoring a particular execution is still expensive.

This paper presents the first monitoring technique that guarantees detecting on the fly all of the access anomalies occurred first that appeared in a particular execution of debugged program. We assume that the parallel programs may have single-level parallelism such as parallel loop constructs. The programs have neither causes of nondeterministic executions except access anomalies, nor synchronization operations such as events or semaphores except the operations to fork or join the parallel control threads.

Section 2 describes definitions and notations to be used through the rest of this paper, and discusses previous monitoring techniques which can not guarantee that an anomaly occurred first is an anomaly detected first. Our study is motivated by the drawback of the previous techniques. Section 3 determines a set of assertions on anomalies occurred first, introduces a set of monitoring data structures based on the assertions, and then presents new monitoring technique, called *FOASET detection protocol*. Section 4 introduces a related work and analyzes our technique. We conclude our discussion in section 5.

2 Background

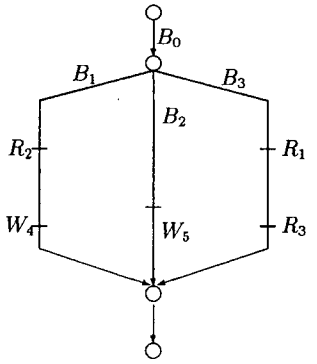
In this section, we describe access anomalies with a static view of debugged programs and its dynamic view represented by directed acyclic graphs. Then, we discuss previous monitoring techniques, which can not guarantee that the anomaly occurred first is the anomaly detected first. Our study is motivated by the drawback of previous techniques.

2.1 Access Anomaly

In this work, we consider parallel programs with single-level parallelism and no inter-thread coordination, which are expressed with parallel loops in this paper. Any kinds of parallel loops can be considered, so that they can be expressed by a single notation, `PARALLEL DO`. In an execution of a parallel loop program, multiple execution threads of control can be created and terminated at a closed construct, `PARALLEL DO` and `END DO`. More than one thread can be *forked* at a `PARALLEL DO` statement, and *joined* at the corresponding `END DO` statement. Such fork and join operations are called *thread operations*. Figure 1.a shows a parallel program with a parallel loop indexed with `I` using the increment of one.

The concurrency relationship among threads is represented by a directed acyclic graph, called *Partial Order Execution Graph (POEG)* [2]. A vertex of POEG represents a thread operation, and an arc started from a vertex represents a thread started from the corresponding thread operation. Since the graph captures the *happened-before* relationship [6], it presents a partial order on a set of the events, executed by threads, that make up an execution instance of a parallel loop. Figure 1.b shows a POEG that is an execution instance of a parallel program shown in Figure 1.a, where a dash on a thread represents an event executed by the thread. Concurrency determination is not dependent on the number and the relative execution speeds of processors executing the program. An event e_i *happened before* an event e_j, if there exists a path from e_i to e_j in a graph. e_i *is concurrent with* e_j, if and only if there exist no paths

```
    ...
PARALLEL DO I = 1, 3
    ...
    IF ··· = X
    ...
    IF ··· = X
    ...
    IF ··· X = ···
    ...
END DO
    ...
```



(a) (b)

Figure 1: A parallel program and its POEG

between them. The *maximum concurrency* of a program execution is defined as the maximum number of mutually concurrent threads. For example, consider events appeared in Figure 1.b, where R_i and W_i denote a read and a write access, respectively, and i means the accesses are executed in that order. R_2 happened before W_4, since there exists a path from R_2 to W_4. R_2 is concurrent with W_5, since there exist no paths between them. The maximum concurrency of the graph is three.

Two accesses to the same variable are *conflicting* if at least one of them is a write. An *access anomaly* exists when two or more concurrent threads perform conflicting accesses to the same shared variable in an execution of a program. If two accesses e_i and e_j consist of an anomaly, the anomaly is denoted by e_i-e_j. An *anomalous event* is defined as the access that consists of an access anomaly. Given two accesses e_i and e_j, e_j *is affected by* e_i, if e_i happened before e_j *and* e_i is an anomalous event. Note that e_j is not affected by e_i, if e_i is concurrent with e_j *or* e_i is not an anomalous event. An anomalous event that is not affected by any event consists an access anomaly that occurred first. The set of such anomalies in a program is called *First Occurred Anomalies SET (FOASET)* that occurrs in the same execution. We define an *affected anomaly* as an anomaly that is not in FOASET. Consider the accesses to a shared variable X in the POEG in Figure 1. There exist six anomalies in the execution; R_1-W_4, R_1-W_5, R_2-W_5, R_3-W_4, R_3-W_5, and W_4-W_5. All accesses in the POEG, therefore, are anomalous events. Among these, only two anomalies, R_1-W_5, R_2-W_5, are in FOASET, since R_3 and W_4 are affected by R_1 and R_2, respectively. The removal of the two anomalies occurred first, therefore, may make some or all of the remaining four anomalies that is affected disappear.

2.2 On-the-fly Anomaly Detection

On-the-fly detection technique is a method for run-time detection of access anomalies. To detect access anomalies in run time, each potentially anomalous accesses to a shared variable must

be monitored during an execution of a parallel program. The existence of an access anomaly involving a shared variable, therefore, is solely a function of which events access it and the concurrency relationship between the events, so that we can consider access anomalies for each shared variables independently.

In on-the-fly detection, each thread is associated with concurrency information, called a *label*, that is generated on each thread operation by a *labeling algorithm*. Labeling is still simple for programs we consider in this paper. The label of each thread can be a loop index of the thread, since the program has single-level parallelism and no coordination. An access is concurrent with another access, if and only if the labels of threads that have executed two accesses are not same. Each shared variable is associated with an *access history* that is maintained to monitor each access by an *anomaly detection protocol*. The access history for a shared variable X is a set of accesses represented by labels of the threads which have accessed X. $AH_R(X)$ and $AH_W(X)$ denotes the reader set and the writer set of the access history, respectively. Whenever X is read or written, the access history is examined to determine if the current access is anomalous with previous accesses. A previous read in $AH_R(X)$ is deleted, if a current read is not concurrent with the previous one. $AH_R(X)$ contains, therefore, labels which are mutually concurrent. On the other hand, $AH_W(X)$ contains at most one label, since two concurrent writes always conflict. By deleting obsolete entries, the size of an access history is bounded by the maximum concurrency of the monitored execution, and at least one anomaly is guaranteed to be reported for every variable which is accessed anomalously. However, if there exist multiple affected anomalies involving the same variable, the anomaly occurred first is not quaranteed to be the anomaly detected first, since certain information may be lost by the compaction of access history. Our study is motivated by this drawback of previous techniques.

Consider on-the-fly anomaly detection for the execution instance shown in Figure 1.b. After R_1, $AH_R(X)$ contains R_1. The next read R_2 is added to $AH_R(X)$, since R_1 is concurrent with R_2. When R_3 is the next, however, R_3 is added to $AH_R(X)$ and R_1 is deleted, since R_3 is not concurrent with R_1. The next W_4 is concurrent with R_3, so that R_3-W_4 are reported as a detected anomaly, and then W_4 is saved into $AH_W(X)$. At last, W_5 examines both $AH_R(X)$ and $AH_W(X)$, and reports the detection of three anomalies; R_2-W_5, R_3-W_5, and W_4-W_5. The number of detected anomalies by this on-the-fly detection, therefore, is four only among six anomalies occurred in this execution. To make the matters worse, the anomalies reported by the detection include only one anomalies occurred first, R_2-W_5, among four anomalies reported.

3 On-the-fly FOASET Detection

In this section, we examine the partial order on anomalous events appearing in execution of parallel programs that we consider, and determine a set of assertions on such events, called *candidate*, that can consist of the anomalies that occurred first in each thread. Based on the assertions, we introduce a set of monitoring data structues that is necessary to collect the candidates during an execution of debugged program. Finally, we present new monitoring algorithm, called *FOASET detection protocol*, using the data structures.

3.1 FOASET Assertions

An access anomaly in the execution of a program exists when two or more concurrent threads perform conflicting accesses to the same shared variable. In execution of the program we

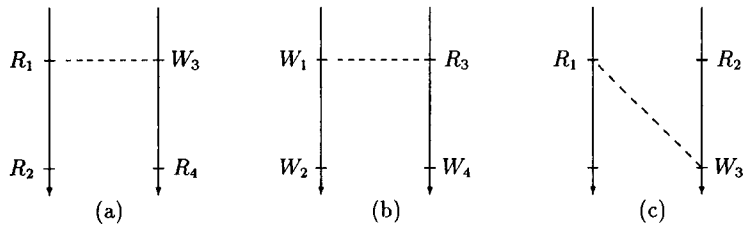


Figure 2: Examples of FOASET

consider, there can exist at most one candidate in a thread. By definition, a candidate is an anomalous event that is not affected, so that there exist no anomalous events that happened before the candidate in the thread. The other accesses occurred in the thread are either affected by the candidate or not anomalous. To determine if each access is a candidate, therefore, we need to investigate it with each type of accessing orders in a thread. At most four types of orders can appear in a thread;

1. a read R_i happened before another read R_j,
2. a read happened before a write,
3. a write W_i happened before another write W_j, and
4. a write happened before a read.

Figure 2 shows three examples, each of which has a POEG representing an execution of two parallel threads.

Consider a read access. In type 1, the candidate is R_i, since R_i is also anomalous if R_j is anomalous. Figure 2.a shows an example of this type. In this example, there exists a write that is concurrent with both R_1 and R_2. These two reads are anomalous with the write, so that the candidate is R_1 since R_1 happened before R_2. Therefore, we can represent a series of reads in a thread by the read that occurred first in the same thread. In type 2, the read is a candidate, if there exists a write that is concurrent with the read. Figure 2.b shows an example of this type. In this example, there exist two writes that are concurrent with both R_3 and W_4. Both R_3 and W_4 are anomalous, so that the candidate is R_3 since R_3 happened before W_4. Figure 2.c shows another example of type 2. In this example, there exists one read that is concurrent with both R_2 and W_3. Because anomalous is not R_2 but W_3, the candidate is not R_2 but W_3, although R_2 occurred first in the thread. A read that occurs first in its thread, therefore, is not always a candidate. In type 4, the read is not candidate, since there can exist an event that is concurrent with the write. Figure 2.a shows an example of this type. In this example, there exist read events that are concurrent with both W_3 and R_4. Because only W_3 is anomalous and W_3 happened before R_4, the candidate is not R_4 but W_3. Therefore, a read is not a candidate if an event happened before it.

Assertion 1 *A read event is a candidate, if and only if it occurs first in a thread and is anomalous.*

Consider a write access. In type 2, the write can be a candidate, since there cannot exist any writes that are concurrent with the read. Figure 2.b shows an example of this type. In this example, there exist two writes that are concurrent with both R_3 and W_4. Both R_3 and W_4 are anomalous, so that the candidate is not W_4 but R_3 since R_3 happened before W_4. Figure 2.c shows an example of type 2. In this example, there exists one read that is concurrent with both R_2 and W_3. Because anomalous is not R_2 but W_3, the candidate is not R_2 by Assertion 1 but W_3 that happened before R_2. Therefore, a write that does not occur first in its thread can be a candidate. In type 3, the candidate is W_1. Figure 2.b shows an example of this type. In this example, there exists two events that are concurrent with both W_1 and W_2. Therefore, these two writes are anomalous, so that the candidate is W_1 since W_1 happened before W_2. Therefore, we can represent a series of writes in a thread by the write that occurred first in the same thread. In type 4, the write is candidate, since there can exist an access that is concurrent with the write. Figure 2.a shows an example of this type. In this example, there exist read events that are concurrent with both W_3 and R_4. Because only W_3 is anomalous and W_3 happened before R_4, the candidate is not R_4 by Assertion 1 but W_3. Therefore, a write that occurred first in the thread is a candidate.

Assertion 2 *A write event is a candidate, if and only if it occurs first in a thread or a read that is not anomalous happened before it in the same thread.*

Candidate events consist of the anomalies in FOASET if the events include at least one write. Therefore, detecting FOASET is collecting candidates in an execution of parallel program.

3.2 Monitoring Data Structures

To detect FOASET on the fly, each access to a shared variable must be monitored during an execution of debugged program to decide if it is a candidate. From FOASET assertions, such a decision for the current access requires a set of checks with each previous access.

If a current access is a read R_c, we must check the two candidate conditions of Assertion 1 in which the candidate read (1) occurs first in a thread and (2) is anomalous. For the first condition of Assertion 1, the previous accesses must be checked to determine if R_c occurs first in its thread, so that they are required to have been saved. For the second candidate condition, R_c must be anomalous, which implies the existence of a previous candidate write that is concurrent with R_c. To check the existence on the fly, such previous writes are required to have been saved in a data structure. We call the data structure, denoted by AH_W, an access history for write accesses. If there exists no such a previous write that is concurrent with R_c, the R_c must be saved for a write that may appear hereafter and determine if the write is a candidate. For this case, a data structure is required to save R_c. We call the data structure, denoted by AH_R, an access history for read accesses. Therefore, monitoring reads requires two types of access histories to maintain each type of conflicting accesses.

If a current access is a write W_c, we must check the two candidate conditions of Assertion 2 in which a current write (1) occurs first in a thread or (2) a read R that is not anomalous happened before it in the same thread. For the first candidate condition, W_c must occur first in its thread, which can be determined by checking AH_R and AH_W. For the second condition of Assertion 2, it must be checked if there exist no candidate write that is concurrent with both W_c and R. If there exists no such a candidate write in AH_W at that time, W_c must be saved for the possibility of such candidate write that may appear hereafter. If the candidate write

does not appear at the end of the program execution, W_c becomes a candidate. Otherwise, W_c must not be considered as a candidate. For this case, W_c can not be saved in AH_W, since W_c must be deleted when a candidate write that is concurrent with W_c appears. This type of the write before which a read happened is called a *read-write event*. Another type of data structure is required to save such a current read-write event. We call the data structure, denoted by AH_RW, an access history for read-write accesses.

Therefore, the monitoring technique to detect FOASET requires three types of data structures for each shared variable X;

- an access history for read accesses ($AH_R(X)$),
- an access history for write accesses ($AH_W(X)$), and
- an access history for read-write accesses ($AH_RW(X)$).

3.3 FOASET Detection Protocol

On-the-fly FOASET detection is collecting and reporting every conflicting pair of candidates during an execution of debugged program. Collecting candidates is accomplished by cooperations among parallel accesses. Each access checks the candidate condition of its FOASET assertion. If the condition is satisfied, the candidate is saved in the corresponding access history after reporting first occurred anomalies that can be detected only currently. There are two protocol functions for the two types of conflicting accesses, called **checkread()** and **checkwrite()**. The protocol consists of three phases in each access;

1. determining if current access is a candidate,
2. reporting first occurred anomalies made by the candidate, and
3. saving the candidate in the corresponding access history, and clearing information that is no more necessary.

The FOASET detection protocol is shown in Figure 3, where *current* is the current event accessing a shared variable X, and $ordered(a, current)$ is a boolean function that satisfies a happened before *current*.

In each read, **checkread()** is performed. During the first phase, *current* is compared concurrency relationship with the previous events in both $AH_R(X)$ and $AH_W(X)$. If *current* is ordered with an event in the access histories, it does not occur first in the thread or is not anomalous so that it is not a candidate by Assertion 1. Then, the protocol is terminated by **return**. (line 1-2) During the second phase, it reports first occurred anomalies. If *current* is a candidate, every first occurred anomaly including *current* can be reported into $WR_FOASET(X)$ and $RWR_FOASET(X)$ respectively. (line 3-4) During the third phase, the *current* is saved in the corresponding access history to used by accesses appeared hereafter. (line 5)

In each write, **checkwrite()** is performed. During the first phase, *current* is compared concurrency relationship with the previous events in $AH_R(X)$, $AH_W(X)$, and $AH_RW(X)$. (line 8-13) If *current* is not ordered with an event in these access histories, it occurs first in the thread, and then *current* is a candidate by Assertion 2. If *current* is ordered with a candidate in $AH_W(X)$ and $AH_RW(X)$, the protocol is terminated by **return**, since *current* is not a candidate by Assertion 2. (line 8-9) If *current* is ordered with a candidate R in $AH_R(X)$,

```
0  checkread(X, current)
1       for all a ∈ AH_R(X) ∪ AH_W(X) do
2            if ordered(a, current) then return;
3       for all a ∈ AH_W(X) ∪ AH_RW(X) do
4            add <a, current> to WR or RWR_FOASET(X);
5       add current to AH_R(X);
6  end checkread

7  checkwrite(X, current)
8       for all a ∈ AH_W(X) ∪ AH_RW(X) do
9            if ordered(a, current) then return;
10      for all a ∈ AH_R(X) do
11           if ordered(a, current) ∧ AH_W(X) ≠ ∅ then return;
12           if ordered(a, current) ∧ AH_W(X) = ∅ then move a to rw;
13      endfor
14      if rw = not null then
15           for all a ∈ AH_R(X) do
16                add <a, current> to RRW_FOASET(X);
17           add current to AH_RW(X);
18           move rw to AH_R(X);
19      else
20           for all a ∈ AH_W(X) ∪ AH_R(X) do
21                add <a, current> to WW or RW_FOASET(X);
22           add current to AH_W(X);
23           clear AH_RW(X);
24           clear RRW_FOASET(X) ∪ RWR_FOASET(X);
25      endif
26 end checkwrite
```

Figure 3: FOASET Detection Protocol

current is a read-write event because R happened before it. So, checks must be done for the existence of a write candidate in $AH_W(X)$ that is concurrent with both R and *current*. (line 10-11) Such checks are as simple as checking if $AH_W(X)$ is empty, since the concurrency relationship was already checked with *current* in line 8-9. If $AH_W(X)$ is empty, the protocol is terminated by **return** since Assertion 2 is violated in this step. (line 11) Otherwise, *current* is a candidate read-write. In this case, we move R from $AH_R(X)$ to a temporary location, called *rw*, to simplify the next phase that is to be performed for this read-write candidate, *current*. (line 12) During the second phase, it reports first occurred anomalies since *current* is a candidate in this step. If *rw* is empty, then *current* is a candidate write. Otherwise, *current* is a candidate read-write. If *current* is a candidate read-write, every first occurred anomaly including *current* is reported into $RRW_FOASET(X)$ with every pair of *current* and a candidate in $AH_R(X)$. (line 15-16) If *current* is a candidate write, every first occurred anomaly can be reported into $WW_FOASET(X)$ and $RW_FOASET(X)$ respectively with every pair of *current* and a candidate in $AH_W(X)$ and $AH_R(X)$. (line 20-21) During the third phase, *current* is saved in the corresponding access history, and information that is no

more necessary is cleared. Saving *current* is necessary for the potentially forthcoming accesses to detect first occurred anomalies including them. (line 17, 22) Clearing for the read-write candidate is simpler than that for the write candidate. If *current* is a read-write candidate, *rw* is moved to $AH_R(X)$ to be used for monitoring the potentially forthcoming accesses. (line 18) If *current* is a write candidate, the writes in $AH_RW(X)$ violates Assertion 2. Therefore, these accesses and reported anomalies including them must be cleared, since the accesses is not candidate at this step. (line 23-24)

As an example of on-the-fly FOASET detection, consider the program execution in Figure 1. When R_1 and R_2 are performed, these are saved in $AH_R(X)$ without reporting. The next access R_3 is returned, since R_1 happened before it. W_4 is a read-write candidate, because R_2 happened before it and $AH_W(X)$ is empty at that time. Therefore, the pair of W_4 and R_1 that is in $AH_R(X)$ consists of a first occurred anomaly, so that the pair is reported to $RRW_FOASET(X)$. The next access W_5 is a write candidate, because it occurs first in its thread. Since there exist two candidates R_1 and R_2 in $AH_R(X)$ and no candidate in $AH_W(X)$, W_5 consists of two access anomalies occurred first, W_5-R_1 and W_5-R_2, which are reported to $RW_FOASET(X)$. Finally, $AH_RW(X)$ and $RRW_FOASET(X)$ are cleared. Therefore, $RW_FOASET(X)$ is the result of on-the-fly FOASET detection for the parallel program in Figure 1.

4 Related Work and Analysis

Choi and Min [7] propose a method for reexecuting monitored program to reproduce undetected anomalies. To address the problem of the undetected anomalies, they show how to guarantee deterministic reexecution of the program up to the point of the first detected anomalies, allowing additional instrumentation to be added that can locate the originally undetected anomalies. The effect of the method is the same as reproducing complete anomaly histories from the abridged anomaly history collected during program execution, allowing *cyclical debugging* approach such as breakpoints or trace-based replay to be applied during reexecution. Such reproducing and debugging eventually results in a program without any access anomalies. This method, therefore, requires additional programmer's task to detect first occurred anomalies in debugged program.

Our new monitoring technique guarantees run-time detection of first occurred anomalies without requiring subsequent reexecutions of the program. This technique requires a reasonable amount of monitoring cost. Since all the accesses that must be kept in each access history for a shared variable are mutually concurrent, the bound on the number of entries in an access history is the maximum concurrency in the monitored execution of the debugged program. Therefore, our technique requires the same worst-case complexity of space and time that is with previous monitoring techniques. These techniques, however, can not guarantee that the anomaly *occurred* first is the anomaly *detected* first.

5 Conclusion

This paper presents the first monitoring technique for run-time detection of access anomalies occurred first in shared-memory parallel programs with single-level parallelism. The technique is focused on the monitoring protocol performed in each access, which determines if the access is

a candidate, reports each occurrence of first occurred anomaly, and maintains the corresponding access history. The technique, therefore, makes on-the-fly anomaly detection more effective and practical in debugging a large class of shared-memory parallel programs, since the removal of the anomalies that occurred first may make the first detected anomalies disappear. We are implementing and extending the mechanism to synchronizable and nestable parallelism along with NR Labeling algorithm [3].

Acknowledgement

We wish to thank Dong-Gook Kim and Jin-Sook Lee for their useful comments. This work was supported by Korea Science and Engineering Foundation.

References

[1] McDowell, C. E., and D. P. Helmbold, *"Debugging Concurrent Programs,"* Computing Surveys, 21(4): 593-622, ACM, Dec. 1989.

[2] Dinning, A., and E. Schonberg, *"An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection,"* 2nd Symp. on Principles and Practice of Parallel Programming, pp. 1-10, ACM, March 1990.

[3] Jun, Y., and K. Koh, *"On-the-fly Detection of Access Anomalies in Nested Parallel Loops,"* 3rd Workshop on Parallel and Distributed Debugging, pp. 107-117, ACM, May 1993.

[4] Emrath, P. A. and D. A. Padua, *"Automatic Detection of Nondeterminacy in Parallel Programs,"* 1st Workshop on Parallel and Distributed Debugging, pp. 89-99, ACM, May 1988.

[5] Emrath, P. A., S. Ghosh, and D. A. Padua, *"Detecting Nondeterminacy in Parallel Programs,"* Software, 9(1): 69-77, IEEE, Jan. 1992.

[6] Lamport, L., *"Time, Clocks, and the Ordering of Events in Distributed System,"* Communications of ACM, 21(7): 558-565, ACM, July 1978.

[7] Choi, J., nad S. L. Min, *"Race Frontier: Reproducing Data Races in Parallel-Program Debugging,"* 3rd Symp. on Principles and Practice of Parallel Programming, pp. 145-154, ACM, April 1991.

An Approach to the Scalable Software Development with QUICKPLAY System

Alexander Biriukov, Victor Meliokhin
Computing Centre of the Russian Ac.of Sci., 40, Vavilova Str., 117967 Moscow, Russia
email: bir@sms.ccas.msk.su

Abstract. A software system that allows to easily implement an application from a class of apllications on the different transputer networks and the class itself are described. The resulting code does not use universal routing facilities. Instead, the original program text is automatically modified and simple multiplexers are generated and inserted into the code so as not to cause any additional overhead.

1. Introduction

Efficient programming for the distributed memory computers of which the transputer networks are the common example is widely recognised to be much harder than that for the vector and shared memory computers. The approach based on the manual specification of all communications in the program, although providing high performance, can only be applied to smaller applications and networks with a fixed architecture thus decreasing the scalability of the resulting code. Common solutions, such as routers, significantly facilitate programming, but can cause hardly predictable overhead over the application performance.

The main reason for the universal routers' inefficiency is related to their principal inability to account for the specific communication patterns of a particular application. Some authors ([1],[2]) studied the possibility for the user to implement application specific routers instead of using any universal harness.

It appears, on the other hand, that a wide class of applications exists that can be effectively implemented on the transputer networks without exploiting general and time-consuming routing facilities. The primary objective of the QUICKPLAY (QUICK Procedure LAYout) project described below is to provide a user with a software tool allowing development of truly scalable parallel applications that work without the universal router.

The rest of the paper is organised as follows. In Section II the overview of the project is given. The principal aim of this Section is to introduce the main notions the system is based upon and justify the approach itself. In Section III the main components of the system are described. Experimental results are presented in Section IV. Section V concludes the paper and outlines some directions of the future work.

2. Basic Notions

2.1 The Application Model

The class of applications we are interested in comprises programs where the procedures called are the most time consuming parts of the code. We use the term **procedure** as the general programming concept instead of the language specific 'subroutines', 'functions' etc. although with C functions in mind, as QUICKPLAY is currently

based on 3L C language. An example of the numerical algorithm, that can be effectively parallelised using this idea, can be found in [3].

We assume there is so-called **parallel library** developed either *ad hoc* or of a common use, which containes parallel procedures in the special format not described here. The only principal point is that the library contains not object modules, but the source texts of the parallel procedures. We also assume that each parallel procedure may consist of an arbitrary number of **components**. The components, unlike **threads**, are executed on different processors, each component may comprise any number of threads. The communications between the components of the procedure are described manually. Any thread of any component, in turn , may contain any number of procedure calls.

QUICKPLAY's strategy of parallelising the program is, having started from the source text, to recursively look through it and those of procedures called, and replace the sequential procedures with their parallel implementations taken out of the library.

2.2 Static vs. Dynamic Code Distribution

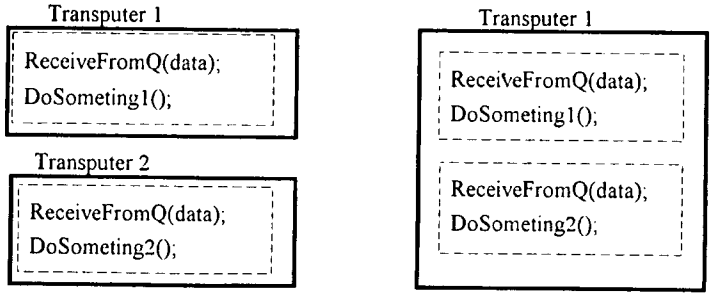
Given a user program containing procedure calls and a library of parallel procedures, a straightforward solution would be to load components dynamically once the corresponding procedure is called. As this strategy implies significant overhead on the application performance, we have chosen another approach. QUICKPLAY analyses the texts of the applicaion and the procedures called and generates a group of modules, each module consisting of several components, one module for each transputer in the network. To do this it is first necessary to specify for each component the transputer it should be placed on. We call the correspondence between the components and the transputers the **layout**. It appears, however, that different layouts can cause different overheads, as Fig.1 shows, where two components are put on the two (a) and one (b) transputers.

If the components appear to go in parallel, it can be easily seen from Fig.1, that (a) does not introduce any additional overhead, while in (b), where the components work concurrently as threads, they slow each other down due to the internal transputer time sharing.

Let, for example, DoSomething1 take time *t1*, DoSomething2 time *t2* and data from Q arrive at the components at the same moment *t0*. In the first case DoSomething1 terminates at the moment *t0+t1* and DoSomething2 at the moment *t0+t2*, while in the second case they will both terminate at the moment *t0+t1+t2*.

A good idea would be to reserve a distinct transputer for each component. This solution, however, not only requires lots of processors, but restricts the number of procedures called in parallel with the number of links on the transputer, otherwise making it necessary to use a router. The latter is exactly what we would desire to avoid.

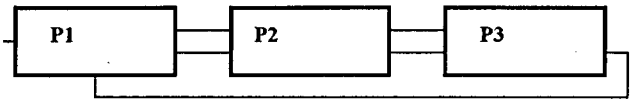
QUICKPLAY makes a compromise allowing only those components to occupy the same transputer that will never work in parallel. This means that none of them will be



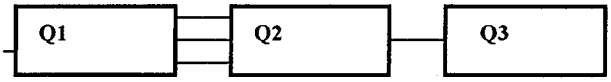
a) Two components on the different transputers

b) Two components on the same transputer

Fig.1 Two different layouts



The Transputer network for the procedure P



The Transputer network for the procedure Q

Fig.2 The incompatible procedures

allowed to share time slowing down each other as is the case in Fig.1b. It should be noticed that the different components of the same procedure are always considered to be working in parallel by default (to keep the procedure truly parallel as it originally was). This implies a certain number of processors to be necessary to implement given application.

But what if there are not enough processors in the network? One possible solution would be to attempt to keep parallel components separated until it is possible and then somehow put all the rest on spare processors. This might cause, however, unpredictable overhead that we again would like to avoid.

It should be noted here that not only the number of transputers available is critical for the layout to exist, but also the restricted number of links per transputer. Let, for example, two procedures, P and Q, be implemented on two different transputer networks as shown in Fig.2. It can be easily seen that the procedures cannot be put simultaneously on any transputer network consisting of less than five transputers.

To solve this problem QUICKPLAY introduces the notion of the **version** of the parallel procedure. An important consequence of this solution is the possibility for the user to almost automatically develop scalable applications, dedicated to the particular transputer architectures.

2.3 Versions

Each parallel procedure in the library may have any number of versions. The versions have the same functionality, but are tailored for different networks, e.g. may comprise different numbers of components. Normally, the versions have different performances depending not only on the number of transputers used, but also on the whole context of their calls. By the context we mean parameter values, global data values etc.

The problem consists in choosing the best set of versions to achieve the highest performance of the given application. As not only the user program itself calls procedures, but some of them may also contain further procedure calls, this set includes versions of all procedures called from anywhere. The resulting set of versions, actually, looks like a **caller-callee tree** with the nodes representing the versions of the procedures called and the edges denoting procedure calls. The root of the tree represents the user program.

As the experiments have shown that the best tree can hardly be deduced automatically from the texts of the source program and procedures called, QUICKPLAY provides the user with an opportunity to manually build the tree. This implies, that the user must know the structure of the library to make his/her choice reasonably. If the tree that is the best one from the user's viewpoint cannot be implemented on a given network (e.g. due to the shortage of transputers or spare links), the user can select another set of versions and try the resulting code fairly quickly.

The key point is that the original application need not be manually adjusted to the particular network architecture, since QUICKPLAY itself automatically modifies the texts when building the resulting code.

2.4 Some estimations

Before going further some simple estimations should be given.

The most interesting question is how many processors are really necessary to implement given application, following the idea described above. Unfortunately, this number appears to be strongly dependent on the application structure, the structures of the procedures called etc.

Suppose, for example, we have a single-threaded user program containing n procedure calls, each procedure consisting of exactly K single-threaded components. Let also all procedure calls from within any procedure occur within a single component and the number of calls from every component is equal to n. The resulting balanced n-ary tree is shown in Fig.3.

It can be easily seen that under the conditions above any n neighbour subtrees in our tree always work sequentially. This means none of the components belonging to the one subtree can go in parallel with a component from the neighbour subtree. In other words, those components need not be placed on different transputers. Assume further that all nodes in our tree represent the same procedure, their components thus being connected in the same way. In this very special case the total number Q of transputers required can be obtained easily to be equal to

$$Q = K*\log_n(N+1),$$

where N is the total number of procedures called (including user program).

If, on the other hand, we assume that all procedure calls from every procedure in the tree above are executed in parallel occuring in different threads or different components, then Q can increase up to the total number of components. For example, if n=2, K=2, $Q = K*N+1$. This is not, however, always true, because of the restrictions imposed by the number of links available on the transputer. For example, the situation when more than four procedures should be called from a procedure in parallel, is clearly unimplementable.

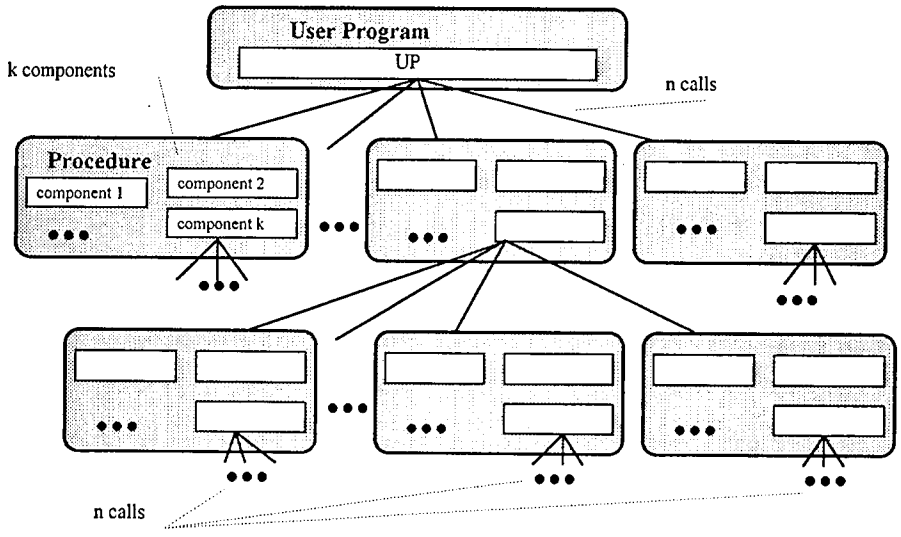


Fig.3 A caller-callee tree example

3. The Main QUICKPLAY's Components

3.1 The Preprocessor.

The aim of the Preprocessor is to identify parallel procedure calls within the texts of the user program and the procedures called and to build up the caller-callee tree, which serves as the input to QUICKPLAY's further phases. Let's have a closer look at how preprocessor works.

Assume, for example, that the user program contains a call of the parallel procedure P. This call must have the form

$<*label*>$P$<*v-string*>$(<*parameter-list*>) (*)

where P is the name of the procedure, and $<*v-string*>$ is the string to be replaced later on with the actual version number N, producing the name of the form PvN, which is the QUICKPLAY's default for the name of the N-th version of the parallel procedure P. The <*v-string*> is called the **version specifier**.

The <*label*> above is an integer which is used by the Layout Generator (see below) to distinguish between the calls from within the different threads of the same component.

Besides, some parameters in the <*parameter-list*> may have the form $<*p-string*>$ (called **parameter specifier**). This means that the values of these parameters must be provided by the user at the preprocessing stage. These parameters might be useful to introduce e.g. the information concerning data distribution among processors, which is often critical from the viewpoint of the performance of the application. As the structure of the caller-callee tree is unknown when creating the procedures, the only way to define the data distribution strategy for the particular procedure is to do it when building the tree. The preprocessor modifies the texts substituting the values into the calls.

<*v-string*> and <*p-string*> are both ordinary strings used to verbalise the request from the preprocessor for the user-defined values. Let's consider a simple example.

Let <*v-string*> in (*) be "P Version Number", and there exists single <*p-string*> in its parameter list, this <*p-string*> being "P's first parameter value". Assume also that P calls a procedure Q, the <*v-string*> and a single <*p-string*> of this call being "Version of Q called from P" and "Q's second parameter value" respectively.

The preprocessor first looks through the text of the user program and finds out the call (*). Then it outputs:

User Program, P call: Enter P Version Number
>3
User Program, P call: Enter P's first parameter value
>12345

Here '>' sign denotes user's responces. Then the preprocessor recursively analyses the text of the procedure P in the library requesting for the <*v-string*>s and <*p-string*>s of the calls found.

Procedure P, Q call: Enter Version of Q called from P
>1
Procedure P, Q call: Enter Q's second parameter value
>2

The whole process repeats recursively until there are no more calls unprocessed. This is user's responsibilty to make the specifiers clear and self-explanatory. Those for the library procedures are the part of library documentation.

We omit here the technical details of the library organisation, such as the structure of files and indices in it. The resulting caller-callee tree once constructed cannot be modified. The only possibility for the time being is to delete it and re-build by running the preprocessor once more.

3.2 The Layout Generator.

To produce the layout, the Layout Generator accepts the caller-callee tree built with the Preprocessor, extracts the information about the components of the procedures involved from the library and attempts to assign the processors the components. The Layout Generator also has to configure the network making use of spare links (i.e. enable the configuration information for the Text Generator). As the problem of finding the layout is very time consuming, the Layout Generator uses a fast heuristic algorithm to solve it.

First, the Layout Generator analyses the tree to identify pairs of components working in parallel and generates the so-called matrix of constraints specifying which components may be placed on the same processor. The (i,j)th element of this symmetrical matrix is equal to 0 if ith and jth components always work sequentially, and 1 otherwise. Besides, by default, of the K components of some procedure, the component that is responsible for receiving parameters and sending the results back (the so-called **leader**) is always placed on the same processor as the caller. This is done to avoid parameter passing down the links.

Then the Layout Generator fetches the structures of all procedures called from the library and generates an initial layout. If no transputers require more than four links to communicate with the others, it reports success, draws all connections required and terminates. Otherwise, it splits the loadings of those transputers, where more than four links are required, occupying more transputers, if any. If no more spare transputers are available, the Layout Generator fails.

It is important to note that the Generator does not attempt to minimise the amount of the transputers used. If the Generator fails for whatever reason (the solution either really does not exist or cannot be found with the algorithm used), it generates information for the user to help him/her to analyse the situation. An example of how Layout Generator works is shown in Fig 4.

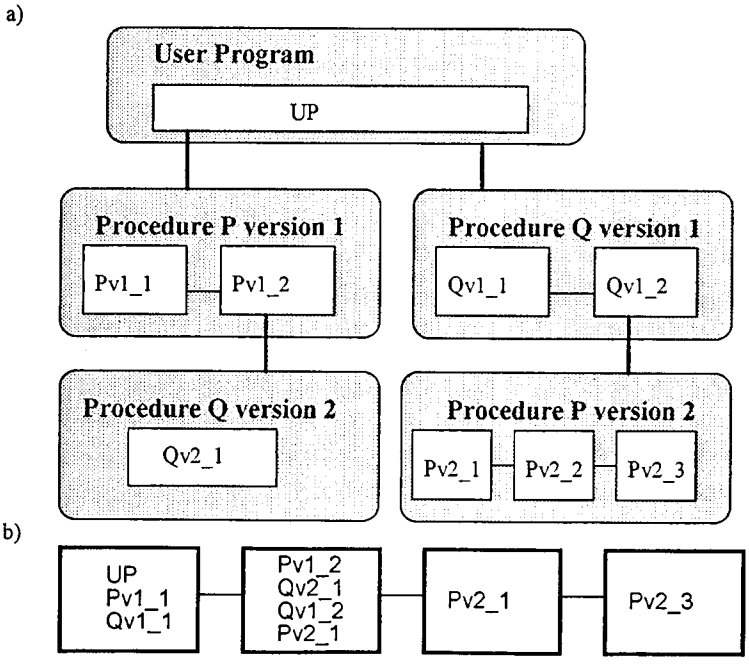


Fig.4 a) An example caller-callee tree, b) final layout

The user program calls two procedures, P and Q, both versions 1, which in turn call the procedures Q and P, versions 2 respectively. We assume that both calls in the user program are executed from the different threads thus making the procedures P, version 1, and Q, version 1, working in parallel (this situation can be recognised using *<labels>* defined above). Boxes representing components are named *<Procedure Name>*v*<Version Number>*_*<Component Number>*, so, e.g. component 1 of a procedure Q, version 2 is denoted as Qv2_1. Bold lines denote calls, plain lines represent the connections between the components. The final layout shown in b) includes both the layout of the components itself and the resulting network architecture which is in our case the simple pipeline.

3.3 The Text Generator

The task of the Text Generator is to build the texts of the output **modules**, given the parallel library, the caller-callee tree obtained from the Preprocessor and the layout generated by the Layout Generator. Besides, the Text Generator has to produce the configuration file (in 3L C terms) and the file configuring the transputer network, if necessary.

Each output module is a C program to be later loaded onto a single transputer. It includes all components, each component being a collection of threads, as was mentioned above, and special code initialising the threads and global data. As the module has to be designed so as to enable correct interface between its components and the outer world, each data packet is to be correctly delivered to the corresponding destination component. The module containing this component has therefore to distinguish between the packets addressed to the different components within it. This causes another piece of code to be generated and included into each module. Later on we call it **multiplexer**. Actually, the multiplexers can be considered as the constituents of a very simple router, enabling correct data delivering down the statically predefined paths without any buffering.

To make it possible for the multiplexers to distinguish between messages, the latters are provided with the **envelopes**, identifying the destination and the sender. The length of each message therefore increases of a predefined constant. The overhead imposed by this solution has been experimentally studied (see Sect.IV). Special communication procedures provided by QUICKPLAY automatically extend each data packet in the process of communication with the correct envelope.

4. The Experiment

To validate the approach we implemented with QUICKPLAY the back-propagation method for three-layered perceptron. We chose the straightforward implementation when each transputer in the network held the whole neural network, and this was the set of patterns that was distributed among transputers (this type of parallelism was called the *training related parallelism* in [4]). The transputers were connected so as to form the tree structure. Each transputer, except for the root, processed its own set of patterns and then sent the weights properly modified to its ancestor. .The root processor collected all modifications in the resulting matrix. For the simplicity the number of steps of the back-propagation algorithm was chosen to be constant.

The speed-up of the parallel algorithm appeared to be strongly problem dependent varying significantly as the structure of the network, number of patterns and distribution strategy changed.

The single-threaded user program, besides of the prologue and epilogue, contained the single call of the procedure LEARN. There were three versions of the LEARN in the library, tailored for one, two and three transputers respectively, as shown in Fig.5.

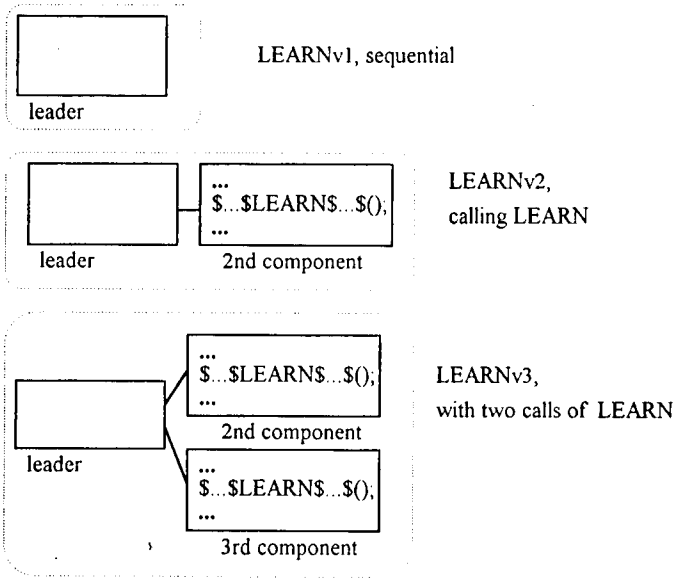


Fig.5 Different versions of LEARN

Actually, using these three versions of LEARN one can arrange the transputers in a tree with the subsets of patterns transferred down the edges. An example tree including four transputers is shown in Fig.6.

We first found that the less the neural network and the size of the set of patterns are, the less is the total speed-up.We then studied a lot or trees to find out the best distribution of the set of patterns among transputers using the *<p-string>* mechanism described above. We divided the set of the patterns passed to the procedure into subsets

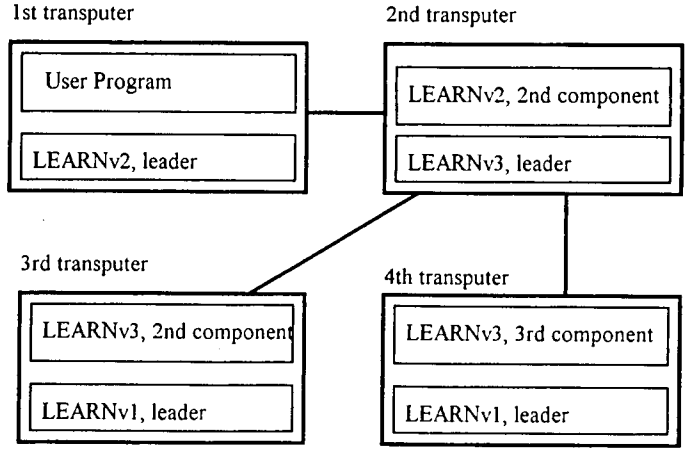


Fig.6 A transputer tree

Tab.1. Experimental results

Neural networks struktures	Sequential program	Parallel program implemented manually		Parallel program implemented with QP		QP versus manual
		time	speed-up	time	speed-up	
4-4-4	5587	2721	2.05	3566	1.57	1.31
4-16-2	14125	5206	2.71	6393	2.21	1.23
8-8-8	15568	5176	3.01	6267	2.48	1.21
4-16-16	35548	10516	3.48	11391	3.21	1.08
16-16-16	50104	15070	3.25	15458	3.24	1.03
16-32-16	93620	28943	3.23	29272	3.20	1.01
32-32-32	177474	56859	3.12	57345	3.09	1.01

depending on the user-defined parameter of the call, and passed the subsets to the components. The uniform distribution appeared to be the best one if only the size of the set was sufficiently large and well balanced tree was used.

Finally, we compared the implementations of the back-propagation for different neural networks and different trees. For each combination the pure sequential implementation was first written. Then the algorithm was manually implemented and finally automatically generated with QUICKPLAY.

The results obtained for the simple transputer network of three transputers and the set of 48 patterns are shown in Tab.1. The structure of the application was as follows. The user program called LEARN, version 3, while the latter called two LEARNs, both version 1.

In the first column of the Tab.1 the structure of the neural network is shown. Three numbers denote the numbers of neurons at the input, hidden and output layers respectively. The second column shows performance of the sequential version of the back-propagation (all times are given in processor ticks). The third and fourth columns give the performances of the parallel implementations developed manually and generated with QUICKPLAY (denoted as QP) along with their speed-ups with respect to the sequential implementation. The last column displays the overhead that is due to QUICKPLAY itself, i.e. the ratio of the times, presented in columns three and four respectively.

The superlinear speed-ups in the Tab.1 are due to the implicit cacheing, inherent to the transputer. It follows from the Tab.1 that as the size of the neural network increases, the overhead caused by QUICKPLAY is going down, which was the case in all our experiments. Actually, the relation between the NN structure and the performance of the parallel back-propagation appeared to be not straightforward, being dependent not only on the total size of the NN, but rather on the numbers of neurons at different layers.

5. Summary

A system aimed at the facilitating transputer programming has been described. The class of applications has been defined that can be parallelised without any universal routing mechanism. The parallelisation technique consists in replacing the sequential procedures called with their parallel implementations extracted from the parallel library. Using different versions of the parallel procedures one can generate applications dedicated to the different transputer networks. This is how the scalability of the application can be achieved.

The approach has been validated for a simple example of a neural network modelling algorithm. Experiments shown that even for the modest transputer networks the development of the efficient parallel implementation of the learning algorithm was a serious problem. The solution to the problem normally depended on lots of parameters and required much efforts to manually implement. Using QUICKPLAY allowed to facilitate the task, providing the tools which performed the job almost automatically.

The system has been tested for the small transputer network of 7 transputers and demonstrated low overhead caused by the approach itself. In the nearest future we are going to develop the version of QUICKPLAY for 64-node Parsytec GC machine. A library of common numerical procedures is also to have been implemented.

Another interesting point with QUICKPLAY is that it can be viewed as the software tool providing the user with the Remote Procedure Call (RPC) mechanism. If extended in this direction, QUICKPLAY would allow each parallel procedure in the library to use a router enabling correct interface between the routers. A larger application-oriented router could be thus implemented as the collection of smaller routers. For example, one could imagine a "complex" processor farm, where some workers serve as masters in the lower-level farms etc. Real experience with larger applications will much help in studying the problem in more details.

References

[1] C. Bonello, F. Desprez, B. Tourancheau, Basic Routines on a Reconfigurable Network, To appear in Parallel Computing.

[2] F. Desprez, B. Tourancheau, LOCCS: Low Overhead Communication and Computation Subroutines, Tech.Rep. 92-44, LIP-IMAG, Dec.1993.

[3] Rajan Kumar Gianey, An implementation of parallel matrix inverse algorithm on a transputer array, Transputer Initiative Mailshot, Sept.1991.

[4] Magali E.Azema-Barac, A Generic Strategy for Mapping Neural Network Models on Transputer-Based Machines, in Transputer Applications,1991, Glasgow, UK

A Simulation Environment for Finite-State Machine Models on Parallel Transputer-based Architectures: a Plant Automation Case Study

Valeria Filippi
ENEL Società per Azioni
Automatica Research Center
Via Volta 1, 20093 Cologno Monzese (MI)
Italy
e-mail : filippi@cra.enel.it

Gian Luca Redaelli
ENEL Società per Azioni
Automatica Research Center
Via Volta 1, 20093 Cologno Monzese (MI)
Italy
e-mail : redaelli@cra.enel.it

Summary

This paper deals with the feasibility study about distributing and executing finite-state machine models, represented by means of automata networks, on a hypercube Transputer-based architecture. In this architecture the basic processing element (named *complex-node*) is made of two Transputers coupled by means of a dual-port shared memory, accessed by the standard memory bus. Such complex-node offers 8-links towards other nodes and therefore can be used to build *n-cube* topologies with up to 128 nodes.
The study has been developed considering a sample applicative case, regarding the simulation of High Voltage (HV) substations, whose behavior can be described using the automata network methodology.

Keywords : Transputer, Express, Parallel Programming, Simulation, Finite-State Machine Models.

1. Introduction

In the plant automation field, both simulators and control systems generally suffer from the requirements of strictly *real-time* performance. Moreover, control systems require *fault-tolerant* characteristics that only proper hardware architectures may guarantee in embedded systems.
Several applications have been delivered which have to reach a compromise between real-time performance and functionality offered to the user. Thus, many systems have been built with a reduced functionality set to cope with faster response time.
Moreover, simulators that have been built in order to *verify* control systems in virtual environments (running in simulation time) may hardly be used to *validate* the real-time characteristics of control systems, when control systems are transferred to target machines. For this reason, in many cases, the real-time performance of embedded systems may only be proved when they operate on the actual plant, possibly causing problems for the human safety and the plant integrity.
A possible solution to increase the system performance, that may allow to have faster and/or more complex systems, could be the distribution of the system functions on a closely-coupled multi-processor architecture. Furthermore, parallel architectures may allow the necessary hardware redundancy to be obtained in order to reach the required degree of fault tolerance [1].
In our laboratories, Transputer-based parallel architectures are going to be analysed from several viewpoints in order to meet mainly the real-time performance requirements of simulator systems, possibly using custom architectures. In the future, we will deal with the required fault tolerant characteristics of control systems.

In synthesis, the viewpoints are:

- hardware: both conventional and innovative Transputer-based architectures are being studied. An off-the-shelf Quintek Transputer board with nine T800 Transputers has been compared with a *complex-node* based board [2], in which each node is obtained by coupling a pair of T800 Transputers by means of a shared memory. The complex-node is a custom processing element, that was realised in a joint-project between Enel and the Genova Engineering University. Complex-nodes may be used as basic components, each of them having 8 links, in order to build highly connected networks.
- application: a feasibility study has been performed for distributing the HV substation simulator on a parallel hardware. A particular emphasis has been paid to the communication/synchronisation features and load-balancing policies of the distributed functions. This study has been generalised to address an entire class of simulators whose functionalities may be modelled by finite-state machines
- performance: stochastic models [3] will be developed in order to model both the applicative software and the hardware architecture. By means of them forecast on execution time, speed-up, communication overhead, etc. versus the software distribution will be available since the earliest design phases.

This paper is mainly focused on the application and the generalised simulator that has been identified, thus a detailed hardware description of the complex-node and performance evaluations using stochastic models will be presented in future papers.
As already pointed out, within this framework we only consider the subclass of those automation systems that may be modeled by finite-state machine. These systems perform logic-sequential activities [4] and can be described using *automata networks* [5] [6].
In section 2, using the sample case study of the HV substation simulator, we explain the automata network methodology for describing system functions and their intrinsic parallelism. In section 3 we describe the generalised simulation environment that has been developed in order to execute any automata network. Section 4 reports the most significative results about the execution time versus automata distribution over a conventional Transputer network; the complex-node inner architecture is also shown.

2. Describing finite-state machine models using automata networks

A finite-state machine model can be decomposed into finite-state component machines, and so forth till the most elementary components are identified. During this decomposition process, the interactions among components have to be as well identified.
The automata network methodology [7]supports the system decomposition capability: at each elementary component an automaton may be associated, and interactions are modeled by the network connections.
In the following this methodology is made clearer by means of the sample application of an HV substation.
The HV substation is a node of the electric High-Voltage network, providing voltage transformation and connection among electric lines. The substation's components are basically breakers (B), insulators (I), transformers and busbars. While breakers can be opened or closed when current flows in them, insulator can not. A proper collection of these basic components, co-ordinated to do a particular task, is called a *functional-unit* (e.g. Dd, Ae, Fa).

The simplified HV substation considered in this paper is composed of a *busbar* (two bars, A and B), split in two sections, two *bar section isolators* (Ae1, Ae2), two *line connectors* (Fa1, Fa2), one *bar coupler* (Dd1), as shown in fig. 1

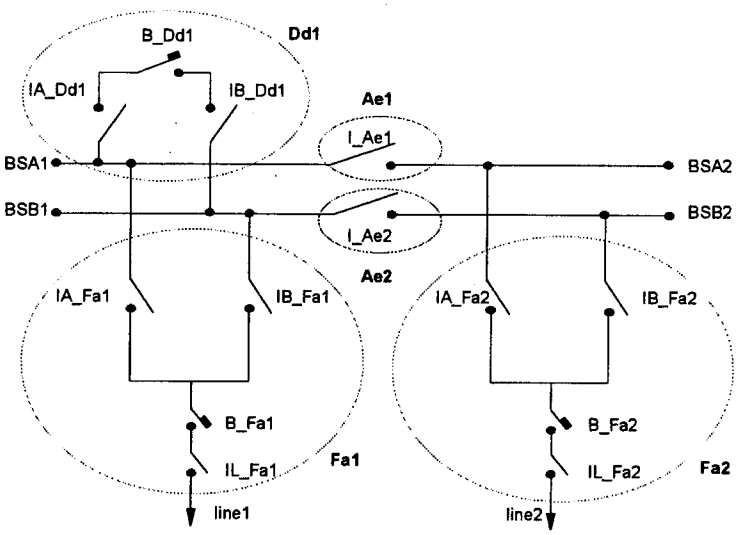


Figure 1 - The HV substation scheme

The overall system functionality may be described using a hierarchical network of finite-state automata, where lower levels generally represent elementary components functionality, intermediate levels represent co-ordination activities, and higher levels represent strategies (see fig. 2 where each box corresponds to an automaton). The hierarchical model subsumes the components relationships: automata belonging to the same level are independent, on the contrary, automata belonging to different levels may interact each other. Interactions among automata belonging to different levels are expressed by means of communications.

Each automaton describes a component behavior and is placed on a certain level of the hierarchy; it may interact with lower or higher level automata, but must not interact with automata at the same level. For instance, `Coord_Fa1` represents the coordination automaton for the functional unit `Fa1`, it gets a concise order from the `Strategic` automaton and gives orders to each of the elementary components it coordinates. Also it recognises the states of its elementary components in order to give a concise information to the `Strategic` automaton (i.e. if `IA_Fa1`, `B_Fa1` and `IL_Fa1` are closed and `IB_Fa1` is open then `Coord_Fa1` is closed on bar A; if each of them is open then `Coord_Fa1` is open, and so forth).

Composing several automata in a network makes them behave *synchronously*, i.e. the state transition of each automaton happens at the same time. These state transitions could depend on the state of other automata in the network; when two or more transitions must happen at the same time, this is equivalent to a bound transition in a Petri Net.

Referring to the previous example, if `Coord_Fa1` is closed on bar A and `B_Fa1` gets open, then `Coord_Fa1` gets open at the same time.

As an implementation mechanism to bound transitions of different automata, we adopted explicit communications using *channels*. Thus, when `B_Fa1` has the capability to get open (i.e. the transition towards "open" is enabled), on a dedicated channel it communicates this condition to

`Coord_Fa1`, and `Coord_Fa1`, that is closed on bar A, verifies the presence on that channel of such condition that enables it to move towards the "open" state. As already mentioned, after the communication has been performed, the two automata may change their state (synchronously).
To execute automata networks, a scheduler has been implemented which schedules *sequentially*

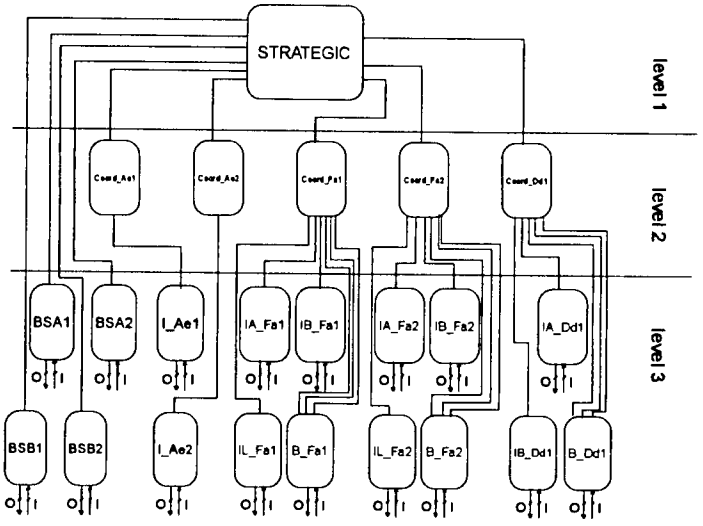


Figure 2 - Automata hierarchical network

each level, and *concurrently* the automata belonging to the same level. The schedulation activity is cyclic and each cycle consists of two phases: *upward,* all automata belonging to the lowest level are scheduled concurrently; then, when they have completed, the upper level is considered, and so on, till the top level (one automaton) is reached; *downward,* the top level automaton is scheduled first, and then the lower levels are considered till the lowest level is reached.
During each automaton activity any channel access may be performed, therefore, enabling automata to communicate over the network across different levels.
Before the upward phase, all the inputs are collected and put on special channels connected to the input/output interface and after the downward phase the outputs are written on the corresponding channels of the same interface.
Each automaton, when is executed upward, reads its inputs from both communicating channels coming from the lower levels and input channels if any, evaluates the transition condition set - thus identifying the enabled subset - and then writes its outputs on the channels linked to the upper levels. When it is executed downward, it reads the orders coming from the upper levels, according to them it selects one of the conditions in the subset - thus determining the future state - and then writes its outputs on the channels connected to the lower levels or on the output channels if any.
Of course, the top level automaton is an exception because during the upward phase it doesn't write any output and during the downward phase it doesn't read any input.
Eventually, all the automata change their state synchronously according to the future state previously determined.

As an example let us consider a short circuit acting on a bar section (see fig.3): such event involves all the functional units "closed" either on that bar section or on an electrical connected bar section. In this case the information of the short circuit coming from the busbar section automaton, e.g. `BSA2`, placed at the lowest level, arrives to the `Strategic`, which states which functional units are involved (in this case `Fa1`, `Dd1`, and `Fa2` because `Ae1` is closed), considering the whole substation state. Then, each functional unit selected by the `Strategic` whose state is "closed on bar A" (`Fa1` and `Fa2` in this example), orders its breaker to open (notice that `Ae1` has no breaker).

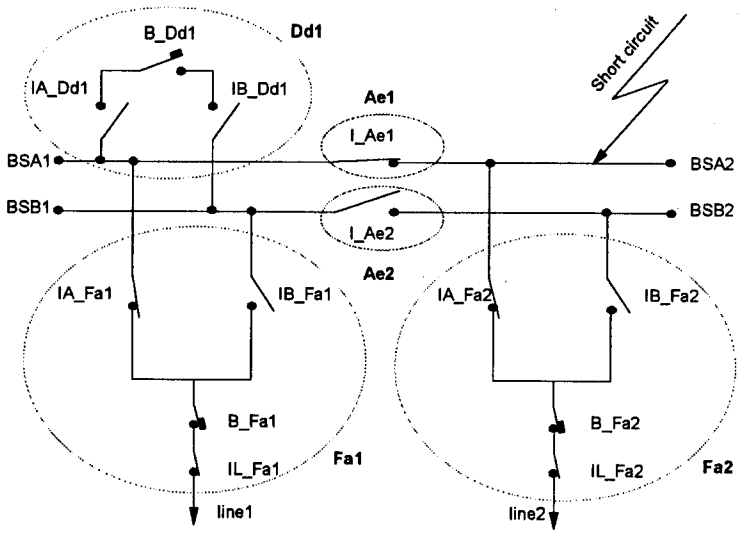


Figure 3 - Fa1 and Fa2 closed on bar A

3. The simulator software architecture

The simulator has been developed using the Parasoft Express software environment [8] along with 3L parallel C [9] on a Quintek Fast9 Transputer board. We are currently facing the problem of porting this application on a complex-node based architecture, for which we have already developed a proper routing subsystem.
The simulator is suitable for simulating any automata network, described according to the adopted model. It consists of:

- *file-templates generator* - this module asks the user about some automata characteristics such as the automaton name, state codes, automaton I/O, position in the hierarchical network, etc. in order to generate the file templates. The user shall fill in the generated templates with the state equations and transition conditions, compile and link them (*editor, compiler, linker* in fig.4);
- *configuration module* - this program is used to generate automatically three configuration files (TOPOLOGY.DAT, CONFIG.DAT, CHAN_TAB.DAT), which respectively contain the hierarchical network description, the match between each Transputer node and the corresponding program name, and the channels allocation over the nodes;

- *cubix* - it's the Express program used to download and run the user's programs over the Transputer network, as stated in the CONFIG.DAT file;
- *scheduler* - we developed an *application-independent* scheduler, whose goal is to execute the automata functions according to the hierarchical order as written into the TOPOLOGY.DAT file.

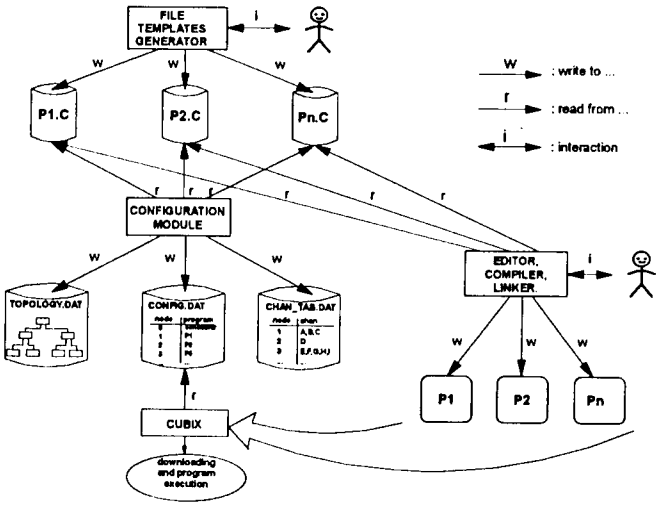


Figure 4 - The simulator framework

In figure 4 the modules involved in the application distribution phase and their interactions are shown.
Cubix downloads the scheduler to the master processor and the executable programs P1,...Pn to the other nodes as defined in the CONFIG.DAT file (see fig. 5).
Before executing the actual application, an automatic start-up phase is performed, during which each main node process allocates the proper channels as stated in the CHAN_TAB.DAT file.
During the simulation the scheduler sends the proper interrupts to the nodes in order to activate concurrently all the automata required (either on different nodes or on the same node in multitasking). Each automaton may access any channel, either on the same node or on a remote node, thus activating a remote task (sending a proper interrupt).

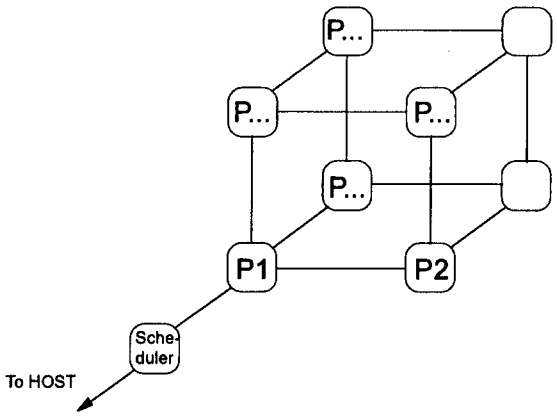


Figure 5 - Program mapping on an hypercube topology

4. Distributing automata over the Transputer network

In our first implementation, we defined an application design in which each program (P1, P2, Pn) has the following tasks: one for each automaton that has to be executed on that node, one for writing on the remote channels and one for reading from the remote channels (local channels are directly accessed).
Since Express allows to run at most ten tasks concurrently on each node and two of them are dedicated to handle the channels, eight tasks are available to run eight automata.
With this constraint we have distributed our application code in two different ways: firstly using three slave nodes (notice that the twenty-three automata composing the application require at least three slave nodes), and secondly using eight slave nodes that is the maximum number available on the Quintek board; in both cases the master node is reserved for the scheduler.
Naming **Taa** the average execution time (for both the upward and downward phases) for an automaton in case it doesn't access any remote channel, the theoretical minimum time spent for the execution of the whole automata network may be obtained from the following formula:

$$T_{\min} = \sum_{l=1}^{L} \max_{i=1}^{N} \left\{ T_{aa} \cdot n_{li} \right\}$$

where

n_{li} is the number of automaton at the level **l** that are allocated on processor **i**
L is the number of levels and
N is the number of processors.

In other words, basing on the fact that the scheduler executes sequentially each level and concurrently all the automata belonging to the same level, we have to add up the elapsed time for each level, and calculate those contributions referring to the time spent on the slowest node, i.e. the node that, for the considered level, has the most number of automata allocated on it.
With the former distribution we can estimate the minimum execution time as (6+1+1)*Taa=8Taa (i.e. we have allocated 6 automata of the lower level on node 1, 6 on node 2 and 5 on node 3, and the maximum among {6, 6, 5} is 6), with the latter we have (3+1+1)*Taa=5Taa (i.e. we have allocated at most 3 automata of the lower level on the same node), thus the theoretical maximum velocity improvement is 60%. Experiments on the complete case study (with also remote

communications) prove that the velocity improvement is about 32% because of the increased communication overhead.
We are currently analysing another programming model in which only one task, a *micro-scheduler*, is active on each node.
Since the automata execution time is very short in respect to remote task activation, in this case we expect to have a better overall system performance.
Early sperimentations regard the complex-node based hypercube architecture.
The complex-node is made of two Transputers coupled by means of a shared memory directly accessed by each Transputer using its data-bus (see fig. 6). Using the Transputer data bus, all the four links for each coupled Transputer may be connected to other nodes. Moreover communications through the shared memory are almost six times faster than through the serial links.
For this architecture a routing subsystem that allows to send/receive messages through the shared memory as well as through the links has been realised.

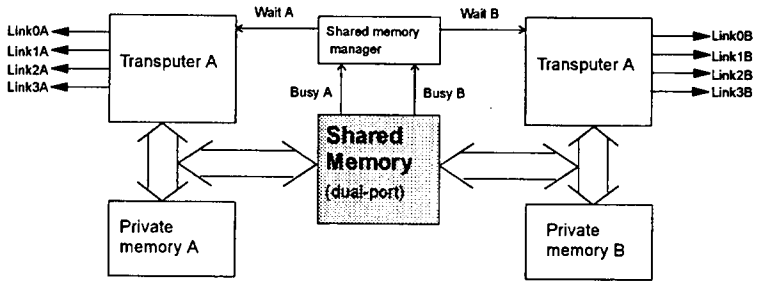


Figure 6 - The complex-node architecture

5. Conclusion

In this paper we have shown the major results of our early sperimentations about distributing plant automation system functionalities over a parallel architecture. In particular, we focused on the implementation of a generalised simulator able to execute finite-state machine models, expressed by means of automata networks.
We are now analysing other programming models, within the same framework, to verify alternative solutions in order to get better performance. Furthermore, we are porting the applicative case study on a complex-node based hypercube, using a suitable routing system.
Besides we are analysing the feasibility of implementing an efficient *micro-kernel* that will make possible to consider each complex-node as a single processing unit with 8 links; in this way the fact that each processing unit is made of two coupled Transputers will be transparent to the user.
Furthermore, another project about developing stochastic models of both the applicative software and the hardware architecture has just started. Such models may be used to get a forecast of execution time, speed-up, communication overhead, etc., since the earliest design phases.

References

[1] L.Brown, W Jie, "Optimal Triple Modular Redundancy Embeddings in the Hypercube", *Proc. Euromicro MPCS'94* , Ischia, Italy, May 2-6, 1994.

[2] AAVV, "Architetture a Transputer per elaborazione di segnali e immagini," *PIXEL* num.11, 1991.
[3] O. Botti, F. DeCindio, "Process and Resource Boxes: an Integrated PN Performance Model for Applications and Architectures," *Proc. IEEE-SMC'93*, LeTouquet, France, October 17-20, 1993.
[4] R. Gargiuli, P. Mirandola, et al., "ENEL Approach to Computer Supervisory Remote Control of Electric Power Distribution Network," *Proc. CIRED*, Brighton, 1981.
[5] L. DiLorito, R. Meda, "Un esempio di metodologia e di ambiente di supporto allo sviluppo di funzioni di automazione," *Automazione e Strumentazione*, January 1990, pp. 127-134.
[6] V. Filippi, R. Meda, "A Support System Shell to develop and configure Automation Functions for Electric Plants," *Proc. IEEE-SMC'93*, LeTouquet, France, October 17-20, 1993.
[7] F. DeCindio et al., *Le reti di automi sovrapposti: una classe modulare di Reti di Petri*. Tech. Rep. ENEL-CRA, July 1987.
[8] *Express 3.1 Introductory Guide*. Parasoft Corporation, Pasadena, CA, 1988-1991.
[9] *Parallel C 2.2.2 - User Guide*. 3L Ltd., Edinburgh, UK, 1991.

PipeLib : A Parallel Library for Writing Pipeline Applications

Luis Moura Silva João Gabriel Silva

University of Coimbra
PORTUGAL
E-mail : lams@pandora.uc.pt

Abstract

This paper presents the design of a parallel library (PipeLib) that offers support for writing parallel applications which follow the pipelining paradigm. The library was implemented on top of the Helios Operating System and runs on transputer networks. The aim of the library is to offer a high-level support to the application programmer in order to increase the programming flexibility and to promote the reuse of code. Another important feature of the library is the support for fault tolerance. The library interface will be presented in the paper together with some examples.

Keywords : Parallel Programming; Parallel Libraries Design; Pipelining Applications; Programming Paradigms; Transputers; Helios OS.

1. Introduction

One of the most important features in a parallel programming environment is *programmability* [1]. The programming interface should provide enough functionality to cover the needs of the application programmer and at the same time be sufficiently compact and intuitive to be generally comprehensible. Most of the message-passing systems only provide a low-level communication service leaving the programmer with some of the hard work to parallelize an application. One way to facilitate the job of parallel programming is to offer a high-level support in the form of parallel libraries oriented to the parallel programming paradigms [2]. One of the best examples of this guideline is a collection of parallel libraries (PUL) (written at the Edinburgh Parallel Computing Center) and running on top of CHIMP [1] that offer high-level programming support oriented to the programming paradigms. With such support the application programmer is able to spend more time on the details of the application rather than in low-level parallel programming details, and at the same time it encourages the reuse of code. Another example is the one herein presented which involves the Helios Operating System [3]. On top of Helios there has been implemented some similar parallel libraries, namely: the *FarmLib* [4], a special library for writing farming applications, the *TupleSpaceLib* used for writing Linda-like applications [5], and a last one for grid applications, called the *GridLib* (that is still forthcoming). In this paper, we will present a fourth library - the *PipeLib* - that offers high-level programming support for pipelining applications. A sketch of the parallel programming libraries is presented in Figure 1. These libraries offer programming support for the most widely used programming models, removing the burden of writing such codes from the application programmer. Another interesting aspect that is common to all of them is the support for fault tolerance. In fact, each parallel library has been provided with a fault tolerant mechanism that allows the applications to tolerate partial and total failures of the system. The fault tolerant mechanisms of the remaining three libraries are presented in [6][7][8], while the checkpointing scheme that was incorporated in the *PipeLib* is described in [9]. In this paper, we will have the opportunity to describe just the fault-tolerant programming interface.

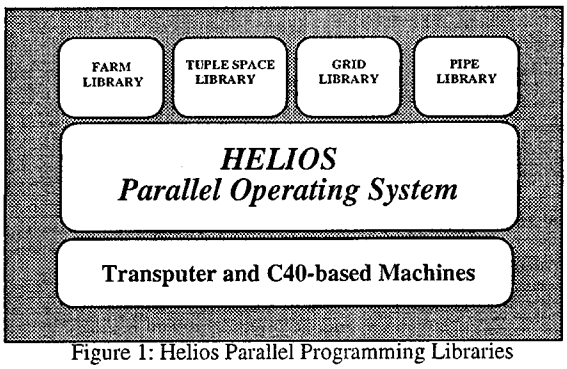


Figure 1: Helios Parallel Programming Libraries

The goal of the *PipeLib* is to hide from the programmer some of the tasks that can be done automatically by the library leaving the programmer free to concentrate on the application itself rather than of aspects of the parallelization. The parallel library should hide as much as possible the architectural details of the underlying machine. In fact, if the parallel applications are written in a higher level of abstraction they can be more easily ported to different parallel machines, providing the library is implemented on those machines with the same interface. Constructing portable programming environments is a very important goal and it is a paramount topic of research [10]. However, providing portability sometimes means sacrificing efficiency but one benefit of high-levels models of parallel programming is that they can be implemented on a variety of different hardware [2]. Another important aspect of the library is to facilitate the re-use of code among the applications that follow the pipelining programming paradigm.

2. Parallel Programming Paradigms

According to the Southampton classification [11] three classes of parallel programming paradigms can be defined for distributed memory MIMD machines :

- Farming Parallelism (replication of independent jobs)
- Geometric Parallelism (utilization of data structures)
- Algorithmic Parallelism (utilization of data flow)

The Farming and Geometric models are supported by the *FarmLib* and the *GridLib* respectively. The *TupleSpaceLib* offers a Linda-like programming environment that provides the abstraction of a virtual shared-memory and it departs from all the other libraries since it can support virtual-shared-memory programs as well as a generative message passing while the remaining libraries just offer a message-passing interface.

In this paper we are only interested in the later paradigm : Algorithmic Parallelism, here called by Pipelining model. In this paradigm some features of the algorithm that can be executed in parallel are identified and each processor executes a small part of the total algorithm. The algorithm is broken up in suitable size pieces and a typical feature of this approach is the organization of the processes in a pipeline structure. In such decomposition structure, the data flows between the processing elements and this is why sometimes this approach is designated by Data Flow parallelism. In this particular paradigm, load balancing issues and communication bottlenecks must be studied with detail in order to assure efficient algorithmic implementations [12], otherwise communication bandwidth problems and load imbalance can become dominant and severely degrade the performance. More than the functional decomposition of the algorithm pipelining exploits some sort of temporal parallelism. Stages in the pipeline behave like "filters" which take inputs from the previous stages and generate data streams to the following stages. The main important characteristic of the pipeline applications is in fact the concept of data flowing through

the several stages [13], but other important features can be identified [14], namely :
(i) It requires a regular and fixed communication structure;
(ii) It exploits the opportunities of overlapping;
(iii) It is asynchronous;
(iv) It reduces the amount of data stored locally on each node;
(v) It has regular operations.

It is clear that applications written with the Farming paradigm have the advantage of automatic load balancing while in the Pipelining paradigm the application programmer has to be concerned with the possible load imbalances between the processes. However, in some class of applications it becomes advantageous (and sometimes mandatory) to use the Pipelining paradigm [15][16][17] instead of the Farming of Geometric paradigm. It is naturally for such applications that the *PipeLib* is oriented for.

3. The Structure of Pipeline Applications

The processor node that is connected to the host is used as the root processor while the others processors are used to run the slave processes. When writing a pipeline application with the *PipeLib* the application programmer has to provide basically the code of the following four processes and the way they interact between them :
- **the Feeder**: runs on the root processor and is the one that creates jobs and sends them to the first slave of the pipeline.
- **the Collector**: also runs on the root processor and is responsible for collecting the results of the slaves.
- **the Provider**: it is used to provide some control information during the execution of the program.
- **the Slave**: is the one that does the real work and runs on every other processor of the network.

All the hard work such as the distribution and mapping of processes around the network, the virtual channels and buffer management, the routing of messages and the fault tolerance is done inside the library, in a transparent way to the programmer.
In terms of the communication structure the *PipeLib* allocates communication channels between neighbour slaves, and between each slave and the root process. Figure 2 presents an outline of the communication structure.

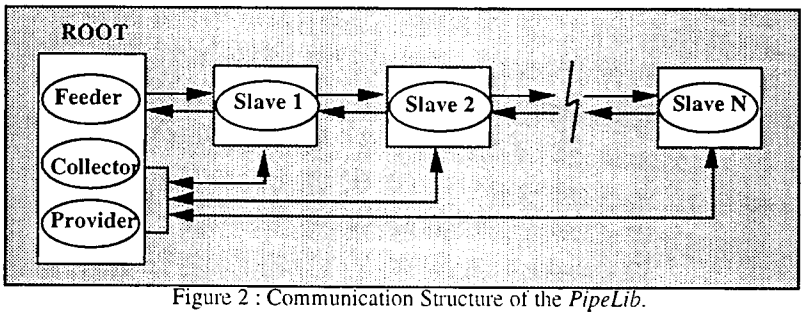


Figure 2 : Communication Structure of the *PipeLib*.

As can be seen by the Figure the library is not just restricted to pure (one-way) pipelining applications. It was our purpose to make the library as flexible as possible in order to be usable with a wide range of applications. In fact, messages can flow in both directions (backward or forward), and it also allocates communication channels between the first slave and the last one, a feature not represented in Figure 2 for the sake of clarity . This means that the *PipeLib* can also be used to run other kinds of applications that have a ring structure.
All the slaves have direct virtual connections to the *Collector* and the *Provider* but just the first

slave is able to receive data from the *Feeder*. The channels between the slaves and the *Provider* are mainly used to distribute initial data and to gather some application-specific information during the execution of the program. Those channels are also used to pass the information and data related to the fault-tolerant mechanism.

4. Library Interface

This section outlines the library calls and variables provided by version 1.0 of the *PipeLib*. Use of any of these routines requires the inclusion of the file `pipelib.h` in the source code.

4.1 Initialization

There are several library variables that can be manipulated by the programmer, allowing him to tune some options according to the needs of the application. The use of all these variables is not mandatory, and unless not changed by the programmer they will take the default values.

```
int   PpNumberSlaves;
bool  PpOverloadRoot;
bool  PpFastStack;
bool  PpFastCode;
int   PpDataMsgSize;
int   PpResultSize;
int   PpInfoSize;
int   PpFeederStack;
int   PpCollectorStack;
int   PpProviderStack;
int   PpSlaveStack;
bool  PpFaultHandling;
bool  PpVerbose;
int   PpInfoPeriod;
```

Figure 3: Initialization variables.

PpNumberSlaves is used to specify the number of slave processes in the pipeline. This number is limited to the maximum number of available processors and the initialization of this variable is mandatory. If the programmer wants to place a slave process on the root processor then she/he should set the *PpOverloadRoot* variable. The next two variables - *PpFastStack* and *PpFastCode* - would allow to run the slave processes with its stack and code on fast on-chip memory. This would allow in some cases a performance improvement of about 50%. By default those variables are disabled, meaning that the code and the stack are loaded into external memory. The following variables - *PpDataMsgSize, PpResultSize* and *PpInfoSize* - are used to specify the typical size of the messages that carry data, results and information. If those variables are initialized with some positive value then the library can perform some optimizations in its buffer allocation strategy, pre-allocating a suitable number of buffers during the initialization phase. Obviously, the application is allowed to use messages with different sizes, but their memory allocation will be done in run-time. The next three variables - *PpFeederStack*, *PpCollectorStack*, *PpProviderStack* and *PpSlaveStack*- are used to specify the stacksize of the Feeder, Collector, Provider and Slave processes. By default each process is given a stack of 5000 bytes, which should be enough for the majority of the applications. The *PpFaultHandling* variable is used to enable the fault tolerant mechanisms of the library. Since fault tolerance is an additional feature this variable is disabled by default. The *PpVerbose* is a flag that when enabled permits the visualization of library run-time warnings that are sent to the standard output. By default this flag is disabled, and is generally just used during the debugging phase of the application. The last variable - *PpInfoPeriod* - indicates the monitoring period in seconds. The library has an internal thread that periodically collects some information

about the load of the processors and the amount memory used by them. These information will be very useful to implement a load balancing scheme, as will be explained in section 4.6. By default, the *PpInfoPeriod* is set to -1, meaning that no information is collected during the run-time execution. The user should chose the most appropriate monitoring period according to the load balancing policy.
Next Figure presents the initialization routines. Those marked with - M - are of mandatory use, while the others are of optional use .

void	*(*PpFeeder) (void);*	*- M -*
void	*(*PpCollector) (void);*	*- M -*
void	*(*PpProvider (void);*	*- O -*
void	*(*PpSlave) (void);*	*- M -*
void	*(*PpRootInitialise) (void);*	*- O -*
void	*(*PpSlaveInitialise) (void);*	*- O -*
void	*PpInitialise (void);*	*- M -*

Figure 4: Initialization routines.

The first four routines indicate the code of the Feeder, Collector, Provider and Slave processes, and must be specified before calling the *PipeLib* initialization routine - *PpInitialise()*. In some rare applications there may be the need to specify different initialization routines, one for the root processor, and other for the slave processors. This can be achieved by specifying the *PpRootInitialise()* and *PpRootInitialise()* routines. Since the Feeder, Collector, and Provider processes run on the same processor care should be taken when accessing global resources, like the C library or some global data of the application. If some of these resources is shared by more than one process then its access should be protected in a mutual exclusion way. The normal interaction between these processes should also be done by using semaphores and global variables.

4.2 Termination

An application running the *PipeLib* will terminates normally when the Collector routine returns, or if one of the processes calls the `exit()` routine. If the Feeder or the Provider process terminates before the Collector there is no problem with that. It means that the Collector is still running because there are still some results to collect. Generally, the Slave routine executes a forever cycle. If a slave returns or exits for any reason while the Collector is running, that slave is treated as having failed in the same way as if the processor on which it was running had failed. If the *PpFaultHandling* option has been enabled the fault tolerant mechanism will perform the recovery of the application to the last checkpoint, otherwise the application will terminate.
Anyway, any of these cases gives a clean exit. It is possible for applications to specify additional tidy-up routines that should be invoked under these circumstances, for instance to delete a lock file. Two additional exit routines that are presented in Figure 5 can be specified during the initialization phase, and when the program terminates normally those routines will be called before leaving the `main()` program.

void	*(*PpRootExit) (void);*	*- O -*
void	*(*PpSlaveExit) (void);*	*- O -*

Figure 5: Termination Routines.

However, it should be worth noting that if the programmer terminates abruptly the application with a CTRL-C, for instance, then the clean-up may not happen, and is the responsibility of the programmer to take the necessary actions.

4.3 Communication Primitives

The communication routines can be divided into two separate groups: those that are used by the root processes (presented in Figure 7), and those used by the slaves (presented in Figure 8). They have just one routine in common, that is the one used to allocate a message buffer used by a sending routine. This routine which is presented in Figure 6, allows the library to optimize the buffer management, providing the programmer have used those initialization variables mentioned before with the typical sizes of the messages used in the application.

```
void    PpAllocBuffer(int size);
```

Figure 6: Buffer Allocation Routine.

An interesting aspect is that the inverse operation (releasing of the buffers) is done automatically by the library when they are no longer required. The programmer should never free them and in fact there is no library routine at all to do that.
The next routines are the used by the Feeder, Collector and Provider processes. All the communication is mailbox-based. This means that the sending operations asynchronous. In order to support an asynchronous communication service the root processors should maintain at least N*2 input mailboxes to keep the messages sent by the slaves and destined to the Collector and Provider. These messages are multiplexed through the same virtual channels, as well as the data used by the fault tolerant mechanism. All these communication routines provides a reliable service, i.e. the programmer can be sure that each message arrives at the destination despite some communication failures of the system. In fact all the message losses are automatically handled by the Helios OS [3] together with the provision of a dynamic routing facility.

```
bool    PpFeed(void *buf);
void    *PpCollect(int from);
bool    PpSend(int to, void *buf);
void    *PpRecv(int from);
```

Figure 7: Root Communication Routines.

The Feeder process sends data messages to the first slave of the pipeline through the *PpFeed()* routine. Obviously, there is no need to specify the destination process. The only thing that is needed is the pre-allocation of the message buffer through the *PpAllocBuffer()* routine. Care has been taken to avoid the overflow of the mailbox at the slave process, by implementing a control-flow scheme between each those two processes based on a sliding-window protocol. In this way, if the mailbox of slave 0 gets some maximum number of messages, then the Feeder blocks inside the *PpFeed()* routine until the slave consumes some messages of its input mailbox. In the current implementation the window size is just 2.
The *PpCollect()* is used by the Collector process. It returns a buffer with a message that has been allocated internally by the library and will be freed by the library when no longer needed. The "from" parameter can take two types of values: or it is a number between 0 and (PpNumberSlaves - 1), specifying the receiving of a message from a specific slave; or instead, the programmer can use the macro *ANY_SLAVE*, that chooses an available message from any slave.
The next two routines - *PpSend()* and *PpRecv()* - are used by the Provider. The same previous observation about the allocation of a message buffer is here applied to the *PpSend()* routine. The parameter "to" can also take two types of values: or it specifies a specific slave (between 0 and (PpNumberSlaves - 1)) to send the message; or the programmer can use the *BROADCAST* macro that sends the current message to all the slaves of the network. The *PpRecv()* routine is quite similar to *PpCollect()*, and all its features are also applied. An interesting aspect is the fact that the buffer returned by a *PpRecv()* can be used directly with a *PpSend()* routine, avoiding the allocation of another buffer through *PpAllocBuffer()*.

Inside the code of the slaves the allowed communication primitives are the ones presented in Figure 8. Each normal slave has 3 input mailboxes: one for keeping messages sent from the backward, other for messages arriving from the forward slave, and a last one for messages sent by the Provider. The exception is the first slave, that has a fourth mailbox to keep the messages sent by the Feeder.

```
bool   PpPut(int to, void *buf);
void   *PpGet(int from);
void   *PpGetAny(int from,int *whoisSender);
```

Figure 8: Slave Communication Routines.

The *PpPut()* routine allows the sending of a message to four different destinations: *BACK* (i.e. the previous slave), *FRONT* (i.e. the next slave in the pipe), *COLLECTOR* and to the *PROVIDER*. The *PpGet()* routine returns a message sent by a specific specified in the "from" parameter, that can take the following values: *BACK*, *FRONT*, *PROVIDER* and *FEEDER*. The message buffer returned by this routine can be used in the *PpPut()* library call, otherwise a buffer should be allocated before that sending routine. The limitation of the *PpGet()* routine is that it only allows the receiving of a message from one specific sender. In some cases, it would be very useful a mechanism to select the first message received from a set of process senders, and to achieve that the programmer should use the *PpGetAny()* routine. The first parameter of this routine indicates the possible sources of the message, like this: (*BACK* | *FRONT*), (*BACK* | *PROVIDER*) or (*PROVIDER* | *BACK* | *FRONT*). The other parameter (whoisSender) returns the identification of the sender process, that can be checked at the slave's code to take the appropriate action.

4.4 Query and Control

The library provides some variables and routines that can be used in run-time to obtain some information about the structure of the application and the physical network, which are presented in Figure 9.

```
int    PpCountProcessors()
int    PpNumberSlaves;
int    PpSlaveNumber;
bool   PpAmIFirst();
bool   PpAmILast();
```

Figure 9: Information variables and routines.

The first routine returns the number of available processors in the network, while the variable *PpNumberSlaves* gives the number of application slaves. The *PpSlaveNumber* indicates the number of the local slave, and is used generally to determine what is the exact code that should be performed by the slave, when we have an application with pure algorithm parallelism. The other two routines - *PpAmIFirst()* and *PpAmILast()* - are useful to make a distinction between the first or the last slave of the pipe and the remaining ones.

4.5 Load Balancing

A Pipeline application does not has the advantage of automatic load balancing as a Farming application. In fact, the programmer should care about the possible sources of load imbalance during the design phase of the algorithm. However, this may be not enough, since load imbalances and communication bottlenecks can occur during the execution of the program [12], and unless some load balancing strategy is performed, they can degrade severely the performance of the program. Providing a load-balancing mechanism is a very important feature, and the most efficient load-balancing strategies should be application-specific [18][19] based on some heuristic remapping algorithm. Some significant improvements can be achieved if the programmer provides

a load-balancing scheme into the application. However, to help the programmer in such task it would be nice if the library would provide some information about the status of the slave processors during run-time execution. The two following routines can be used to provide such information. The library spawns an internal thread that periodically collects some system information of all the processors used by the application. Among such system information there are the CPU load and the amount of memory used.

```
int    PpWhatIsLoad(int slave_id);
int    PpHowIsMem(int slave_id);
```

Figure 10: Routines used for Load Balancing.

The first routine - *PpWhatIsLoad()* - returns the CPU load of a specific slave in the range between 0 and 100, meaning that such value is given percentage. The *PpHowIsMem()* routine also returns a value between 0 and 100, representing the percentage of memory that is been used at the processor of the specified slave. With these two values it is possible to determine if the application has some point of load imbalance, and an in such case some application-specific reconfiguration action should be performed. This load-balancing task can be executed by the Provider process, for instance. However, there is one aspect that should be worth noting, that is: if the fault tolerant mechanism is enabled the library takes a periodic global checkpoint of the application, and this operation uses some memory buffer to keep the intermediate checkpoints until they are saved to disk. Care should be taken by the programmer to avoid the false-triggering of a reconfiguration procedure due to a temporary increase of the memory used by the processors of the network during a global checkpointing.

4.6 Fault Tolerance

Three primitives are available to the programmer to provide support for fault tolerance.

```
bool   PpStable(void * var, int size);
bool   PpRestart()
bool   PpCheckpoint()
```

Figure 11 : The Fault Tolerant Primitives.

The *PpStable()* system call specifies the stable variables that need to be saved in each checkpointing operation. It can be used by one of the three processes of the root (Feeder, Collector or Provider) and by any slave of the application. The programmer should know what is the data that really needs to be saved in case of failure, and such data should be pointed out by using this primitive. With this scheme we reduce drastically the amount of data that is saved in a checkpoint, thus reducing the memory overhead and the performance penalty. Another advantage of this semi-transparent scheme is that, since such data is specific to the application and can be easily converted to other formats, application-oriented checkpoints may be easily portable to heterogeneous systems, providing the recovery system uses the XDR protocol to translate the checkpoint data between different architectures.
The second library call - *PpRestart()* - should be called after specifying all the stable data through the previous routine, and can be viewed as an alternative to the bits of code that perform the initialization part of each application process.
In short, it will check if the application is restarting after a failure or if it is the normal beginning, by inspecting the stable storage in the search of a checkpoint file. If there is no checkpoint file the computation proceeds normally with the initialization; otherwise, the stable variables are loaded with the values they had on the last checkpoint, and the message queues will be filled up with the contents they had at the time of the last checkpoint.
At last, the *PpCheckpoint()* library call is the one that triggers a global checkpointing operation. It should be carefully placed by the programmer, and the placement policy is application specific. In some applications it would be adequate to take a global checkpoint from time to time, while in

others it would be desirable to have a checkpoint after receiving N results from the pipe. It really depends on the application and the freedom to chose the placement policy can be seen as an advantage, since in this way, the programmer can tune the fault tolerance according to his needs. However, since the root is the coordinator of the checkpointing protocol the *PpCheckpoint()* routine can just be used at one of the root processes (it is normally used by the Collector).

5. A Programming Example

For the sake of clarity, we present a sketch example of a pipeline application to demonstrate briefly their use. As can be seen the writing a parallel application can be a straightforward task by using the *PipeLib*.

```
#include <helios.h>
#include <pipelib.h>

Semaphore     Go;

static void feeder(void);
static void collector(void);
static void slave(void);

int main(void)
{
PpNumberSlaves = 8;
PpFeeder       = feeder;
PpCollector    = collector;
PpSlave        = slave;
PpOverloadRoot = PpTRUE;
PpDataMsgSize  = sizeof(data_buffer);
PpResultSize   = sizeof(res_buffer);
...
PpInitialise();
}



static void feeder(void)
{
data_buffer dt_buff;
int         i;

Wait(&Go);
for(i=0; i < NJOBS; i++){
 dt_buff=PpAllocBuffer(sizeof(data_buffer));
  ...  fill dt_buff
  PpFeed(dt_buff);
  }
}
```

```
static void collector(void)
{
buffer         res_buff;
int              i;

... init global_result data structure
Signal(&Go);

for(i=0; i < NJOBS; i++){
   res_buff = PpCollect(ANY_SLAVE);
   ... update results
   }
}

static void slave(void)
{
data_buffer  buf_1;
res_buffer   buf_2;

... init local_result data structure

forever{
   if(PpAmIFirst())
       buf_1 = PpGet(FEEDER);
   else
       buf_1 = PpGet(&A^K);
   ... Do Work with buf_1
   PpPut(FRONT,buf_1);
   if(PpAmILast()){
       buf_2= PpAllocBuffer(
                          sizeof(res_buffer));
       ... fill buf_2
       PpPut(COLLECTOR,buf_2);
       }
   }
}
```

Figure 12: An example of the use of the PipeLib functions.

6. Conclusions

A parallel programming tool can be distinguished by some important features, namely: Programmability, Portability, Reusability and Robustness. It is our purpose to achieve a collection of parallel libraries that offer a high-level support oriented to the programming paradigms and hide from the programmer the details of the underlying architecture and some of the hard tasks of parallelization. In this paper, we have presented the interface of the *PipeLib*, a library that offers programming support for applications that follow the pipelining paradigm. Most of the hard tasks are done internally by the library, leaving the programmer free to concentrate on the application

itself rather than of aspects of the parallelization. Additionally the library offers some support for load-balancing and provides fault-tolerance for long-running applications. We are currently gathering some performance results of the fault-tolerant mechanisms, and there are future plans for using the library. Such plans involve the parallelization of some image-processing algorithms from another research group. In fact, we have observed that most of the image-processing algorithms follow the pipelining model and they would be more easily programmed with the support of the *PipeLib*.

Acknowledgments

L.M.Silva is supported by JNICT under the "Programa Ciência" (BD-2083-92-IA). We would like to thank the technical support given by Bart Veer from Perihelion Sofware (UK) during the implementation of this parallel library.

References

[1] R.Bruce, S.R.Chapple, N.MacDonald, A.S.Trew. "CHIMP and PUL : Support for Portable Parallel Computing", Technical Report EPCC TR-93-07, 1993

[2] B.Veer. "The World of Parallel Programming", Proc. 2nd Workshop on Abstract Machine Models for Highly Parallel Computers, Leeds, April 93

[3] Perihelion Software. "The Helios Parallel Operating System", published by Prentice-Hall, ISBN 0-13-381237-5, 1991

[4] B.Veer. "The Helios Farm Library", published by Perihelion Distributed Software Ltd, part number H090030, 1992

[5] L.M.Silva, B.Veer, J.G.Silva. "The Helios Tuple Space Library" Proc. Euromicro Workshop on Parallel and Distributed Processing, Malaga Spain, pp. 325-331, January 1994

[6] L.M.Silva, B.Veer, J.G.Silva. "How to Get a Fault-Tolerant Farm" World Transputer Congress'93, pp. 923-938, Aachen Germany, 1993

[7] L.M.Silva, B.Veer, J.G.Silva. "A Fault-Tolerant Tuple Space Library" Technical Report DEE-UC-023-93, University of Coimbra

[8] L.M.Silva, B.Veer, J.G.Silva. "Checkpointing SPMD Applications on Transputer Networks", To appear in Scalable High Performance Computing Conf. SHPCC'94, Knoxville USA, May 1994

[9] L.M.Silva, J.G.Silva. "Checkpointing Pipeline Applications" Technical Report University of Coimbra (submitted for publication).

[10] J.E.Boillat, H.Burkhart, K.M.Decker, P.G.Kropt "Parallel Computing in the 1990's : Attacking the Software Problem" Technical Report IAM-91-011, University of Bern, Switzerland, 1991

[11] D.J. Pritchard. "Mathematical Models of Distributed Computation" Proc. of OUG-7, IOS Press, pp. 25-36, 1988

[12] A.J.G.Hey. "Supercomputing with Transputers - Past, Present and Future" Proc. Int. Conf. on Supercomputing, pp. 479-489, June 1990

[13] P.A. Nelson. "Parallel Programming Paradigms" PhD Thesis, University of Washington, 1987

[14] C.T. King, W. Chou, L.M. Ni. "Pipelined Data Parallel Algorithms - Concept and Modeling", Proc. Int. Conf. on Supercomputing, pp. 385-395, 1989

[15] R.Ralha. "Parallel Computation of Eigenvalues and Eigenvectors Using Occam and Transputers", PhD Thesis, University of Southampton, May 1990

[16] K.Siegl. "Groebner Bases Computation in Strand : A Case Study for Concurrent Symbolic Computation in Logic Programming Languages" Diplomarbeit, University of Linz, November 1990

[17] B.L.Menezes, I.L.Ricarte, R.Thurimella. "Analysis of Pipelined External Sorting on a Reconfigurable Message-Passing Multicomputer", Parallel Computing, Vol. 19, pp. 839-850, 1993

[18] A.N.Choudhary, B.Narahari, R.Krishnamurti. "An Efficient Heuristic Scheme for Dynamic Remapping of Parallel Computations", Parallel Computing, Vol. 19, pp. 621-632, 1993

[19] M.Smith, G.Wilson. "Dynamic Load-Balancing on a One-Dimensional Mesh" Technical Report EPCC TN-91-11, 1991

PPS-Construction: A Methodology for Computer Aided Construction of Parallel Program Systems

Vladimir D.Ilyin

Department of Methodology for Programming,
Institute for Informatics Problems of Russian Academy of Sciences,
30/6, Vavilova St., Moscow,
117900, RUSSIA
E-male:root@ipiand.ipian.msk.su.

Summary: This paper addresses the problem of creation of an effective methodology for computer aided engineering of parallel program systems (PPS). It describes the approach to computer aided engineering of PPS that is the attempt to solve this problem combining the generalization of our methodology for computer aided construction of non-parallel program systems (NPS) and our automatic parallelizing method designed for construction of PPS. The theoretical foundations of PPS-construction, task knowledge representation language and automatic parallelizing method are presented and described in short.

Keywords: Parallel Program Systems, Computer Aided Construction of Parallel Program Systems, Automatic Parallelizing Method

1. Introduction

The problem of program construction is bound up with the automatic parallelizing problem. To create an effective methodology for PPS-construction it is necessary to fix methodological differences between development of NPS and PPS. Besides it is important to ascertain ideas that are effective for development of NPS and PPS as well.

The known approaches to facilitation of safe and efficient programming of a parallel architectures involve the development of following features: languages (such as concurrent PASCAL, ADA, OCCAM, etc.) with their programming environments, parallelizing translators and runtime support environments. The data-flow model of computation is widely used methodological element of these approaches.

The functional model of computation is used in functional languages [1]. This model does not depend upon architectural notions. Functional principles of execution are used in data-flow systems [2]. Both the functional model of computation and functional principles of execution have influenced on creation of the representation of algorithm to be parallelized (Section 4.).

It is very important for development of safe PPS to create a methodology for computer aided construction of PPS. The purpose of this paper is to present and describe in short the theoretical foundations of methodology for computer aided construction of PPS, OBRAZ language for task knowledge representation and the automatic parallelizing method called MPM.

2. PPS-Construction as A Formal Process

PPS-construction is a formal process based upon both the set of task constructive objects and the family of construction rules. The formal representation of task constructive object is a special graph called task relation graph (G-graph). The family of construction rules is based on both the set of G-graph operations and the set of inference procedures which search subgraphs of G-graphs called solving structures.

PPS-construction is realized in spaces of task constructive objects and consists of such processes as object creation, concretization, specialization, generalization, notion shell replacement and new objects construction. From the formal view point PPS-construction may be considered as the process of inductive set construction.

The parallel program systems development in PPS-construction environment is considered as task constructions building. For this purpose the notion of task constructive object is introduced along with the number of other notions derived from it. They form the complex of notions used in modelling of the process of transition from the problem to be programmed to the program. This complex allows the user to formulate his ideas in terms of tasks at no diffidence and to advance step by step towards the producing of specified program system keeping in mind the meaning of every transition without difficulties.

The process of PPS-construction consists of other processes which are performed at different stages of working with the objects in the world of programable problems. This world is understood as the aggregate of task constructive object spaces.

The construction processes of NPS and PPS have common components. The first of them is the creation of basic objects (b-objects) and it is called p-creation. The specialization of basic objects and the derived ones (p-specialization) – is the process of setting of the correspondence between some object and non-empty set of derived objects. Derived object is designed for solving the problem which is a partial case of the problem represented by its base object. Object notion shell replacement (p-replacement) assigns certain problem domain names in accordance with a given lexicon. Making constructions is the process of linking task constructive objects by means of special communication functions. The process of object concretization called p-concretization defines the correspondence between the problem formulation and the methods of its solution, program templates and programs in accordance with implementation specification. The concretization is a "stage to stage" transition from a problem formulation (WHAT-representation of problem to be programmed) to a program (final HOW-representation). There are intermediate HOW-representations of problem on the way from formulation to program (methods and program templates) that are necessary for taking into account the problem solving environment specification. There are some additional components in the process of PPS-construction that will be mentioned later (Section 4.).

3. OBRAZ Language for Task Knowledge Representation

The conception and formalisms of task knowledge representation are developed for building of task knowledge base of the system of computer aided program construction [3]. They form the conceptual basis of OBRAZ language.

OBRAZ is designed to describe notion systems, which contain knowledge that are necessary for supporting the processes of task constructive objects definition, specialization, generalization, construction and concretization (Section 2). The language implements an idea to replace the development of task specifications by the development of the specification of application domain notions system. Interpreting of this notions system specification produces specifications of tasks that are necessary to solve fixed problem. This task specifications and specification of problem solving environment (computational resourses, data sizes, etc.) provide all information necessary to build program system.

The central concept of OBRAZ is a notion defined by its name and specification. Notion specification is a set of other subordinate notions (attributes) and relations between them. Language statement usually contains notion definition including such relations as membership and inheritance. Language statements can also describe constraints for notion value set and equivalence of notions as a special case of constraint.

The notion can have several definitions which express various points of view. Language interpreter considers every definition as a separate notion. User can establish his own view to the problem area selecting the only definition among multiple possible ones.

OBRAZ language is developed to be a tool that supports representation of knowledge that is necessary to build a target program system producing (computing) values of some notions by values of other notions. This goal have determined that notion model should be based on relations represented by programmable tasks and called task relations. Except task relations the notion specification can contain some other relations on notion value sets that are called constraints and are used to define suitable conditions when application of this specification is possible.

The task that formulates a relation on set of notions is also a notion with special meaning. Task specification contains two non- intersecting subsets of attributes called input and output, the aggregation of input and output is called task memory.

The program construction environment has no means for evaluation of task relations stored in its database. This determines the absence of variables in OBRAZ because there is no temporary data to store. The notion specification is close to concept of object class in object-oriented languages. To be briefly we'll sometimes skip words "specification" or "description", so one should consider terms "notion" and "task" as "notion specification" and "task specification". Text enclosed in /* and */ (multiple lines) is a comment. Text between // and end of line is also a comment.

3.1. Notion representation

The basic structural unit in OBRAZ language is a notion definition that is represented by its name and specification which in turn is a collection of definitions of other notions and relations between them. Specification can be formed from other specifications using inheritance.

Some notions in OBRAZ have built-in specifications. First of all, there are the most general notions such as "text", "number", "set" for which OBRAZ supports constants and special syntax extensions that increase readability and compactness of notation. There are also notions that are

used to build the task based models, such as "task relation", "task memory element", "input", "output", "task relation graph", "query". All this notions are called predefined.

Every notion in OBRAZ is defined inside the specification of some another notion. If notion A is defined inside notion B then we'll say that B is owner of A and A is attribute of B. This creates membership relations hierarchy that covers the whole notion system. Every notion that can be specified using OBRAZ is either the owner of something or the attribute of something or both. Membership relations are transitive. For any notion the owner of its owner is also its owner, and the attribute of its attribute is also its attribute.

3.2. Operations on notion specifications

Notion specifications can be built on the base of other notion specifications by applying operations of concatenation, intersection and exclusion. General form of notion definition is

<name> : <expression>

where <expression> contains operators: "+" – for concatenation, "–" – for exclusion and "*" – for intersection. Operands in expression are specifications represented by explicit set of definitions enclosed in curly braces {} or by names of notions possessing them. The notion is called a base notion if its specification is an operand in expression defining other notion which is called derived one.

The concatenation operator "+" adds to its leftside operand all attributes, relations, constraints that are contained in the rightside specification. The notion defined by concatenation of some base notions contains specification of any its base. Those attributes that appear in result specification from specifications of other notions after concatenation are called inherited unlike own attributes that were explicitly defined in expression inside curly braces.

The exclusion operator "–" selects only those attributes and relations from the leftside specification that are not defined in the right side specification. The latter can contain some notions that are unknown in the left side specification - this notions are ignored. The specification to be excluded can be presented by a notion name or as explicit list of attributes and relations enclosed in curly braces.

Examples:

```
set : { element; };          // notion "set" has specification
                             // containing only attribute "element"
      // "list" notion is defined as concatenation of "set"
      // specification with additional specification containing
      // some details
list : set + {               // "list" is a "set" where
      element {              // any element is owner of its
            index : integer; // ordering number,
            prev,            // previous and
            next : element;  // following element
```

```
        };
        size : integer;
        head : element + { index = 1; } - { prev; };
                // "head" is "list.element" where attribute "index" is
                // equivalent to 1 and there is no attribute "prev"
        last : element + { index = size; } - { next; };
                // "last" is "list.element" where attribute "index" is
                // equivalent to "list.size" and there is no attribute "next"
};
text { length : integer; };
string_list : list + { element : text; };
```

The group of two or more specifications separated by the intersection operator "*" defines new specification containing only those attributes and relations that are defined in every member of this group.

Example:

```
matrix { rows, columns : integer; };
row_vector : matrix + {
        rows = 1;
        size = columns;
};
column_vector : matrix + {
        columns = 1;
        size = rows;
};
vector : row_vector * column_vector;
```

Result of the intersection is the definition:

```
        vector { rows, columns, size : integer; }
```

Relations defined by operators "=" and "<>" are more than simple constraints on the value sets. The operator "=" means also synonymy, mutual concatenation of specifications and establishing of memory links between the elements of task construction. Operator "<>" prohibits the concatenation and making equivalent for the notions that are connected by it and for their derivatives too.

3.3. Task relation

The task relation is a notion whose specification is formed similar to the specification of any other notion. The "Task RELation" notion is predefined in OBRAZ with following description:

```
trel {
        mem;                        // general concept of "memory element";
        input, output: mem;         // "input" and "output" elements are
};                                  // derived from it
```

This specification is interpreted as follows: there is the "task relation" notion TREL that has attribute "task memory element" TREL.MEM and attributes "task input element" TREL.INPUT and "task output element" TREL.OUTPUT derived from TREL.MEM. To specify some notion X as a task relation one should derive it from TREL:

```
X : trel + { ... };
```

Concatenating X with TREL specification gives to X inherited attributes X.MEM, X.INPUT and X.OUTPUT. This attributes should be used as base notions when specifying real task memory elements.

```
matrix {
        rows, columns : integer;
        element {
                row, column : integer;
                row <= rows;
                column <= columns;
        };
}
reverse_matrix : matrix;
reverse : trel + {                  // "reverse" task relation reverses
        A : matrix + input;         // input matrix A and produces
        B : reverse_matrix + output;// output matrix B
}
```

"Task relation" notion has special built-in derivatives: "function", "equation", "program". The "function" notion differs from TREL by the only output element that is implicitly equivalented with the function itself. The task relation is recognized as a function specification when it has the only output memory element with the same identifier as the function one. The task relation presented by function simplifies the specification of compound task relations which may be given in functional form.

```
reverse (A : matrix -> reverse : reverse_matrix);
compound_task (A : matrix -> B : matrix) {
        multiply (A, reverse (A) -> B);
}
```

"Equation" is a task relation where all memory elements are derived from the MEM notion, not from INPUT or OUTPUT. The language interpreter dynamically redefines equation's memory according its current needs during the search of problem solution so that one of the memory

elements becomes the output one and all others form the input of equation. So the equation can be considered as a compact notation defining multiple task relations in one specification.

The "program" notion is a task relation containing some attributes that allow to describe the process of source or object program code generating. The attribute set structure depends on software and hardware environment used to build the final program system and to execute it. The built-in in OBRAZ "program" notion itself is only template and cannot specify any program code generation process, but there are some notions in the construction system knowledge base that are derived from the "program" and define proper attributes to describe construction in various special environments.

3.4. Task relation graph

From the formal view point every notion can be considered as a task relation graph if it contains some task relations or other task relation graphs which all play the role of graph nodes. The joint set of all memory elements of all tasks in the task relation graph forms graph memory. Equivalence of the nodes memory elements represents memory intersections that form graph edges. Every group of mutually equivalented notions with at least one task memory element in the group forms single graph's memory element. All members of such group are synonyms. Every task memory element that isn't connected with any other notion also forms graph's memory element.

4. MPM: Automatic Parallelizing Method

A new automatic parallelizing method called MPM is developed to support construction of safe and efficient PPS. MPM-parallelizer is designed to transform non-parallel algorithm to parallel one.

MPM-parallelizer is an abstract machine for automatic parallelizing of algorithms represented by task constructive objects (Section 2). Input of MPM-parallelizer consists of non-parallel algorithm representation and runtime environment specification. Its output contains representations of parallel algorithm and runtime environment configuration.

4.1. Task Constructive Automaton

Task Constructive Automaton (TCA) is designed for representation of algorithm to be parallelized. From formal point of view algorithm A is considered as the operations set S on which the order relation R is defined (A = <S,R>). If algorithm PA = <SPA,RPA> is derived from algorithm NA = <SNA,RNA> by parallelizing then SPA = SNA and either RPA ≠ RNA or RPA = RNA. If RPA = RNA algorithm NA may not be parallelized.

TCA represents the task constructive object to be executed in parallel environment. It has hierarchical unlimited memory (TCA- memory) with defined input and output subsets (TCA-inp and TCA-out). Intersection between TCA-inp and TCA-out sets is null. If TCA-inp is available (TCA-inp[a]), then TCA is ready to be executed (TCA[re]). When TCA[re] obtains computational resource it becomes executable (TCA[e]). Then output elements of TCA[e] become available (TCA-out[a]).

There are two types of TCA: TCA-unit and TCA-block. TCA-unit has one-level unlimited memory. For example single operation of algorithm may be represented by TCA-unit. TCA-block consists of TCA-units that are related by TCA-block memory relation. TCA-block memory elements set is formed by mapping of union of sets of memory elements of included TCA-units. Intersection between sets of TCA-unit and TCA-block memory elements is null. Communication between TCA-block memory and memory of TCA-unit obeys the following assertions.

- If the memory state condition of corresponding domain of TCA-block memory is "true" then memory access of TCA-unit to TCA-block is allowed to read from TCA-block and write to TCA-unit memory.

- If the memory state condition of TCA-unit output is "true" then memory access of TCA-unit to TCA-block is allowed to read from TCA- unit and write to TCA-block memory. As a result TCA-block memory state is altered.

TCA-block may consist of other TCA-blocks which behaviour is the same as behaviour of TCA-units.

4.2. Parallelizing stages

There may be two different situations of MPM-parallelizing. The first one is outside the PPS-construction system and the second is inside it.

In the first situation non-parallel algorithm is represented by non- parallel source code and parallel algorithm is represented by parallel source code. In the second situation non-parallel algorithm is produced from problem specification and algorithm representation is non-parallel TCA-block. Representation of parallel algorithm is parallel TCA-block.

The first situation differs from the second one by that the automatic parallelizing process includes two additional components: transformation from non-parallel source code to non-parallel TCA-block and transformation from parallel TCA-block to parallel source code.

Thus MPM-parallelizing stages are:

a. Transformation from non-parallel source code to non-parallel TCA-block;

b. Transformation from non-parallel TCA-block to maximum parallel TCA-block;

c. Transformation of the maximum parallel TCA-block to parallel TCA- block implemented in accordance with computational resources specification.

d. Transformation from parallel TCA-block to parallel source code.

The stage b is the main one. This transformation is step by step process that is implemented in accordance with the assumption that computational resources of maximum parallel TCA-block are unlimited.

The stage b. obeys the following rules:

i. If there are one or more TCA-units that are TCA [e] on k-step of parallelizing of non-parallel TCA-block then the transformator includes them in k-box of parallel TCA-block (k = 1,...,N).

ii. If there is none of TCA-units that are TCA [e] on N-step then MPM-parallelizing is finished.

iii. Parallel TCA-block is k-box sequence (k = 1,...,N).

5. Conclusions

At this time PPS-construction system is partially implemented on the basis of PPS-construction methodology. When the system is complete, it may be used for several parallel architectures.

PPS-construction system consists of five interactive components: the knowledge base including PPS-specification editor, the knowledge based parallel programs constructor with automatic parallelizer embedded, the knowledge based runtime support constructor and graphic tools. PPS-construction system produces PPS to be run in specified runtime environment.

Acknowledgements

Author would like to thank his colleagues at Institute for Informatics Problems of Russian Academy of Sciences for their contributions to the research discussed in this paper. Victor Borisov have contributed to OBRAZ language development and have implemented OBRAZ language interpreter. Alexander Ilyin have contributed to MPM-parallelizing development. Boris Kurov have contributed to the preparing of some benchmark programs to test MPM method. Many others have participated in discussion of PPS-construction methodology.

References

[1] J.Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm.ACM 21, 8 (Aug. 1978), 613-641.

[2] J.B.Dennis. First version of a data flow procedure language. In B.Robinet (Ed.). Programming Symposium: Proc. Collogue sur la Programmation (Paris, France, Apr. 1974). Lecture Notes in Computer Science, Vol. 19. Springer-Verlag, New York, 1974, 362-376.

[3] V.D.Ilyin. Program Generating System. Nauka, Moscow, 1989, 264 pp.

An Integrated Hardware-Software Development Environment for Transputer-based Systems (IHSDETS)

Juri Spletukhov
Russian Academy of Sciences
Institute for Informatics Problems (IPIAN)
30/6, Vavilova Str., Moscow, 117900, RUSSIA

Summary

One of the central problems in current parallel systems development is the creation of a practical methodology for hardware-software development which provides a basis for both the description of parallel processing algorithms and their efficient mapping onto the physical resources of parallel systems. This is especially true for private memory MIMD systems where the algorithms are profoundly influenced by the interconnection topology and node-to-node channel bandwidth as well as by the characteristics of the computing nodes themselves.
The IHSDETS system, developed at the Russian Academy of Sciences, supports integrated hardware-software development of Transputer-based private memory MIMD systems. Both the parallel algorithms and the supporting interconnect topology can be varied jointly to verify design and code correctness and determine total processing efficiency.

1. Requirements for Integrated Hardware-Software Development System

Past research into parallel algorithms, with attention to both the classes of problems to be solved on various parallel architectures, and the amount of data to be processed has led most observers to conclude that:

The implementation of parallel algorithms is strongly dependent on the architecture of the multiprocessor computer on which the algorithm runs. Factors such as the processor's speed and memory capacity, number and bandwidth of communication channels between the processors and to secondary storage, and the speed and capacity of secondary storage all must be considered.

This is especially true for real-time problems or problems where there is a great deal of data to be processed. Even then, a variety of parallel algorithms are possible for a given architecture and configuration. In such problems the real issues are development of parallel algorithms which both perform correctly, and yield results quickly enough to be useful.

The systems designer thus must frequently evaluate different parallel implementations of an algorithm for performance and correctness. In addition he must be able to vary the hardware configuration along with various parallel implementations to observe the impact of hardware changes on the software and vice versa.

The advent of new classes of high performance multiprocessor systems such as Transputer Nets (TN) offers new possibilities in parallel processing. TN also require new development environments to capitalize on their potential for parallel processing in various configurations.

First, they require a development environment which allows the designer to explore various mappings of the algorithm onto various TN topologies and various language and program ming features of the subtasks of the algorithm.

Second, numerical measures of efficiency are needed to allow the designer to quantitatively access the performance of competing algorithms as it relates to its supporting architecture. In addition to classical measures such as communications time, new concepts such as "efficiency coefficient" and "acceleration" are introduces to provide these measures.

For example, for each class of problems there is an optimal configuration (in terms of efficiency coefficient) which depends on both the algorithm in question and the amount of data to be processed. Adding more than the optimal number of processes may decrease the execution time, but reduce the efficiency coefficient as well.
In the design of integrated hardware-software systems, it is necessary to consider the following as different aspects of the integrated systems design:
-various methods by which the parallelism inherent in the problem can be expressed;
-different ways of arranging the parallel computations on a given hardware configuration;
-various interconnect topologies of Transputer Nets.
Thus, the efficient utilization of Transputer Nets makes it essential to develop an integrated design tool which supports the following design activities:
-development of algorithms which solve a variety of problem classes;
-coding, debugging, and running different prospective designs on an actual Transputer Net;
-checking the hardware/software design for deadlock condi tions;
-balancing the communications channels and checking the hardware/software design for speed-critical sections of code;
-configuring and re-configuring the Transputer Net topology and internode bandwidth for different classes of problems;
-optimizing the net configuration through prototyping of the parallel computations to determine the efficiency and characteristics of a chosen topology.
In addition, the integrated development environment should comprise advanced problem specification techniques as well as Transputer Net configuration techniques to adequately map the algorithmic parallelism.

2. The Integrated Hardware-Software Development Environment for Transputer Systems

The Integrated Hardware-Software Development Environment for Transputer Systems (IHSDETS) developed at the Institute for Informatics Problems of the Russian Academy of Sciences addresses all of the above requirements. It consists of two parts: the Communications Subsystem and Program Development Tools.

2.1 Communications System (CS)

The CS provides the following functions:
-naming and simultaneous execution of tasks in the Transputer Network;
-synchronous and asynchronous communication between executing tasks. SEND and RECEIVE can be used for data exchange among tasks or SEND can be used to direct data to a single transputer (or host Personal Computer), or broadcast to all Transputers in the Network;
-specification of an actual interconnection topology by means of the Traffic Matrix (TM) of size NXN (where N is the number of Transputers in the network) in two ways: TM can be specified directly or it can be generated automatically by IHSDETS during the initialization of the communications system. The chosen interconnect topology is PHYSICALLY IMPLEMENTED between the Transputers through a fast switching matrix.

2.2 Program Development Tools (PD Tools)

PD Tools provide the following functions:
-automatic configuration of the Transputer Network in basic interconnect topologies (pipeline, ring, tree, 2D array, square array, torus, sphere, etc.);
-design of the parallel subtasks as a set of communicating sequential processes (CP's);
-high level CP algorithm description using COMPUTE, SEND, RECEIVE primitives;
-prototyping of the parallel algorithm and then hierarchical development and implementation in a high level parallel programming language;
-determination of the timing characteristics of the chosen topology and algorithm;
-checking the chosen topology and algorithm for performance and deadlocks.

3. Initial Uses of the IHSDETS

IHSDETS may be used for development of distributed data bases, speech and pattern recognition systems, real time embedded systems, etc. It has been used at the Institute for Informatics Problems in Moscow to develop a library of

parallel programs for solving computationally intense mathe matics problems on Transputer Nets. This library, known as TRANSLIB, has been used for the following classes of problems:
-solution of combined linear algebraic equations with dense, tape or sparse high-order matrices;
-finding eigenvalues and eigenvectors of dense, tape, or sparse high-order matrices;
-solution of large, combined non-linear algebraic or transcendental equations;
-computational integration of Coshy problems for large combined normal differential equations;
-statistics and filtering problems.

TRANSLIB is based on the following design principals:
-universality - the library may be used will all extant programming systems on the Transputer Net;
-transparency - the library is easy to use for a programmer who is not familiar with the intricacies of parallel programming;
-efficiency - the library provides optimal configuration of the Transputer Net topology and algorithms for a wide range of concrete problems.

4. Summary and Conclusions

After using IHSDETS to develop TRANSLIB we have confirmed the following:
-the transparent user interface used by TRANSLIB greatly reduces programmer errors in coding communicating procedures;
-the design time for designing complex parallel hardware-software systems is greatly reduced;
-debugging - including the detection and elimination of deadlocks is greatly simplified.

We are currently investigating the potential for extending the system to other types of processors such as the TMS 320C40 and the T9000. Initial results indicate that software produced on the present system is easily ported to these systems.

Section VI : Parallelisation Techniques, Scheduling and Load Balancing

DYNAMIC LOAD BALANCING IN A PARALLEL PROCESSING SYSTEM

Nilgun Baykal*, Kayhan Erciyes**
*Research Assistant, **Associated Professor
Ege University, Department of Computer Engineering, Izmir,TURKIYE
bilnil@trearn.bitnet, bilker@trearn.bitnet

SUMMARY:
A distributed system consists of a collection of processes connected by a communication network. The random arrival of processes in such an environment can cause some processors to be heavily loaded while other processors are idle or lightly loaded. Dynamic load balancing improves the performance by transferring tasks from heavily loaded processors, where service is poor, to lightly processors where the task can take advantage of computing capacity that would otherwise go unused.

In our study, the parallel/distributed system which could be a Massively Parallel Processing (MPP) system is divided into subsystems called domains, each consisting of a number of processors. A two level load balancing strategy is implemented. At the first level, load is carried out with individual domains where the central node of each domain acts as a centralized controller for its own domain. At the second level, load is balanced among different domains of the system, thus providing a distributed environment among domains. As the first approach a fully distributed method which is not considering real-time tasks is used. As the second approach Bryant & Finkel [1] algorithm is used with the deadline consideration facility added for the real-time tasks.

Keywords: MPP, semi-distributed, domain, load balancing, real-time.

1.INTRODUCTION:

Recent advances in computer hardware production have resulted in parallel processing systems which can consists of hundreds of processors and called as MPP systems. However, the software support such as operating systems, tools, debuggers, for the application are still at infancy in such systems.

We propose a dynamic load balancing model for a parallel processing system, which is based simply on dividing the pool of processors into groups called domains which are managed centrally for various services and distributed for others. A distributed operating system kernel designed provides the necessary group communication in multicast mode for the manager processes. A system process in each domain called central load balancer (CLB) first tries to balance the load within its domain. If this is not possible, it communicates with other CLBs to find a destination node for the candidate process for migration.

We have implemented 3 methods using this model. In the first one, loads of the sending & receiving nodes considered and it is fully distributed. The second approach finds the destination node by using the load information of the other domains instead of randomly. Finally, dynamic load balancing of aperiodic processes of a real-time application are considered by modifying the Bryant & Finkel's algorithm for real-time processes.

2. FULLY DISTRIBUTED APPROACH:

Dynamic load balancing solves the remapping problem in a MPP system at run time, where many processors are needed to be allocated evenly to multiple processor nodes. The aim is to migrate processes from busy to idle nodes in order to achieve higher resource utilization. Even in a homogenous distributed system, at least one processor is likely to be idle while other processors are heavily loaded.

In order to decide whether a processor is heavily loaded or lightly loaded, a load index must be determined. Load index predicts the performance of a task if it is executed at some particular node. Load indexes that have been used usually are the length of the CPU queue, the average CPU queue length over some period, the amount of available memory, the context switch rate, the system call rate, and the CPU utilization [2].

In this study, a semi-distributed dynamic load balancing model is developed for a distributed memory computer system and the cpu queue length is used as the load index. In this case, the processors of an MPP system is divided into domains of fewer processors. Each domain has a CLB which communicates with central load balancers of other domains in a distributed manner.

Table.1 Intradomain Load Balancing Algorithm

```
Sender=Deque(Sender_List);
Receiver=Deque(Receiver_List);
DO
{
 While ((Sender.node_load != AVG) AND (Receiver.node_load !=AVG))
 {
  msg="Select and mark a process to migrate"+Receiver.node_id;
  Send_Msg(Sender.node_id,msg);
  msg=Sender.node_id;
  Send_Msg(Receiver.node_id,msg);

  Sender.node_load= Sender.node_load - 1;
  Receiver.node_load= Receiver.node_load + 1;
 }
 If (Sender.node_load > AVG )
   Receiver=Deque(Receiver_List);
 Else
 {
   If (receiver.node_load < AVG)
     Sender=Deque(Sender_List);
   Else
   {
     Sender=Deque(Sender_List);
     Receiver=Deque(Receiver_List);
   }
 }
}UNTIL (One of the lists is EMPTY)
```

This model proposes a two level load balancing strategy. At the first level, load balancing is carried out within individual domains where the central node of each domain acts as a centralized controller for its own domain. At the second level, the load is balanced among different domains of the system, thus providing a distributed environment among domains. The design of such a strategy involves designing an algorithm for performing optimal task scheduling and load balancing within a domain as well as among domains, and developing efficient means for collecting state information at inter domain and intradomain levels.

Table.2. Interdomain Load Balancing Algorithm

```
Execute in parallel
{
 If (mynode_type=SENDER)
 {
  If Sender_List still has elements
  {
   Send a multicats message including domain number and load information of the domain
   to the CLBs of other domains.
   Wait for reply
    If there is any reply
    {
      Form a pair with the replying CLB.
      Receive message including load information of the receiver node
      Find the number of processes to be migrated.
      Determine the which processes to be migrated from which nodes of the domain.
      Migrate the selected processes to the replying CLB.
    }
  }
 }
else  /*Receiver node*/
{
   Wait for any messages from other domains.
    If there is a message showing that a domain wants to send you some of its load
    {
       Send a message to that domain including your node id and form a pair with it.
       Send your load information to the destination node.
       Determine the which nodes can accept new processes.
       Take the processes coming from the source CLB and send them to the appropriated nodes
of the domain
     }
  }
}
```

Central load balancers are responsible for dynamically assigning processes to individual nodes of the domain, transferring the load to other domains if required, and maintaining the load status of the domain and nodes. As a result of load balancing and sharing, a process can be completed earlier due

to the utilization of otherwise idle or lightly loaded processors. Thus process migration is used as a tool for dynamic load balancing among processors of an MPP system.

In our system, each node in a domain periodically sends its CPU queue length to the CLB of that domain. CLB collects this data and determines an average load value for its domain. By adding and subtracting a tolerance value to that average it finds an interval. Then CLB starts to decide in which list to put each node in its domain. It has three lists named, Balanced_List, Sender_List and Receiver_List. If the load value of a node is within this interval, this node is said to be balanced and put into the Balanced_List. If the load value of the node is greater than this interval it is said to be heavily loaded and put into the Sender_List, otherwise it is lightly loaded and put into the Receiver_List. Sender_List is sorted as the most heavily loaded node is the first element. Receiver_List is sorted as the most lightly loaded node is the first element. An algorithm which tries to balance load among nodes of a domain in a proper way is then executed and nodes which will perform process migration are marked. CLB informs the sender node (source node) and receiver node (destination node), for migration and wants to source node to determine a process to migrate (Table.1). While determining a process for migration some criteria must be considered. For example a process, having short code, long remaining execution time, short data and less communication links is appropriate for migration.

After performing intradomain load balancing process, if there are nodes that are still heavily loaded or lightly loaded, inter domain load balancing algorithm given in Table.2 is executed.

As it can be understood from this algorithm, sender and receiver domain pairs are formed randomly, since the CLB of the receiver domain selects the host of the first coming message as the destination domain. This approach make sense when the most lightly loaded domains and most heavily loaded domains form a pair as a chance. Otherwise its performance is low. So we also used another heuristic when selecting domain pairs. At that time CLBs of all domains know the global state information and domains are sorted according to their loads and most heavily loaded domains and most lightly loaded domains are formed pairs. It is observed that by using this heuristic load is distributed more evenly among domains.

3. REAL-TIME APPROACH:

In a conventional multitasking operating system, processes are interleaved with higher importance (or priority) processes receiving preference. Little or no account is taken of deadlines. This is clearly inadequate for real-time systems. These systems require scheduling policies that reflect the timeliness constraints of real-time processes.

A realistic hard real time system must have the following properties:

- guaranteeing both periodic and non-periodic hard real-time processes on the same processor.
- utilization of spare time by non-critical processes.
- initial static allocation of periodic processes.
- migration of aperiodic processes for response to changing environment conditions or local overload.

Schedulers produce a schedule for a given set of processes. If a process set can be scheduled to meet given pre-conditions, the process set is termed feasible. A typical pre-condition for hard real-time periodic processes is that they should always meet their deadlines. An optimal scheduler is able to

produce a feasible schedule for all feasible process sets conforming to a given precondition. For a particular process set, an optimal schedule is the best possible schedule according to some pre-defined criteria. Typically a scheduler is optimal, if it can schedule all process sets.

Our system model is described below:

- Both periodic and aperiodic processes are allowed.
- Computation times for a given process are constant.
- Computation_time≤ Deadline≤ Period (i.e. the processes have computation time less than their deadlines, and the deadline is equal to their period at worst).
- Precedence constraints among processes are enforced by using the process's start times and deadlines.
- No process resource requirements are considered.
- Context switch have zero cost.
- Multi-node, multi-domain systems with dynamic process allocation.

3.1. Deadline Characteristics:

The maximum response time of a process is termed as the deadline. Between the invocation of a process and its deadline, the process requires a certain amount of computation time to complete its execution. Computation times may be static, or vary within a given maximum and minimum. The computation time must not be greater than the time between the invocation and deadline of a process.

$$\text{Computation_time} \leq \text{Deadline}$$

Periodic processes are characterized by their period and their required execution time per period. For each periodic processes, its period must be at least equal to its deadline. That is, one invocation of a process must be completed before successive invocations. This is termed a runnability constraint.

$$\text{Computation_time} \leq \text{Deadline} \leq \text{Period}$$

The activation of an aperiodic process is essentially, a random event and is usually triggered by an action external to the system. Aperiodic processes also have timing constraints associated with them; i.e. having started execution they must complete within a predefined time period. It is not guaranteed that aperiodic processes will certainly meet their deadlines. If it is not possible to schedule an aperiodic process on any processor before its deadline, this process is said to be unschedulable. Non periodic processes can be invoked at any time.

A well defined dynamic load balancing system must ensure that processes that miss their deadlines are non-critical; i.e. the order in which processes miss their deadlines has to be predictable. Thus, non-critical processes can be forced to miss their deadlines.

It has been showed that the algorithms that are optimal for single processor systems are not optimal for increased numbers of processors. In a multiprocessor or distributed system processes that are considered likely to miss their deadlines have to be migrated to other processors. But it has also been showed that it is better to statically allocate periodic processes rather than let them migrate and, as a consequence, potentially downgrade the system's performance [3].

3.2. Scheduling Policy:

Each process is characterized by (A,S,C,D) known at the time of process arrival, where A is the process's arrival time, S is the earliest possible time at which its execution may begin (start time), C is the maximum computation time and D is the deadline by which it must complete its execution. As we mentioned above, processes may be aperiodic and periodic. An aperiodic process is described by (A,S,C,D). For each periodic process, an (A,C,P) describes its arrival time, computation time, and period. Given this tuple, for each of its periods additional tuples (A,S,C,D) may be derived.

A process is preemptable if its C units of computation time can be satisfied by one or more time slots that sum to C. Preemption is important in dynamic real-time systems for three reasons. First, process performing exception handling may need to preempt existing processes so that responses to exceptions may be issued in a timely fashion [4]. Second, as with exception handling processes, different levels of criticalness expressing process importance also exist among tasks in real-time applications [5], again prompting systems to permit process preemption when possible. Third, more efficient schedules can be generated if preemption is allowed [6].

The algorithm schedules sets of processes ordered by increasing deadlines. Given such a set, the algorithm selects the first process and schedules it as near to its start time as possible (i.e. at the earliest available time after its start time). The process is scheduled by simply accumulating all unused processor time past the process's start time until sufficient computation time is found. If the resulting schedule permits this process to complete before its deadline then the process is schedulable, else it is unschedulable.

The scheduling information used by this algorithm is recorded in a doubly linked list. Each element of the list represents a time slot already assigned to a process, and has four fields: starting time, ending time, a pointer to the next list element, a pointer to the previous list element.

Given this list, a process's schedulability is analyzed by searching the list for available time intervals between two elements. This search starts at an element compatible with the process's start time and ends at a time point compatible with the process's deadline or when the accumulated length of available time is equal to the process's computation time. The process is schedulable if sufficient computation time is found before its deadline during this search, else the algorithm reports the process as unschedulable. After a process is scheduled the list is updated.

In our system aperiodic processes don't have hard deadlines, hence they can be migrated to other processors when they can not be scheduled on present processor. On the other hand periodic processes have hard deadlines and they can not be migrated to other processors. They are statically allocated when the system starts.

In order to schedule an aperiodic process dynamically Bryant & Finkel's algorithm is used.

3.3. Bryant & Finkel's Algorithm:

Bryant & Finkel's algorithm is a dynamic and physically distributed algorithm. In our system, we used Bryant & Finkel's algorithm in a semi-distributed fashion, considering the deadlines of the aperiodic processes. To make decision, processors cooperate by sending negotiation messages. The decisions are sub optimal and heuristic approach is used to find solution.

A newly arriving aperiodic process can be called as schedulable only if its scheduling does not danger previously scheduled processes. First, the new aperiodic process is placed in order into the list, which holds all previously scheduled processes on this processor. Then processes in the list is rescheduled, using the algorithm explained above. If any of the previously scheduled processes is unschedulable now, then newly arriving aperiodic process is determined as unscehdulable on this processor. Otherwise it is schedulable. When a process is determined as unschedulable at that node, a timer starts to work. When the timer reaches the value that is equal to deadline minus the execution time of that process, then this process is said to be unschedulable else where.

As in the fully distributed approach, unschedulable newly coming processes are tried to be scheduled at intradomain level. Each node in a domain sends its schedule list and unschedulable aperiodoc processes list to the CLB periodically. CLB collects this information, then tries to find appropriate empty time slots on different nodes of its domain, for unscheduled aperiodic processes. If it can find such a node, then this node is determined as destination node, and process migration takes place. Otherwise the process is said unschedulable within this domain. If such a condition occurs the strategy works in the interdoamin level in the following way:

1. The CLB of domain A (CLB_A), sends a query to one of its nearest neighbors CLB of domain B (CLB_B), to form a temporary pair, which enables a controlled, stable environment suitable for process migration. The query has two purposes:

a. it informs the CLB_B that CLB_A wishes to form a pair, and

b. it contains a list of processes and time constraints for each processes.

2. CLB_B after receiving the query can perform one of three options:

a. rejecting CLB_A 's query; this implies that CLB_A must send a query to another neighbor domain

b. form a pair with CLB_A; this implies that CLB_A as well as CLB_B reject all incoming queries until the pair is broken.

c. postpone CLB_A when CLB_B is in a migrating state, that is, sending processes this implies that CLB_A must wait until CLB_B forms a pair with it, or rejects it- CLB_A cannot query anyone else.

3. After establishing a pair, CLB_A sends unschedulable aperiodic processes list to the CLB_B. Then CLB_B broadcasts this information to all of its nodes. Nodes try to schedule these processes on their own schedule table. If this is possible, scheduled processor id and its response time is returned to the CLB_B. As the last step, CLB_B compares the response times of the same processes on different nodes and selects the node giving the minimum response time as the destination node.

4. If no processes can be executed on CLB_B, then CLB_B informs CLB_A of this fact and pair is broken. Otherwise the processes are migrated. This process is repeated for all remaining processes, until no process is left.

4. IMPLEMENTATION AND RESULTS:

We implemented described load balancing mechanisms on the iPSC/2 hypercube simulator on OSF/1 MACH Unix environment which can simulate nodes up to 128. The existing kernel of the hypercube simulator (Nx/2) is improved by adding group communication facility. The semi-distributed central dynamic load balancers are system processes which communicate with other load balancers to perform load transfer. Process migration facility is simulated.

In the figures given below, S values show the number of activations of the load balancing algorithms. In Figure 1 a system having 32 nodes are divided into 4 domains. Each domain has 8 processors and during inter domain balancing, heavily and lightly loaded domains formed pairs randomly. In Figure 2 domain numbers and numbers of processors on domains are the same, but domains formed pairs according to their load (sorted method) during the inter domain load balancing.

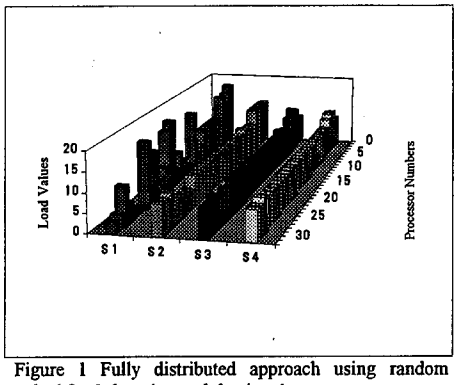


Figure 1 Fully distributed approach using random method for 8 domains each having 4 processors.

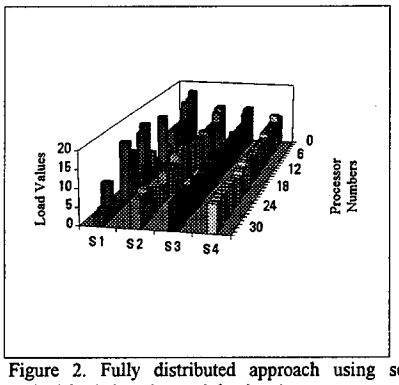


Figure 2. Fully distributed approach using sorted method for 8 domains each having 4 processors.

In Figure 3 and Figure 4 domain number is selected as 4 and each domain has 8 processors. Figure 3 shows the randomly paired inter domain load balancing and Figure 4 shows the inter domain load balancing according to sorting rule.

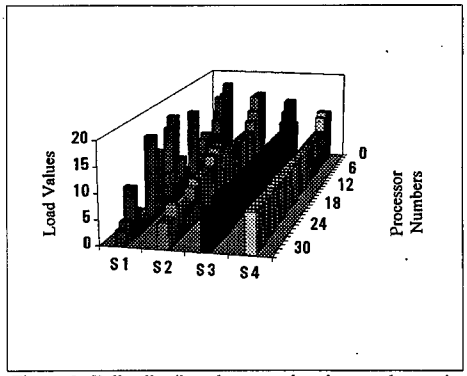


Figure 3. Fully distributed approach using random pair method for 4 domains each having 8 processors.

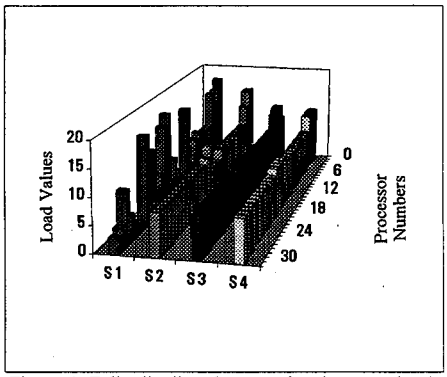


Figure 4. Fully distributed approach using sorted pair method for 4 domains each having 8 processors.

In Figure 5 domain number is selected as 2, and this time each domain has 8 processors. In Figure 6 domain number is again 2, but each domain has 16 processors. It can be seen that increasing processor numbers on each domain algorithm distributes the load in less steps than the otherwise.

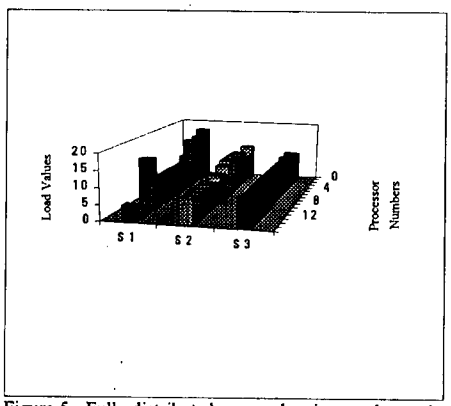


Figure 5 . Fully distributed approach using random pair method for 8 domains each having 2 processors.

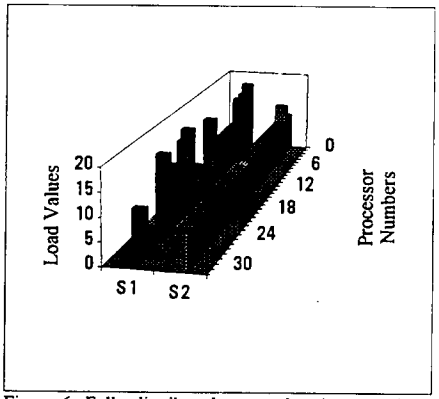


Figure 6. Fully distributed approach using sorted pair method for 16 domains each having 2 processors.

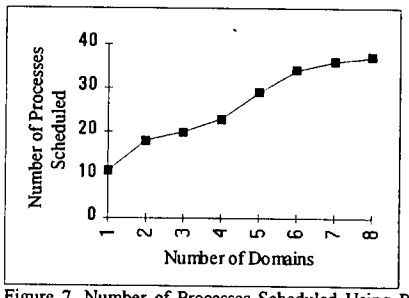


Figure 7. Number of Processes Scheduled Using Real-time Approach For Varying Number of Domains

Real-time approach is implemented and system's performance is observed for different number of domains and process numbers. Schedulability ratios for 50 processes on varying number of domains are represented in Figure 7. Each domain has 4 processors in this case. It can be deduced from the figure that as the number of domains in the system increases, schedulability chance of the processes also increase.

5. CONCLUSIONS:

The starting point of our research was the fact that neither fully distributed nor centralized load balancing policies yield good performance for MPP systems. We therefore designed a semi-distributed model which makes use of both approaches. The iPSC/2 hypercube simulator unfortunately created problems after 64 processors, so the maximum domain number we could test is 16 each with 2 processors. As can be seen from the figures, the random load balancing gives similar results as fully distributed when the number of domains is low. However, when the number of domains increase to 16 for example, we observe very quick response in fully distributed policy with respect to random policy as depicted in Figure 6. For the real-time case, we observe that the

scheduled aperiodic real-time processes rise sharply as the number of domains increase which is expected.

Our work is ongoing and we need to do more experimenting especially for the real-time policy. Also, more distributed approaches other than Bryant&Finkel's algorithm are to be investigated for the real-time application case.

6. REFERENCES:

1. Bryant R.M., Finkel R.A., "A Stable Distributed Scheduling Algorithm", Proceedings of the 2nd International Conference on Distributed Computing Systems, 1981.

2. I. Ahmad, A. Ghafoor, "Semi-distributed load balancing for massively parallel multicomputer systems", IEEE Trans. Soft. Eng., Vol.17, No.10, Oct. 1991.

3. Audsley N., Burns A., "'Real-time system scheduling", Technical Report, Department of Computer Science, University of York, UK.

4. Schwan K., Zhou H., "Dynamic scheduling of hard real-time tasks and real-time threads", IEEE Trans. Soft. Eng., Vol.18, No.8, August 1992.

5. S.R. Biyabani, J.A. Stankovic, and K. Ramamritham, "The integration of deadline and criticalness in hard real-time scheduling", in Proc. Real-Time Systems Symposium, 1988.

6. B.A. Blake and K.Shwan,"Experimental evaluation of real-time scheduler for multiprocessor system", IEEE Trans. Soft. Eng. Vol.17, Jan.1991.

Mapping and scheduling data-parallel dataflow programs on massively parallel architectures

Abderrahmane MAHIOUT, Jean-Louis GIAVITTO, Jean-Paul SANSONNET

LRI - U.R.A. 410 du CNRS
Bât. 490 - Université Paris-Sud
F-91405 Orsay cedex - France
email : mahiout@lri.lri.fr

Abstract : We propose a technic for mapping and scheduling numerical programs on new massively parallel architectures, through a set of heuristics. These heuristics are based upon a modelisation of the program and the architecture. They are designed to take into account both data and control parallelism with interprocessor communications. We present some experiments done on a set of dataflow graphs, given some architectural characteristics, for a better understanding of our heuristics behaviour, and we analyse the results obtained.

Key-words : Mapping, scheduling, dataflow graphs, massively parallel architectures.

1. Introduction

The new generation of massively parallel architectures enables the simultaneous exploitation of two kinds of parallelism: data and control parallelism (see for example [1], [2]). The main contribution of this paper is in setting a new abstract framework for the automatic distribution and static scheduling of data-parallel dataflow programs [3] on such architectures. Our goal is to provide a general technic for the automatic data distribution and task scheduling, independently of any architectural specificities, such as network topology or some particular architecture granularity. This article is structured in the following way. Section 2 outlines the context we work in. Next, we describe our modelisation of a dataflow data-parallel program, and the architecture model we shall use. In the following two sections, we'll present the principle of our heuristics and some performance evaluation of two versions. At last we'll conclude by analysing the results and outlining some future experimentations.

2. The context

The programs we address are those from numerical intensive applications. The kind of parallelism which can be extracted from is mainly the data-parallelism. However, as we'll see later, it can be useful to consider also control parallelism for efficiency reasons. We consider these programs having enough informations at compile-time to choose a static approach for mapping and scheduling and thus avoid the run-time overheads of a dynamic solution. However, we cannot reasonably completely avoid a dynamic solution because the presence of data-dependent instructions. For the moment, we will consider only parts of the program which execution time and data amount can be completely determined at compile-time. We also consider only a non-pre-emptive scheduling of tasks.
The architectures we consider are the new massively parallel computers with distributed memory. These new architectures are mainly characterised by a high bandwidth low latency network. We want to consider neither some specific topology nor a particular number of processors, but we take into account a mean interprocessor communications time [4], [5].
Because the mapping and scheduling problem with interprocessor communications is NP-hard [6], we propose a solution through a set of heuristics which come from works done in operational research. This set of heuristics follow the principle of list-scheduling. Because the list scheduling strategies are particularly

well suited to handle control (or concurrent) parallelism, we enhanced it with a mapping point of view which is rather linked to data-parallelism. The performance criterion of our mapping and scheduling is the makespan, i.e. the maximum of end-times of all the tasks.

3. Program modelisation

3.1 An enhanced dataflow graph

A convenient tool to describe a program from the mapping and scheduling point of view is the dataflow graph [7], where nodes are the tasks of the program and the edges represent both data dependencies (i.e. a data communication between them) and a chronological order of computation. This dataflow graph comes from the partitioning of the program source in many tasks. Given an architecture, the grain of partitioning the program in tasks will have an impact on the mapping and scheduling performances. The dataflow graph is the representation of the partial order of execution between tasks, so it is easy to extract some threads of execution which can be executed in parallel. This model is very convenient for expressing the control parallelism, but lacks the handling of data-parallelism.

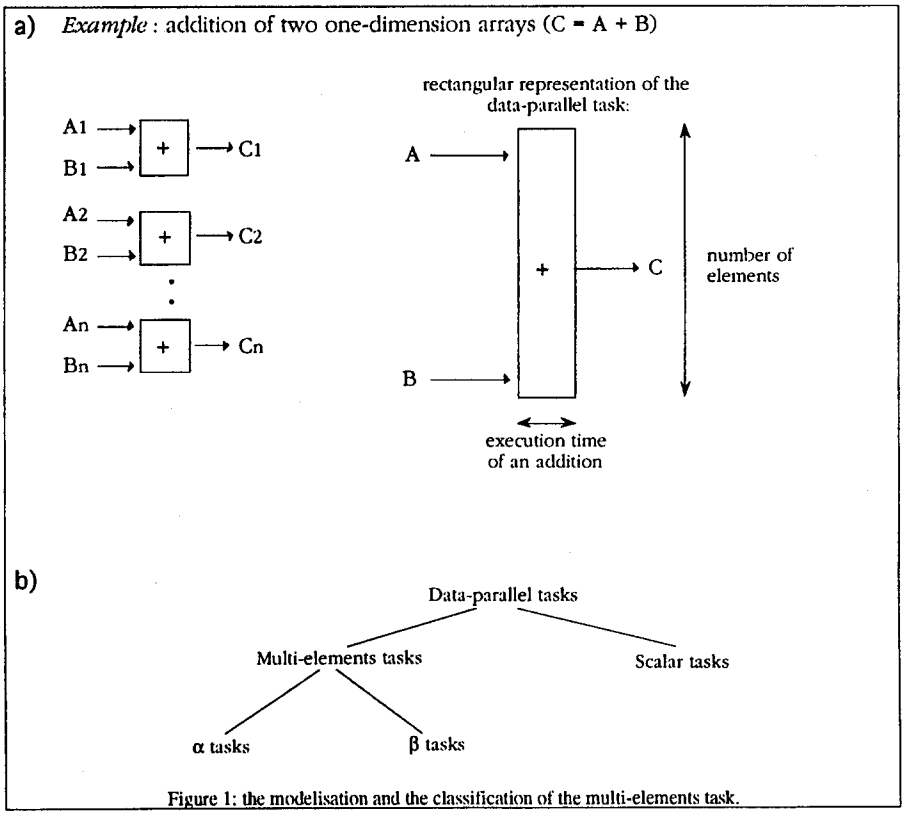


Figure 1: the modelisation and the classification of the multi-elements task.

In a first approach, data-parallelism corresponds to the execution of a sequence of instructions simultaneously on a collection of data (typically an array of data), for example, the addition of two vectors

of 100 elements [8]. We may consider this addition modelled by 100 tasks which add two corresponding elements of the vectors. However, such modelisation is not very realistic in practice because the size of the data. The situation can also be modelled by one *multi-elements task* which has two arguments (the two arrays) and produces a new array. Finally, a dataflow node will have two dimensions: its duration and the number of elements it processes. These two dimensions define the *geometry* of the task (see fig. 1,a).
The data-parallel component of a task represents the ideal number of processor elements for a maximum exploitation of data-parallelism. As we'll see below, providing the maximum number of processors to the task do not always lead to a maximum efficiency.

3.2 A classification of multi-elements tasks

For a finer management, we distinguish two groups of multi-elements tasks (see fig. 1,b): 1) the scalar tasks, which have a data-parallel dimension equal to one, and 2) data-parallel tasks whose data-parallel dimension is greater than one. This distinction is motivated by the observation of numerical program sources, where some tasks are intrinsequely data-parallel, when there are tasks which represent computations between scalar numbers. We can refine this classification and separe the data-parallel tasks group in two subgroups : 1) the α tasks and 2) the β tasks. An α task is a data-parallel task without intrinsic communications between the elements of the task, for example, the addition of two vectors. A β task is a data-parallel task characterised by the presence of intrinsic communications between the elements of the task, as for example,in the reduction of a vector. We distinguished these two subgroups for a better approximation of the dependencies between the tasks and the evaluation of execution times of them: we must include intrinsic communications in the execution time of a β task while an α task has the same duration as a scalar task.

3.3 The modelisation of dependencies between the multi-elements tasks

Enhancing the dataflow graph with the data-parallel dimension leads us to reconsider the meaning of dependencies between two tasks. In a classical single-element task dataflow graph, an edge between two tasks is explained easily in terms of data communications between the two tasks which will be translated to a message passing communication between the predecessor task processor and the successor task one. In a data-parallel context, because we consider subsets (or intervals) of processors for each multi-elements task, the edge between two tasks may reflect many communication schemas (at most, the number of possible elements permutations). For practical reasons, we cannot modelise all the possibilities, so we have to find a criterion that reduce the combinatorial complexity.
Taking into account the use of our model (the data-parallel dataflow graph) in the mapping and scheduling problem leads us to keep only two kinds of dependencies between two tasks. We have a P (point to point) dependency if an element of the successor task depends only on one element of the predecessor task and conversely, we do not have more than one dependence between an element of the predecessor task and an element of the successor task.
All other kinds of dependencies are summarised by a T (total) dependency.
This modelisation of the dependencies between the different kinds of tasks is motivated by the notion of locality. The execution time (or makespan) of a program on a massively parallel architecture is bounded by the network performances, so it is necessary to avoid as much as possible communications between processors. This goal is conflicting with exploiting the maximum of parallelism available in a program, therefore it is necessary to find a compromise which depends, given an architecture, on the partitioning of the program and the mapping and scheduling strategy.
If two multi-element tasks Ti and Tj have a P dependency between them, mapping Ti on the same interval of processors than the Ti one avoid communications between the two tasks. But if we have a T dependency between them, a gap of communications is spent, whatever the physical mapping we choose.
To put a dependency flag on an edge of our data-parallel dataflow graph, it is necessary to consider all the possible couples of tasks. As we saw before, we have three kinds of tasks (scalar, α and β tasks) and two types of dependencies (P and T), this gives us 18 possibilities. But, there are some of them which are impossible, in accordance with this semantics. For example, it is not possible to have a P dependency between a multi-element task and a scalar task.
We have an overview of all the possibilities in fig. 2. For a better understanding, let us take two examples which illustrate the rules we applied for choosing the types of dependencies.
The case n°2 is forbidden because the two tasks do not have the same number of elements. This case represents the broadcast of the value produced by a scalar task to all the elements of an α task. So, a gap of communication must be spent and a T dependency is required (see the case n°5).

1	s —P→ s	Ok	10	α —T→ s	Ok
2	s —P→ α	No	11	α —T→ α	Ok
3	s —P→ β	No	12	α —T→ β	Ok
4	s —T→ s	No	13	β —P→ s	No
5	s —T→ α	Ok	14	β —P→ α	No
6	s —T→ β	Ok	15	β —P→ β	No
7	α —P→ s	No	16	β —T→ s	Ok
8	α —P→ α	Ok	17	β —T→ α	Ok
9	α —P→ β	Ok	18	β —T→ β	Ok

S : scalar task α : alpha task β : beta task
P : P dependency **T** : T dependency

Figure 2: overview of all the dependency configurations

Case n°10 represents a selection of one element of the vector resulting from an α task to use it as an entry of a scalar task. But, at compile-time, we cannot know the element to select, so, we have to wait the computation of all the elements of the α task, and add a gap of communication to let the data reach the processor we mapped the successor task on. Therefore, when we have a T dependency, the locality criterion is useless, and the mapping will depends only on the processors availability, given a scheduling time.

To summarise, we enhanced the dataflow graph with the data-parallel dimension and a modelisation of the dependencies between the multi-elements tasks (fig. 3).

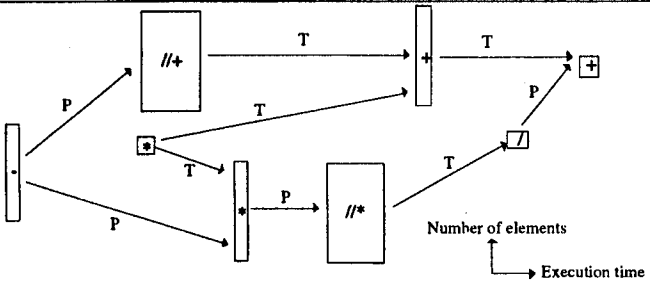


Figure 3: a data-parallel dataflowgraph

4. The architecture modelisation

To choose an architecture model, we must before ask a question: what are the main characteristics of the new massively parallel architectures which have an impact on the choice of a mapping and scheduling strategy ?
The new architectures are characterised by a marked improvement of network performances which leads to quite lower standard deviation of communication costs between any pair of processors. So, the architecture topology parameter tends to be useless in the mapping and scheduling strategy. Moreover, the network topology (fat-tree for the CM5, ring hierarchy for the KSR, etc.) do not match with the data topologies we meet in numerical problems (for example, meshes in finite difference schemes).
All these observations lead us to adopt a homogeneous communication cost among any processor pair: IPC = Ax + B. The first parameter A is the communication cost per unit of data, the second one B is the start-up duration of the communication. We can use either values of existing architectures or any other ones. The third parameter which decribes the architecture is the number of processors (Npe). The last one is the execution time (Tex) of an arithmetic operation, this parameter allows to include the processor performances in the architecture model.
We assume that the architecture allows computations and communications simultaneously, i.e. a specific network interface performs the communication while the processor computes.
For the time being, we take into account neither communication pipelines nor vectorial processors. Theses topics will be included in the next versions of the architecture model.

5. Principle of the heuristics

5.1 Introduction

Our goal is to provide mapping and scheduling technics which are both compatible with the data-parallel nature of the target applications and the exploitation of the multi-threading we can find in these applications.
In the data-parallel mapping problem, many solutions were proposed. Generally, they were based on mapping of the data, and by implication, computation follows. Data arrays are mapped with compiler directives, and computation follows implicitly by the 'owner computes' rule. With this rule, the computation required to assign an expression value to a data array element is performed by the processor to which that data element has been mapped. This approach suffers of the two following drawbacks : 1) the alignment/distribution are linked to the data, which implies a further step to map the computation, and 2) the information about data-mapping is explicitly required from the programmer. This approach is then low-level but efficient if the programmer is aware of the target architecture. Even so, it is very difficult to include the topology of the architecture in a 'manual' mapping approach because (as we said before) the architecture topologies do not always match with the data topology of an application. Moreover, the improvement of the data-mapping we may obtain can be lost by the owner computes rule. This rule has the advantage of simplicity but in some cases lacks the efficiency.
We propose to infer a static mapping and scheduling for each expression. Because the mapping is linked to expressions, that is to computations, there is no need of two separate phases. A multi-element task is both a representation of a data-parallel computation and a linear representation (typically, a one-dimensional array) of the data which are handled by. This linear representation of data is well suited to the architecture model we work on (no topological feature). Of course, numerical applications do not manipulate only one-dimensional arrays, but multi-dimensional ones can always be decomposed on many one-dimensional arrays (for example, a matrix N × M can be seen as N one-dimensional arrays of M elements each). The mapping of the multi-element task is both the mapping of the computations of each element and the mapping of the resulting array (for a β task, we consider that the result (a scalar) is on all the processors used for the computation at the end of the execution).
The exploitation of multi-threads allows a better processor utilisation. Even if the data-parallelism is preponderant in such applications, control parallelism can be used to hide communication costs of data-parallel computation. A dataflow graph expresses naturally the number of threads we can compute simultaneously. It has been used as a support in many fields, specially in the scheduling problem. The partial order of tasks execution leads us to have more than one task to choose at each level of the scheduling. So we must put a priority criterion to order the list of tasks which are candidates: this is the list scheduling principle. There are several possible priority criterions, such as the number of successors of the tasks, the critical path, etc.
For the time being, we have chosen two strategies of list-scheduling

In the first one, the criterion is the critical path and an added criterion, the 'area' of the task (the number of elements of the task by the execution time). We greedily fill the list of ready tasks : a task is put on the list as soon as it is possible, i.e. when all its predecessors have been mapped.
The second version differs only on the way the ready task list is managed. In this version, we order the task according to its rank (roughly, the rank of one task is its depth-first distance from the root in the dataflow graph). We put a task of rank i in the ready task list only if all the tasks of the i-1th rank have been mapped. The same priority criterions than the first version are applied inside a rank.

5.2 Splitting the multi-elements tasks

Handling multi-elements tasks as a whole leads to specific problems, one of them is the 'splitting' of the tasks (this notion is similar to the folding of tasks [9]). When we want to map a multi-elements task, it could be possible that the resources available (the processors) at the current time are not sufficient. So, we have to map a 'piece' of the task on the available processors and postpone the mapping of the remainder of the task (fig. 4).

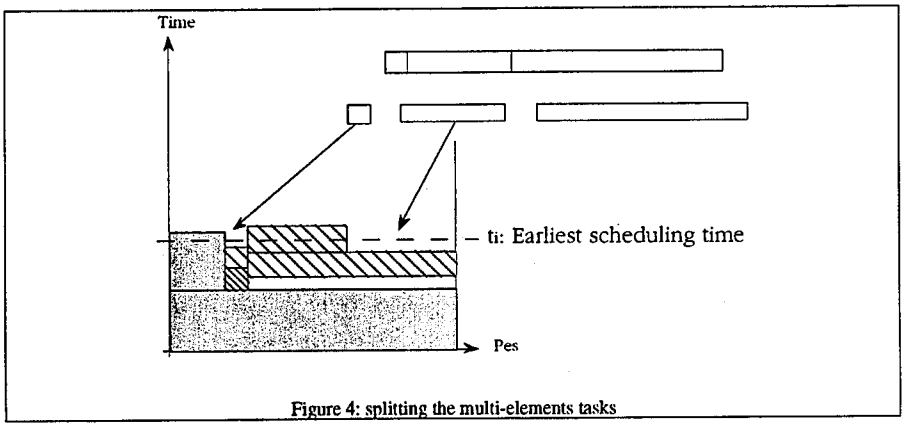


Figure 4: splitting the multi-elements tasks

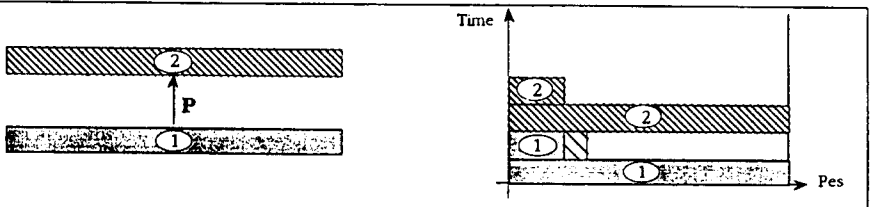
5.3. Exploiting locality

The splitting have an effect on the complexity of locality handling. Remember that we modelise the task dependencies with annotated edges (P or T dependency) which tells the heuristic whether it is possible to take advantage of locality (P dependency) or not (T dependency). Exploiting locality consists on mapping the successor task onto the same interval of processors than the predecessor one, if both tasks are linked with a P dependency. So when we have to map a task, we must ask some questions in order: 1) are there any P dependency edges with the predecessor tasks, 2) if so, we have to check some constraints to know if the locality advantage is preserved. These constraints are: the earliest mapping time imposed by the eventual T dependencies with predecessor tasks and the earliest time imposed by the processors availability (fig. 5).

5.4 The cost function

Splitting the tasks leads to not consider the mapping of all the task at once but a piece of it : the unity of work is a piece of the task. Because tasks may be mapped among several intervals of processors, we have to consider the exploitation of locality between pieces of the predecessor task and the corresponding pieces of the successor task, if these tasks are linked with a P dependence. So the number of pieces to consider depends on the splitting of all the predecessors of the task to map which have a P dependence with it. The problem is then to choose which piece will be mapped first, between all the possible pieces of the task to map. So we have to order the different pieces of the task to map. This ordering is provided by a cost function that gives us an evaluation of the mapping of all the pieces of the task. This cost function

depends on 1) the earliest time the piece of the task can be mapped, under the constraints we discussed before, and 2) on the number of elements of the piece.



If End(1)+IPC < Earliest_time(2) then the mapping of task 2 follows the mapping of task 1.

Figure 5: exploitation of locality

As an example, let's take the graph in fig. 6. The task 4 have P dependency with the tasks 1 and 3 and a T dependence with the task 2. The splitting of the tasks 1 and 3 implies the number of pieces of the task 4 to consider (here we have 3 pieces). So, we set up an evaluation table whose dimension is the number of pieces to consider by the number of P dependence predecessor tasks (here 3 × 2) the table is then filled with the cost function values (negative values correspond to the case where the exploitation of locality is useless). As we see in this example, the best gain is obtained for the mapping of the piece n°1 of the task 4 on the same interval of processors than the corresponding piece of the task 3.

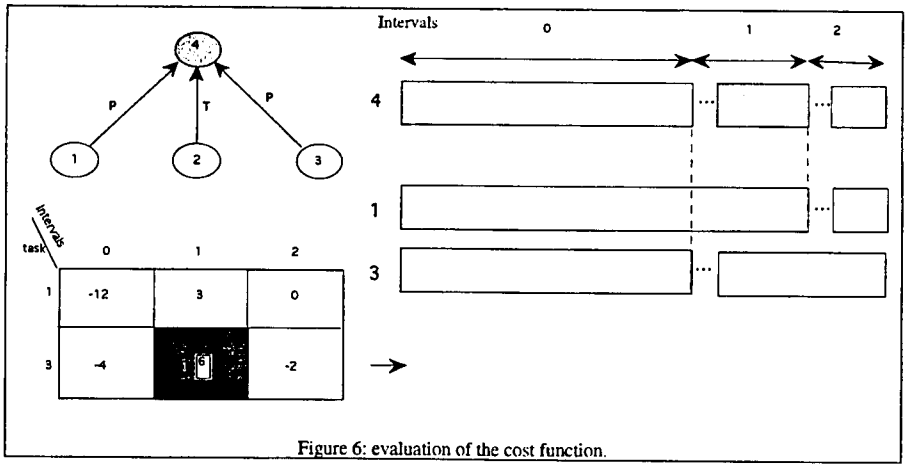


Figure 6: evaluation of the cost function.

6. The heuristics performance evaluation method

We have tested the two versions of the heuristic on a set of data-parallel dataflow graphs which were generated by a parametric graph-generator. This graph generator allows to choose some topological parameters (the length and the width of the graph, etc.) and other parameters like for example, the percentage of α tasks. For simplicity reasons, we considered the number of elements of the multi-elements tasks as constant.

For these first experimentations, we generated 100 graphs which have a length of 30, a width of 10, 80% of multi-elements tasks and 50% of α tasks. We fixed the number of elements of the multi-elements tasks to 1024.

We take architecture parameters from the CM5 communication and processor characteristics (A = 90e-8, B = 8300e-8, Tex = 12e-8). The number of processors Npe is 64.

The behaviour we wanted to test is the evolution of the makespan among different data granularities.
The data-granularity G is the number of data elements computed by one processor. For example, If we apply a granularity equal to 4, a multi-element task of 1024 elements needs ideally and simultaneously 256 processors to be executed in a time equal to 4 × Tex. Of course, this will have also an effect on the flow of communications: IPC = (A.G + B)

We applied data-granularities from 1 to 512 and we looked at the possible differences between the two versions of the heuristic through the range of data-granularities. We also looked what is the optimal value of the data-granularity according to the architecture characteristics we fixed.
These experimentations give us the curves of figure 7 and 8.

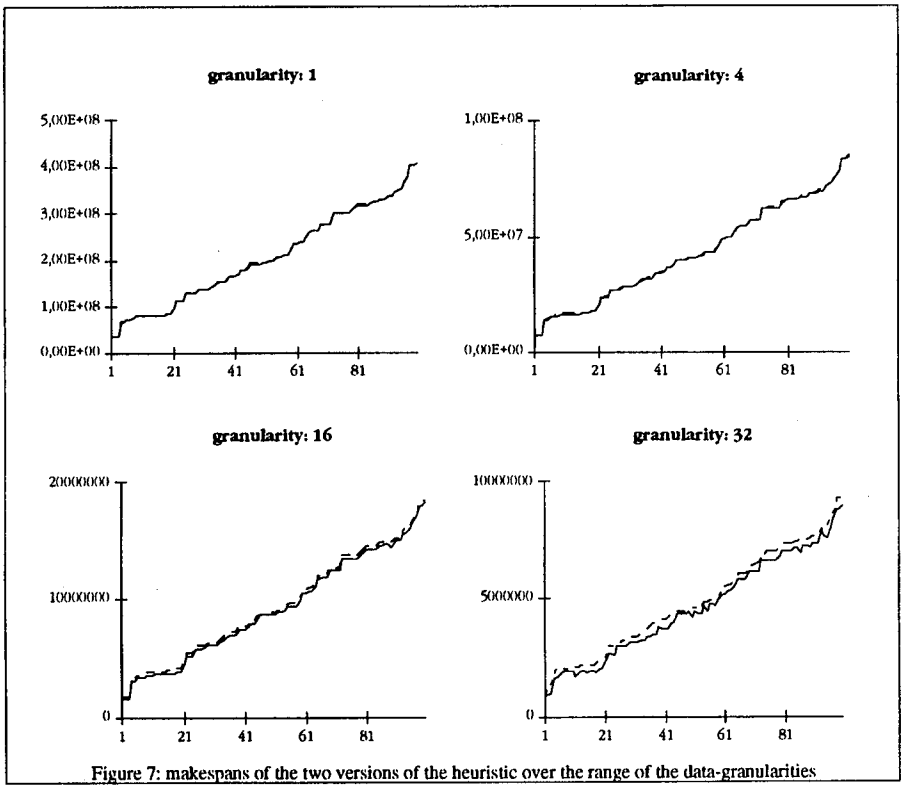


Figure 7: makespans of the two versions of the heuristic over the range of the data-granularities

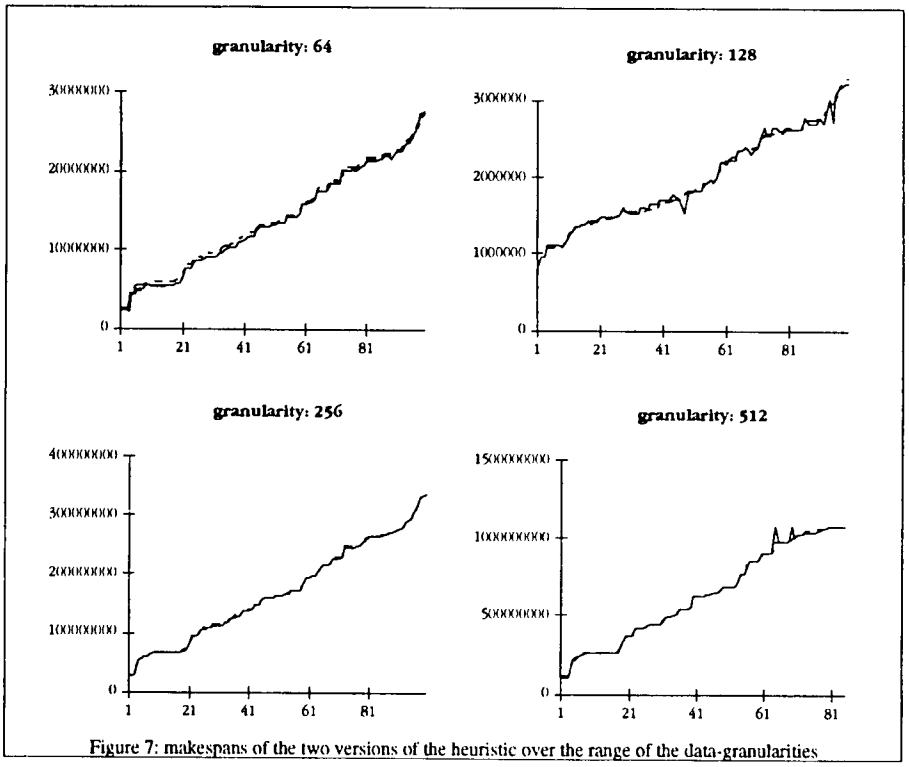


Figure 7: makespans of the two versions of the heuristic over the range of the data-granularities

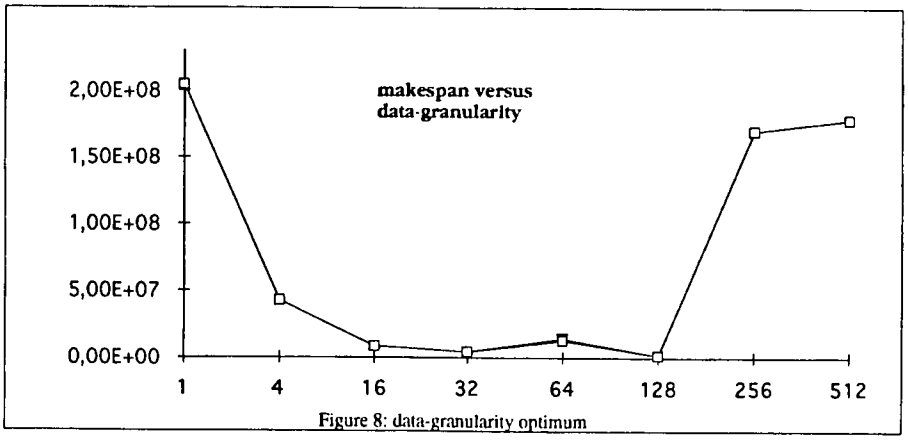


Figure 8: data-granularity optimum

First of all, we see that even there is no marked difference between the two versions of the heuristic, the second one gives in almost cases better makespans (from 0 to 6%). As we said before, the difference

between the two versions is the managing of the list-scheduling. We see that it is better to order the tasks in the rank order and not to fill the list with other tasks until all the tasks of a given rank were mapped. So, the rank criterion is important.
Secondly, we notice an optimum data-granularity (about 120). This value is the reflect of the specific architecture characteristics that we fixed. So for a given architecture, we have to adapt the data-granularity of the data structures we handle before the mapping and the scheduling step.
Of course, we have to validate this point of view with greater graph benchmarks and other architecture characteristics.

7. Conclusion and future experimentations

Our goal is to have a generalised framework for the mapping and the scheduling issues. This leads us to improve the model of program to have more valuable information about the geometries of the objects we handle and the interactions between them. We can also improve the architecture model and add for example a more accurate execution time modelisation to take into account vectorial or superscalar processing.

The improvement of the heuristics comes through adapting the classical technics of scheduling to our specific context and exploiting the enhancement of the programs and architecture models. A work must be also done to evaluate the complexity of our heuristics to try to find a compromise between the level of performances we reach and the complexity of these solutions. As an example, the computation of the mapping and scheduling of a graph in figure 7 takes typically 3mn on a SparcStation10.

8. References

[1] Thinking Machine Corporation, *The Connection Machine CM5, Technical Summary*, october 1991.

[2] F.Capello, J-L Bechennec, J-L Giavitto, PTAH : *Introduction to a new architecture for highly numeric processing*, proc. of PARLE'92, june, Paris, LNCS 605, Springer-Veralg.

[3] J-L Giavitto, *A synchronous dataflow language for massively parallel computers*, proc. of Parallel Computing '91, september 1991, London.

[4] Jing-Jang Hawang, Yuan-Chieh Chow, Frank Angers, Chung-Yee Lee, *Scheduling precedence graphs in systems with interprocessor communication times*, SIAM J. Comp., Vol. 18, N°2, pp 244-257, April 1989.

[5] Gilbert C. Sih, Edward Lee, *Scheduling to account for interprocessor communication within interconnection-constrained processor networks*, 1990 International Conference on Parallel Processing.

[6] Vivek Sarkar, *Partitionning & scheduling parallel programs for multiprocessors*, Ed Pitman.

[7] A.L.Davis, R.M.Keller, *Dataflow program graphs*, Computer, February 1982, pp 26-41.

[8] W.D.Hillis, G.L.Steele, *Data parallel algorithms*, Communication of the ACM, Vol. 29, N° 12, December 1986.

[9] K.K.Parhi, D.G.Messerschmitt, *Static Rate-Optimal Scheduling of Iterative Dataflow Programs via Optimum Unfolding*, IEEE transactions on computers, Vol. 40, N°2, February 1991.

The Enhancement of a User-level Thread Package Scheduling on Multiprocessors

Marisa Gil, Xavier Martorell, Nacho Navarro
Computer Architecture Department
Universitat Politecnica de Catalunya (UPC)
Campus Nord, D6, 08071, Barcelona, Spain
E -mail: {marisa, xavim, nacho}@ac.upc.es

ABSTRACT: Parallel applications on multiprocessors achieve better performance when they run on simpler microkernel scheduling mechanisms with appropriated user level scheduling policies. Our purpose is to offer to application programmers a set of new and simple primitives to get more control over the user-level thread scheduling. This paper presents a new library scheduling approach, based in the CThreads package, running on top of the Mach 3.0 microkernel. To measure and evaluate the proposed primitives, we present experimental results of a multithreaded producer-consumer benchmark.

The proposed scheduling primitives reduce the application execution time from 10 to 20% depending on the number of kernel threads and physical processors.

Keywords: thread packages, user-level scheduling, microkernels, shared memory multiprocessors, parallel applications.

1. Introduction

We present an experimental proposal of a thread package enhancement to support parallel applications running on shared memory multiprocessors. Our main goal is to allow scheduling decisions to be taken at the right place, both at kernel and user level. Applications know how many execution flows may run, which synchronization mechanisms are more suitable and which user-level threads have to run at every moment on the virtual processors given by the kernel. Instead, the operating system manages efficiently physical processors and memory.

The experience has been done on a multiprocessor running the Mach microkernel. This system design model involves new layers in scheduling decisions. The applications have to deal with many entities (kernel, subsystems and libraries). Up to now, the kernel was responsible for thread scheduling. Nowadays there are also libraries that help the user level scheduling. This control, when done independently in both layers, may end in a global low performance. We have exploited and extended the user-level context switch of the CThreads package.

Beside this work, we have also ported and adapted a new event driven scheduling policy of kernel threads based on ESCHED proposals [1][2].

2. User-level parallelism

CThreads is a library provided with Mach which includes primitives to manipulate user-level threads of control and to ease multithreaded programming [3]: primitives to fork and join threads, and primitives for handling mutual exclusion and synchronization based on mutex variables, spin locks and condition variables. All global and static variables are shared among all threads.

The total amount of threads needed by an application is potentially much larger than the number of kernel threads that can be reasonable dedicated to it. The CThreads package allows a pool of threads to run on a specified amount of kernel threads. Some of the services offered are:

- Creation of new cthreads: *cthread_fork (function, argument).*
- Termination of cthreads: *cthread_exit (status).*
- Joining the execution of two cthreads: *cthread_join (cthread).*
- Detaching cthreads to avoid the requirement of joining: *cthread_detach (cthread).*

This work has been supported by the Ministry of Education of Spain (CICYT) under contract TIC 94-439

•To give the cthread virtual processor to another cthread. If no other application cthread can run, do a context switch at kernel level (possibly to another application): *cthread_yield ()*.

•To access to critical regions through the spin locks and mutex locks: *spin_lock(slock)*, *spin_unlock (slock)*, *mutex_lock (mlock)* and *mutex_unlock (mlock)*.

•To synchronize on condition variables: *condition_wait (cond_var, lock)* and *condition_signal (cond_var)*.

•To set the limit of virtual processors (kernel threads) that the application can use. By default, there is no limit, so the kernel maintains a kernel thread for each cthread: *cthread_set_kernel_limit (N)*.

3. Scheduling proposals to the thread package

Kernel threads have been proved to be too expensive to support relatively fine-grain parallelism. User-level thread packages uses procedure calls to provide thread management operations, avoiding the overhead of kernel traps.

In the microkernels arena, the kernel usually relies on message passing mechanisms to block and unblock kernel threads. Some proposals suggest the replacement of this mechanism with a kernel call with less overhead [4].

A handicap for the application execution is the poor information that the current CThreads library returns: there is no knowledge whether a call has succeeded or not and the reason why. And the library often takes flow control decisions transparently to the user. We believe the thread package calls may return more information of what has really been done.

Our purpose is to offer to application programmers a set of new and simple primitives to get more control over the user-level thread scheduling. These new primitives have to work only at user level. The design of our new execution environment has taken care of keeping the semantics the user expects to find at the new tools we have provided. If she/he wants a context switch at user level, and it is not possible, the control is returned to the calling thread to let her/him decide where to continue. We show that this style of solution improves the application execution time.

We have made some modifications to allow CThreads to do user level context switches whenever possible, and consequently to reduce the number of Mach kernel calls. The old implementation relayed on forcing the kernel scheduler to choose between all runnable threads on the target processor. The application is now responsible for choosing the more convenient thread to continue on the same kernel thread. Therefore, the multiplexation of cthreads on top a controlled amount of kernel threads also reduces the large resource allocation needed in the kernel space to manage the task concurrence (by default a kernel thread is instantiated for each user thread).

Although some user-level thread packages ([5], [6]) incorporate scheduling policies, they are mainly designed to work on uniprocessors, thus assuming the presence of only one physical processor [7], or they have other types of restrictions, that reduce their performance, compared with the kernel services.

Moreover, within an application, to limit the number of context switches to the times a thread has to block is the best way to save library overhead time. The application will finish when all its threads terminate, so it seems not to be necessary to add to the user level all the semantics that carries an operating system kernel because the kernel scheduling has to be useful in a general purpose environment, but the user level scheduling deals only with threads of one application.

Other packages use some system services such as the alarm signal in some UNIX implementations, to get a clock interrupt at user level [5]. In this latter case, the application will fail if the programmer attempts to use the alarm signal.

In this work we will show how some of the kernel mechanisms and scheduling policies have been ported to user level to improve application execution times. The library interface attempts to be very general to fit the requirements of efficiency and portability to other systems.

User level scheduling

Microkernels have introduced new scheduling mechanisms. For example, now users can suggest a context switch to the kernel (hint). Then, the kernel applies its scheduling policy and selects another

thread (perhaps in another task) to execute. Users can also directly give their own processor to another kernel thread (handoff) [8]. In this case, the kernel do not apply its scheduling policy and transfers the physical processor to that thread, if it is not blocked.

As an experience, we have implemented a schedule hint and handoff mechanism into CThreads. The new primitive: *cthread_handoff(thread)* tries to give the current virtual processor -the kernel thread- to the target cthread, without caring for any other scheduling policy. There is a new call *cthread_hint()* which searches another runnable cthread according to the current scheduling policy; if there are no candidates, it simply returns control to the caller. This contrasts with the behaviour of *cthread_yield()*, which usually calls the kernel, relying on the kernel scheduling decisions.

In order to introduce user-level priorities in the CThreads library, we have added three new primitives: *cthread_set_prio()*, *cthread_get_prio()* and *cthread_init_prio()*. Applications can set the priority of their cthreads using the *cthread_set_prio(cthread, prio)* library call. All cthreads with the maximum priority are scheduled in FIFO order. Cthreads with less priority are not scheduled if there are enough higher priority cthreads to fill all application virtual processors. After a *cthread_fork(function, arg)* call, new cthreads begin execution at the default priority level.

Originally, the queue manipulation routines of the CThreads library were designed to maintain a FIFO order. We have modified such routines to insert items in queues in the right place according to their priority. Within a priority level, these routines have the traditional FIFO behaviour.

Library priorities are static, that is, the library does not attempt to change them. The priority system purpose is to have some levels of execution. In this way, applications can assign higher priorities to such cthreads that execute the more important work. Priorities can be used to control the execution order of the application cthreads, modifying them and suggesting a context switch (with *cthread_hint ()*, for example).

We have provided a user-level preemption mechanism into the CThreads package, in order to be able to preempt the virtual processor and give it to another one. This mechanism, presented in section 4, allows us to supply a scheduler thread at user level that implements preemptable policies as round robin.

Optimizations to the synchronization mechanisms

It is possible to use the priority system in an attempt to spent less time in getting a spin lock and to reduce the time a cthread is in a critical section. This can be done lowering the priority of cthreads getting an spin lock and raising the priority of cthreads executing in a critical section.

In the standard library, if a critical section is protected with a spin lock, cthreads that find the section locked will be testing the lock during a period of time. The *spin_lock()* routine can decrease the cthread priority to the minimum value and can suggest a user context switch (*cthread_hint()*). It is very important to note that, in this case, *the application must assure* that the cthread in the spin lock is going to receive a virtual processor when the critical section will be freed. This is possible recording the cthreads waiting in a spin lock. When the cthread in the critical section leaves it (with *spin_unlock()*), the application can also decide to give its virtual processor to a waiting cthread, raising its priority to the original value (with *cthread_set_prio()* and *cthread_handoff()*).

Also working with spin locks, it is possible to modify the *spin_lock()* routine to test for the lock during a fixed number of iterations only, and, if the lock is not freed while waiting, suggest a user level context switch with *cthread_hint()*.

On the other hand, when a cthread acquires a critical section with *mutex_lock()*, it is possible to raise its user priority to the maximum value. In this way, the cthread will execute at maximum priority until it releases the lock (using *mutex_unlock()*). As the library is not preemptive, the time in the critical section will be reduced only in case the cthread blocks while is in the section, because when it wakes up, it will receive a virtual processor at the first scheduling time.

Moreover, it is possible to modify the *mutex_unlock()* routine to suggest a *cthread_handoff()* to the first thread waiting in the mutex queue. When a cthread exits from a critical section, it gives its virtual processor to another thread that wants to enter the critical section.

These optimizations are available through compilation options in the modified CThreads library, so for every application it is possible to select the library that fits the application require-

ments.

Some of these implementations are already tested in some systems at kernel level (smart scheduler in Symunix [9]).

4. Environment and performance evaluation

We have taken a multithreaded server as the archetype of applications we want to give support to, that is, those tasks whose goal is to give some service to operating system users [10].

Besides, we have implemented a CPU server, a facility to allocate processors to tasks. It is needed if we want to support a variety of programming models running on a multiprocessor. Fine grain parallel applications need to know how many processors are currently available and allocate them accordingly.

A shared memory event collection mechanism developed at GMD (JEWEL) has let us to take measures of the server process running on Mach. We have compared the standard CThreads library with the modified library.

The current implementation of the benchmark consists on some number of producer threads and some number of consumer threads. As an example, producer threads generate matrices and consumer threads calculate the matrix product. The coordination of duties use the new scheduling primitives.

Mutex and condition variables are used to ensure mutual exclusion to get shared data, following this scheme:

```
type_t                  data;
mutex_t                 data_lock;
condition_t             data_cond;
```

To get access to a data item:

```
mutex_lock (data_lock);
while (!freeData ())
  condition_wait (data_cond, data_lock);
m = getData ();
mutex_unlock (data_lock);
```

To release a data item:

```
mutex_lock (data_lock);
release_data ();
condition_signal (data_cond);
mutex_unlock (data_lock);
```

The producer and consumer synchronization points follow this pattern:

Producer

```
WHILE (TRUE) {
// get data item
// generate data

  // Get mutex to select a consumer
mutex_lock(consumers_lock);
while (consumers_busy())
  condition_wait(consumers_available,
                 consumers_lock);
c = fetch_consumer();
c->free = FALSE;
  // Release mutex
mutex_unlock(consumers_lock);


  // Wakeup consumer
mutex_unlock(c->waiting_for_work);

// Optionally, give virtual processor
// to consumer thread
// (see options below)
}
```

Consumer

```
WHILE (TRUE) {


  // Get mutex to become available
mutex_lock(consumers_lock);
myself->free = TRUE;
  // Wakeup any waiting producer
condition_signal(consumers_available);
  // Release mutex
mutex_unlock(consumers_lock);


  // Block while no work to do
mutex_lock (myself->waiting_for_work);

// consume data
// release data item
}
```

Figure 1: Producer and consumer synchronization

In some cases, when we produce some data and we know which thread is going to use it, we can give control to this thread using the primitives added to the CThreads library: *cthread_handoff()* and *cthread_hint()*. Using them, if it is possible, the library performs a user level context switch

instead of leaving that decision to the kernel. For example, when a producer thread has generated some data to be processed, it gets a free consumer thread and gives control to it (Figure 1).

As the producer knows which consumer thread is waiting for the generated data, we will compare four possibilities to give the virtual processor to the consumer (see Figure 1). *Mutex_unlock()* puts de consumer thread in the ready user level queue. Then, the producer can follow one of these constructions:

- With the standard CThreads library:

•NOP: the first option is do nothing. Usually, users will get this option. In this case, scheduling will occur later, when the kernel selects the consumer or when the producer thread blocks.

•YIELD: the second option is to execute a *cthread_yield()*, which attempts a user level context switch. If it fails, calls the kernel to let it choose another thread.

- With the modified library:

•HANDOFF: the first option is to call *cthread_handoff()*, that attempts to give the producer virtual processor to the consumer. If it fails, it returns control to the producer.

•HANDOFF+HINT: the last option first also tries a *cthread_handoff()* and, if it fails, attempts a *cthread_hint()*, which gives the virtual processor to any user level ready thread. If this also fails, it returns control to the original thread.

The library routine *cthread_handoff()* misses and cannot perform a context switch when the target cthread is already attached to a virtual processor (kernel thread). This happens when the number of kernel threads is not adequately limited. If the programmer goal is to have the maximum number of user level context switches, he has to determine the number of kernel threads that matches the inherent parallelism of his application.

The *cthread_hint()* function fails when there are no user-level ready cthreads.

In order to isolate the application from external interferences, with the assistance of the CPU server, a machine partition with dedicated processors is allocated during the execution of the application. Kernel context switches are then confined within the application virtual processors.

We have run the example of the producer-consumer problem, with four threads of each class, varying the physical parallelism (processors) and the number of virtual processors (kernel threads), for each of the four policies trying to hand over the execution context at user level explained above. The machine is a DEC433MP with four i486 processors running Mach 3.0 MK4.1 (OSF/1 1.1) microkernel.

Figures 2 to 4 show the benchmark execution time depending on the number of kernel threads and physical processors. Although four processors is a narrow experience, we though the size of the chosen problem is adequate and the consequences are scalable with larger architectures and larger parallel applications. Figure 5 presents the number of handoff misses in those experiments.

Comments to the experiences

Uniprocessor scenario

When the application runs on a dedicated physical processor with one to four virtual processors, its behaviour is nearly constant (Figure 2). Each producer thread attempts to give their virtual processor to the selected consumer, and it almost always hits (Figure 5). The kernel is unaware of this changes at user level. In this case, the HANDOFF execution time is better than the NOP or YIELD options (about 10 - 20%).

On the other hand, when the number of kernel threads exceeds the number of producers (4), there are some consumer threads with their own virtual processor. If a producer attempts to give its kernel thread to them, it misses (Figure 5). Moreover, kernel scheduling overhead increases and affects the intended user level scheduling. In fact, looking at the specific execution time results, values differ a lot between different executions. Unpaired attempts of give control produces unpredictable effects. The YIELD option produces more kernel context switches and has unpredictable paths.

With the NOP option, the producer continues until blocks. The other policies are worse at this point because they spend user time at handoff misses.

The scheduling discipline is very sensitive on a uniprocessor. When the number of processors

increases, the policies are much less important. This scenario demonstrates that even with no physical parallelism, the constraint of the number of kernel threads is a good practice.

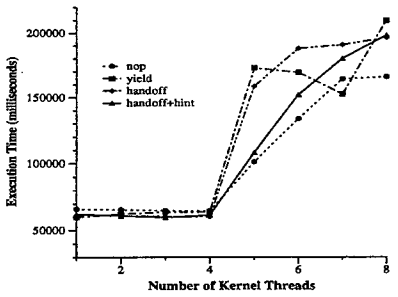


Figure 2: Execution time on 1 physical processor, varying kernel threads and hand over policies

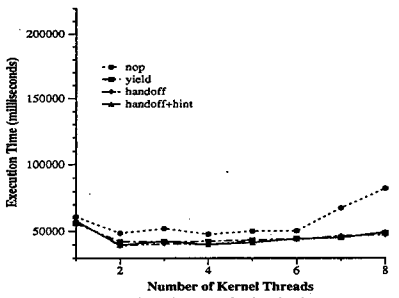


Figure 3: Execution time on 2 physical processors varying kernel threads and hand over policies

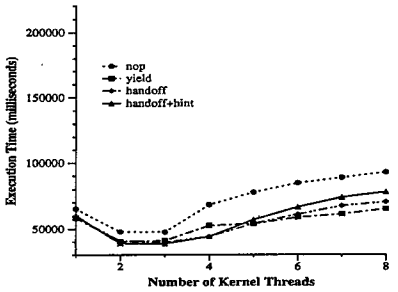


Figure 4: Execution time on 4 physical processors varying kernel threads and hand over policies

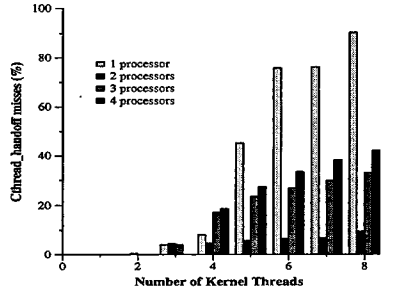


Figure 5: Thread handoff misses on different physical and kernel parallelisms

Two processors scenario

In this situation we have two dedicated processors. The application has enough threads (8) to distribute correctly between the kernel threads. That is the reason why a great number of *cthread_handoff()* hit and hand over the virtual processor from the producer to the consumer.

The producer wakes up the consumer (*mutex_unlock()*) and, before the kernel can dispatch a kernel thread to run it on, the producer give to it its virtual processor.

This is the best case for all the options; we have found the right granularity to this experiment that match the maximum hits of the handoff mechanism (Figure 5).

Four processors scenario

When the physical parallelism increases, consumers complete easier and quickly. Producers have to wait less time to get a free consumer. Producers do not stop for resource shortage (less buffers are needed).

But, in the other hand, the kernel is free to assign a kernel thread to a consumer when a producer wakes it up. So, although the producer knows that the consumer is available and tries to give its virtual processor to it, it founds that 16% times (Figure 4, 4 kernel threads) is already running because, between the *mutex_unlock()* and handoff the kernel has already dispatched it.

When the application has more than four kernel threads, some consumers have a virtual processor associated. They seem to be running at user level, then handoff misses but the kernel may have not selected them for running. In this case, the application suffers more context switches than expected. The application cannot achieve its maximum speed.

Although been an heuristic remark, the benchmark runs better with two physical processors because it has reached the right machine power for this application. Add more processors, and more movement will appear in the kernel side, which conflicts against the user scheduling. User time increases a few due to user level mutex synchronization -more often when there are more processors-, but kernel time increases a lot (up to 150%).

5. User-level preemption and context switch

We notice that user-level packages usually do not allow preemption at user level or, if implemented, it is based on the UNIX signals mechanism. That ties the library to the UNIX environment and also prevents the application of using this mechanism for its own purpose.

We have enhanced our library to support timer expiration upcalls, in a manner similar to Scheduler Activations [11] and First-Class Threads [12] do. A new kernel call sets a one shot or at fixed intervals notification to the user which has to provide a routine and a stack to manage the software interrupt. The kernel brings the state (registers) of the preempted thread as a parameter to the upcall routine. Then, the library is in charge of doing a user-level context switch to the appropriate cthread, allowing round-robin, handoff or whatever scheduling and timeout handling at application level. To prevent deadlocks, we inhibit new interrupts during the upcall handling, quickly save the preempted state, and then handoff to a user-level scheduler thread that, as a average cthread, chooses which to continue.

Table 1 presents the execution times spent in the upcall service routine and, for the user scheduler, to select a thread and perform a context switch. Those times include the management of the CThreads ready queue and the context switch.

Action	Time (µs)
Upcall timer handler	36.22
User-level scheduler	51.06

Table 1: Execution times for the upcall timer handler and select a new thread

The application, besides the ability to decide its own internal scheduling, can also benefit from a cut down in scheduling overhead if it is done in less time at user level. Table 2 shows that there is an order of magnitude difference in cost between user level and kernel level thread context switch.

Context switch routine	Time (µs)
cthread_handoff()	16.29
cthread_hint()	16.26
cthread_yield()	13.90
thread_switch() [kernel call]	134.86

Table 2:Execution times for user-level (handoff, hint, yield) and kernel-level context switch.

The primitives we have added to the package spend a little more time because they select new cthreads by priority, instead of *cthread_yield()* that works FIFO. The thread_switch() measured performs a kernel context switch between two kernel threads of the same task.

The best the application can do is to limit the number of kernel threads to the amount of physical processors it has allocated, set a timer preemption event, and carry out its own flow scheduling.

6. Conclusions and lessons learned

Simpler kernel scheduling mechanisms, if associated with appropriated user level scheduling policies, permit us to obtain better performance. This is due to the fact that the application knows better than the system the behaviour of its own threads, and because the user mechanisms are cheaper.

In the overall executions, when the application creates more user level threads than kernel threads, the modified library executes more efficiently than the standard CThreads package. The programmer knowledge, or implemented implicitly into the library functionalities, is able to foresee and force a successful continuation. The interference of the kernel is reduced, due to the hits of user-level context switch at synchronization points.

If the number of virtual processors exceeds the adequate to the application granularity (the number of instructions between two synchronization points in the sequential execution of a control flow), although this number is till now a heuristic, the fair kernel scheduling do not suit the application needs.

In the other hand, when parallel algorithms can adapt its execution flows to the number of processors assigned, the kernel should not interfere making other scheduling decisions and should inform the application about any changes in resources allocation.

In the parallel applications support field, the microkernel might manage the allocation of processors to tasks but might not schedule threads at a quantum expiration or priority recalculation rate. Kernel should notify the user of system events that may affect the job. This allows the thread package to respond to the event in the most appropriate manner for the application. We are working in this trend, following the scheduler activations mechanism proposal [11].

7. References

[1] "UNIX scheduling for large systems"
J.H. Straathof, A.K. Thareja, A.K. Agrawala
Proceedings of the Denver USENIX Conference, January 1986

[2] "Towards User-Level Parallelism with Minimal Kernel Support on Mach"
Marisa Gil, Toni Cortes, Angel Toribio and Nacho Navarro
OSF Research Workshop, 22-24 June 1993, Grenoble (France)

[3] "C Threads"
E.C. Cooper and Richard P. Draves
Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[4] "Using Continuations to Build a User-Level Threads Library"
Randall W. Dean
School of Computer Science, CMU, March 1993

[5] "Threads, A System for the Support of Concurrent Programming"
Thomas W. Doeppner Jr.
Brown University, Technical Report CS-87-11, June 1987

[6] "Threads Extension for Portable Operating Systems (Draft 6)"
IEEE, P1003.4a/D6, February 1992

[7] "A Library Implementation of POSIX Threads under UNIX"
Frank Mueller
USENIX, Winter'93, San Diego, CA

[8] "Processors, Priority, and Policy: Mach Scheduling for New Environments"
David L. Black
USENIX Winter'91, Dallas TX

[9] "Process Management for Highly Parallel UNIX Systems"

Jan Edler, Jim Lipkis, and Edith Schonberg,
NYU Ultracomputer Research Laboratory
Workshop on UNIX and Supercomputers, USENIX, September, 1988

[10] "Mach 3 Server Writer's Guide"
Open Software Foundation and Carnegie Mellon University
Keith Loepere Editor, January 1992

[11] "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism"
Thommas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy
Proceedings of the 13th ACM Symposium on Operating Systems Principles, October 1991

[12] "First-Class User-Level Threads"
Brian D. Marsh et al.
ACM OSR, Vol.25 Num.5, October 1991.

USE OF THE V-RAY SYSTEM FOR OPTIMIZATION OF SERIAL PROGRAMS FOR CRAY SUPERCOMPUTERS

Dr Vladimir V.Voevodin,
Research Computing Center, Moscow State University,
Lenin Hills, Moscow, 119899, Russia
E-mail: voevodin@vvv.srcc.msu.su

Abstract

Although the problem of optimizing programs to parallel computers exists for more than twenty years its satisfactory solution is not found so far. The paper considers applications of the advanced mathematical V-Ray technology to analysis and transformations of serial programs on vector/parallel supercomputers. As a numerical example we consider optimization of the TRFD Perfect Benchmark for CRAY Y-MP M90 and C90 computers.

Keywords: Vector/parallel supercomputers, optimization of serial programs, the Perfect Club Benchmarks, the V-Ray mathematical technology.

Introduction

Several different approaches were suggested, however, none of them is able now to cover the whole scope of problems arising in optimization of industrial applications to CRAY Y-MP computers. In this paper we consider a principally new strictly mathematical approach based on the so called V-Ray technology. This technology incorporates recent advanced results from the graph theory and the theory of static analysis of programs and allows one to perform an exhaustive analysis of all required properties of programs starting from the commonly adopted data dependency and the control flow analysis up to the optimization of data distribution and data locality.

It should be emphasized that the V-Ray technology provides a mathematical guarantee of detection of all required properties (for example, the whole resource of parallelism) and ensures that all suggested transformations are strictly equivalent (even taking into account round-off errors). As a numerical example we consider in the paper application of the V-Ray technology for optimizing the TRFD Perfect Benchmark to CRAY Y-MP M90 and C90 computers.

The Perfect Club Benchmarks and the TRFD Code

The Perfect Club suite has been used for many years for evaluating of real performance of parallel computers when solving industrial applications. By now there are published a lot of papers, e.g. [1,2], describing its general organization, structure of each code, key features of some important subroutines, performances for a wide range of computers and etc. However, just that high popularity of the suite seemed attractive to us: it is much more interesting to produce a nontrivial result in the field where many researchers have worked very hard and, moreover, this is the only way to show to industrial users potentials of the V-Ray technology as compared with existing approaches to analysis and restructuring of codes. To this end we have chosen the TRFD Perfect Benchmark.

The original version of the code delivers the near 56 Mflop/s (baseline) performance on the CRAY Y-MP computer independently of the actual number of processors available (we remind that the peak performance of the CRAY Y-MP computer equals 333 Mflop/s per processor). The best hand optimization of the code enables one to speedup execution up to 82 Mflop/s for one Y-MP processor ("baseline" as well as "hand" optimization performance data are provided by Jeff Brooks from CRAY Research, Inc., Eagan MN, USA. It is clear that the original TRFD code is not only poorly vectorized, but even hand transformations do not lead to any considerable improvement of the performance. Obviously two questions arise immediately:

- What is the reason of such poor performance?
- Is it possible at all to improve performance of this benchmark using only equivalent transformations?

Running a little ahead we give the answer right now: yes, it is quite possible! In particular, use of the V-Ray technology enables us to improve performance of this code for one processor of CRAY Y-MP computer up to 247 Mflop/s.

The V-Ray Technology

The V-Ray technology involves solutions of several difficult problems, i.e. data dependency detection, analysis of data locality, extraction of superfluous assignments but it is mostly oriented to providing a strong mathematical basis for resolving the whole set of problems related to mapping industrial applications onto parallel and massively parallel computers.

A notion of algorithm graph holds the central place in the V-Ray technology. For the sake of brevity we do not present here a description of the V-Ray technology itself (the reader can easily find all required information in [3,4]). We would like to remind only three basic principles underlying the algorithm graph construction: description of each instance (execution) of each statement, indication of exact data transfer between instances (nodes of the graph) and compact description of the entire algorithm graph in an explicit form.

When mapping a program on a parallel computer we pay a special attention to the three main topics:

- determination of full structure of the program (in the broad sense);
- defining key features of the target computer architecture;
- transformation of the program using a simultaneous analysis of the program structure and the computer architecture.

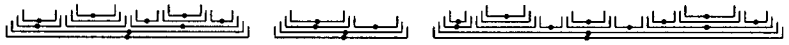
It is rather evident that a particular set of problems which one has to solve is determined by properties of both the code under analysis and the target computer architecture. And in this paper we describe briefly only whose components of the V-Ray technology which were essentially exploited when optimizing the TRFD code (a general discussion of advantages and disadvantages of the V-Ray technology is a subject of an independent big paper).

V–Ray Components for Optimizing the TRFD Benchmark

Reconstruction of multidimensional arrays according to their compact one dimensional form. We mentioned already that the serious difficulty when analyzing the TRFD code is related to use of the compact one dimensional form of the main multidimensional array. To overcome this difficulty we used the technique for reconstructing multidimentional arrays according to their compact one dimensional form. Although sometimes this problem may possess an arbitrary number of solutions we can take any of them which converts index expressions to a linear form. The new form of the arrays may not coincide with their original form before the memory optimization and this is not necessary since any acceptable form can clarify structure of the code. Note that we do not discard the memory optimization problem at all but postpone it until after the "parallel optimization" will be completed. Thus, the structure of OLDA can be described in terms of the linear model of the V–Ray technology.

Determination of the exact informational structure of a program. Conversion of the program to an analyzable form enables us to determine all necessary properties, i.e. existence or absence of dependency between statements and of separate parts of the program, locality of data references, possibility for array privatization amongst others (this is the most important part of the optimization process and it will be briefly described below).

Detection and description of the total potential resource of parallelism. This is a straightforward corollary: since we know the exact informational structure of the code we can easily extract the whole resource of parallelism. At present the V–Ray software system supports detection of parallel operations on several levels: independence of statements, independence of separate arbitrary large parts of the routine, "coordinate" and "skew" parallelisms within loops. In particular, the total resource of coordinate parallelism of the subroutine OLDA is shown on its loop profile below (loops with independent iterations are marked by the dots in the middle of brackets).



The marked loop profile of the OLDA subroutine.

Note that at least one important conclusion immediately follows from this picture: if one relies only on the structure of the innermost and outermost loops of the routine the considerable part of its resource will remain hidden. Using this marked profile (as well as the internal description of the detected properties) we can easily realize a substantial reserve for the further improvement of the performance which is involved in the original code.

Finding a set of admissible elementary transformations of loops. We have just found out that OLDA possesses the large reserve of the coordinate parallelism. How to utilize it? From the one hand it is clear that in this case one can use only simple "coordinate" modifications of the loops, but from the another hand it is necessary first to know the set of admissible transformations among others. To this end the V–Ray system supports a wide range of conventional loop transformations and the following ones were used when optimizing OLDA: loop collapse, loop interchange, loop distribution, loop

fusion and peeling. These modifications were exploited to increase vector length, to perform vectorization along the leading dimension of arrays, to remove dependency from loops and superfluous barriers of synchronization for parallel processing.

Analysis of data locality. This item is of great importance for the efficient execution of programs on any kind of parallel computers although very often it is not taken into account for conventional CRAY vector/parallel supercomputers. In practical use this kind of optimization means a reduction of data transfers between the main memory and vector registers. In terms of the V-Ray technology it requires an analysis of data locality and expediency of temporary use of arrays. In this particular TRFD benchmark this optimization yielded a significant improvement of performance. The most important step of the analysis of data locality, namely determination of points, where ones and the same variables were repeatedly used was done by the V-Ray system automatically.

The Final Results of Optimization of the TRFD Code

We performed a series of numerical experiments with the TRFD benchmark on CRAY Y-MP M90 and C90 computers and the final results are shown in Table 1 which adopts the following notation:

- 'Baseline perf.': corresponds to the original (nontransformed) code;
- 'Hand opt.': hand optimization was performed;
- 'V-Ray opt.': performance after optimization of the TRFD code using the V-Ray technology.

Tab. 1.

Y-MP CPUs	Baseline perf. Mflop/s	Hand opt. Mflop/s	V-Ray opt. Mflop/s
1	56.19*	81.72†	247†
4	54.86	261.64	822
8	54.34	481.03	954+‡
C90 CPUs	**Baseline perf. Mflop/s**	**Hand opt. Mflop/s**	**V-Ray opt. Mflop/s**
1	89.5*	139.71†	579.7†
8	89.6	962.68	2440.5‡

The final results of optimization of the TRFD code.

After analyzing the data from table 1 the following observations can be made:
(+) the value was measured during a nondedicated use of the computers and can be somewhat improved;
(*) the values take the seventh place among baseline performance values of the all thirteen Perfect Benchmarks;

(!) the values take the ninth place among 'Hand opt.' performance values of the all thirteen Perfect Benchmarks;
(†) the values take the first place among 'Hand opt.' performance values of the all thirteen Perfect Benchmarks;
(‡) the values take the second place among 'Hand opt.' performance values of the all thirteen Perfect Benchmarks losing the fist one only to the highly parallel MG3D code.
We would like to especially note that all TRFD optimizations were performed using equivalent Fortran transformations with no CAL fragments or straightforward calls of CAL coded library routines.

Conclusions

The process of optimization of programs for parallel computers involves many difficult problems related to the structure of program to be analyzed as well as to the properties of the target computer architecture. To resolve them successfully we use the V–Ray technology that not only gives keys to their solutions but also makes it possible to understand whether any further improvement of the performance is feasible. We have shown the potential of the V–Ray technology using only one TRFD Perfect Benchmark and we hope to demonstrate soon its capabilities when optimizing codes to the massively parallel T3D computer.

References

[1] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, ***Supercomputer performance evaluation and the Perfect Benchmarks***, Tech. Rep. 965, CSRD, Univ. of Illinois at Urbana, Technical Report, 1990.
[2] C.M. Grassl ***Parallel Performance of Applications on Supercomputers***, Parallel Computing v.17 (1991), pp.1257-1273.
[3] V.V. Voevodin ***Mathematical foundations of parallel computing***, Computer Science Series, v.33, World Sci. Publ. Co., Singapore, 1992.
[4] Vl.V. Voevodin ***Theory and Practice of Parallelism Detection in Sequential Programs***, Programming and computer software, v.18 (1992), n.3.

Section VII : Numerical Algorithms

Analysis of splitting parallel methods for solving block tridiagonal linear systems

E. Galligani and V. Ruggiero
Department of Mathematics, University of Modena,
Via Campi 213, I-41100 Modena, Italy

Summary. This paper is concerned with the solution of block tridiagonal linear algebraic systems by two different parallel methods, the Arithmetic Mean method and the Alternating Group Explicit method. Similar convergence conditions hold for both methods. We describe the parallel implementation of both methods on shared and distributed memory systems and we report the results of numerical experiments on a set of test problems.
Keywords. Block tridiagonal systems, Parallel methods, Arithmetic Mean method, Alternating Group Explicit method.

1 Introduction

We consider the system of n linear equations

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

where the non singular matrix A has the form

$$A = \begin{pmatrix} B_1 & C_1 & & & & \\ D_2 & B_2 & C_2 & & & \\ & D_3 & B_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & D_{q-1} & B_{q-1} & C_{q-1} \\ & & & & D_q & B_q \end{pmatrix} \tag{2}$$

Each square block B_i is a non singular $p \times p$ matrix and the blocks C_i and D_i are square matrices of order p ($i = 1, ..., q$; $D_1 = 0$, $C_q = 0$); thus $n = q \cdot p$. We assume that q is even.

Block systems like these arise for example in the numerical solution of partial differential equations. The simplest and the most well-known examples are block-tridiagonal systems that occur using the finite difference method with the 5-point scheme for the diffusion-convection equation. Other examples occur applying finite element methods to partial differential equations.

There are different ways to parallelize numerical methods for such systems. A standard approach is to use *domain decomposition* techniques combined with finite difference or

finite element methods. They partition the domain, where the partial differential equation has to be solved, into subdomains which are uncoupled and then couple the boundaries of these subdomains. Thus, the discretized version gives uncoupled linear systems, which can be solved in parallel, and a system of coupling equations that combines the local solutions to compute the complete solution [1].

In this note we take a different approach based on *splitting and alternating direction* methods.

2 Splitting and alternating direction methods

An iterative method for solving (1) was introduced by Evans ([2], [3]) and is known as the *Alternating Group Explicit* (AGE) method analogous to the well known *Alternating Direction Implicit* (ADI) method. We consider the following splitting of the matrix (2):

$$A = G_1 + G_2 \tag{3}$$

where G_1 and G_2 are the matrices

$$G_1 = diag\{P'_1, P'_3, ..., P'_{q-1}\} \qquad G_2 = diag\{B'_1, P'_2, ..., P'_{q-2}, B'_q\} \tag{4}$$

with

$$P'_i = \begin{bmatrix} B'_i & C_i \\ D_{i+1} & B'_{i+1} \end{bmatrix} \quad i = 1, ..., q-1, \qquad P'_q = \begin{bmatrix} B'_q & 0 \\ 0 & B'_1 \end{bmatrix} \tag{5}$$

and $B'_i = \frac{1}{2}B_i$ $(i = 1, ..., q)$. An analogous splitting of A is considered when q is odd. For any real positive number σ for which $G_1 + \sigma I$ and $G_2 + \sigma I$ are non singular matrices, we define the following iterative method $(k = 0, 1, ...)$:

$$\begin{cases} (G_1 + \sigma I)\boldsymbol{y}^{(k+\frac{1}{2})} &= \boldsymbol{b} - (G_2 - \sigma I)\boldsymbol{y}^{(k)} \\ (G_2 + \sigma I)\boldsymbol{y}^{(k+1)} &= \boldsymbol{b} - (G_1 - \sigma I)\boldsymbol{y}^{(k+\frac{1}{2})} \end{cases} \tag{6}$$

where $\boldsymbol{y}^{(0)}$ is an arbitrary initial vector approximation of the unique solution $\boldsymbol{x}^*$ of (1). We combine the above two equations into the form

$$\boldsymbol{y}^{(k+1)} = S\boldsymbol{y}^{(k)} + \boldsymbol{d} \tag{7}$$

where

$$\begin{aligned} \boldsymbol{d} &= (G_2 + \sigma I)^{-1}(I - (G_1 - \sigma I)(G_1 + \sigma I)^{-1})\boldsymbol{b} \\ S &= (G_2 + \sigma I)^{-1}(\sigma I - G_1)(G_1 + \sigma I)^{-1}(\sigma I - G_2) \end{aligned} \tag{8}$$

The matrix S is called the AGE matrix. In [3] a condition for the convergence of (7) has been given. Indeed, the following theorem has been proved. (As usual, the iterative method (7) is convergent to the solution $\boldsymbol{x}^*$ of (1) if and only if $\rho(S) < 1$, where $\rho(S)$ denotes the spectral radius of S).

Theorem 1. If G_1 and G_2 are symmetric positive definite matrices and if $\sigma > 0$, then $\rho(S) < 1$. Besides, if the eigenvalues λ of G_1 and μ of G_2 lie in the ranges

$$0 < a \leq \lambda \leq b, \qquad 0 < a \leq \mu \leq b$$

a *good* choice of σ is given by $\sigma = \sqrt{a \cdot b}$.

In this paper we prove a new convergence theorem for the AGE method (6) when A is an M-matrix.

Theorem 2. Let $A \equiv (a_{ij})$ be an $n \times n$ irreducibly diagonally dominant (or strictly diagonally dominant) real matrix with $a_{ij} \leq 0$ for $i \neq j$ and $a_{ii} > 0$, $i = 1, ..., n$. Then, for

$$\sigma > \max_{1 \leq i \leq n} \frac{a_{ii}}{2} \qquad (\sigma \geq \max_{1 \leq i \leq n} \frac{a_{ii}}{2}) \tag{9}$$

the AGE matrix S of formula (8) is convergent.

Proof. We define $E_t = G_t + \sigma I$ and $F_t = \sigma I - G_t$ for $t = 1, 2$. For the condition (9) on σ, the matrix E_t $(t = 1, 2)$ is a strictly diagonally dominant matrix with positive entries on the diagonal and with non positive off-diagonal elements. Thus, E_t is a non singular M-matrix: $E_t^{-1} \geq 0$ [4, Theor. 3.4 and Theor. 3.10]. For the condition (9) on σ, the matrices F_1 and F_2 are non negative: $F_t \geq 0$. Thus, the matrix $S = E_2^{-1} F_1 E_1^{-1} F_2$ is non negative and $S^m \geq 0$ for $m = 0, 1, ...$ Since $E_1 - F_2 = G_1 + \sigma I - \sigma I + G_2 = A$ and $E_2 - F_1 = G_2 + \sigma I - \sigma I + G_1 = A$, we can write

$$\begin{aligned} S &= E_2^{-1}(E_2 - A)E_1^{-1}(E_1 - A) = (I - E_2^{-1}A)(I - E_1^{-1}A) \\ &= I - E_2^{-1}(E_1 + E_2 - A)E_1^{-1}A = I - 2\sigma E_2^{-1} E_1^{-1} A. \end{aligned}$$

We define $Z = 2\sigma E_2^{-1} E_1^{-1}$; then $Z = (I - S)A^{-1}$.

Since the matrix A is an irreducibly (strictly) diagonally dominant matrix with positive entries on the diagonal and with non positive off-diagonal elements, the matrix A^{-1} is positive (non negative) [4, Theor. 3.11]. Since $Z \geq 0$ and $A^{-1} > 0$ $(A^{-1} \geq 0)$, we have the result

$$\begin{aligned} 0 &\leq (I + S + S^2 + ... + S^m)Z \\ &= (I + S + S^2 + ... + S^m)(I - S)A^{-1} = (I - S^{m+1})A^{-1} \leq A^{-1}. \end{aligned}$$

Now, the proof runs parallel to a standard proof given in [5, p. 119].□

Under the same hypothesis on the matrix A, the above theorem holds also when the vector $\boldsymbol{d}$ and the matrix S in (8) have the form

$$\begin{aligned} \boldsymbol{d} &= (G_2 + \Sigma)^{-1}(I - (G_1 - \Sigma)(G_1 + \Sigma)^{-1})\boldsymbol{b} \\ S &= (G_2 + \Sigma)^{-1}(\Sigma - G_1)(G_1 + \Sigma)^{-1}(\Sigma - G_2) \end{aligned} \tag{10}$$

where Σ is an $n \times n$ diagonal matrix with diagonal entries $\sigma_i > \frac{a_{ii}}{2}$ (if A is irreducibly diagonally dominant) or $\sigma_i \geq \frac{a_{ii}}{2}$ (if A is strictly diagonally dominant). (The proof runs in the same way with $E_t = G_t + \Sigma$, $F_t = \Sigma - G_t$ for $t = 1, 2$ and $Z = 2E_2^{-1}\Sigma E_1^{-1}$).

In [6] another splitting of matrix (2) has been considered:

$$A = H_1 + K_1 \qquad A = H_2 + K_2 \tag{11}$$

where H_1, H_2, K_1 and K_2 are the following matrices

$$\begin{array}{rclrcl} H_1 &=& diag\{P_1, P_3, ..., P_{q-1}\}, & K_1 &=& A - H_1 \\ H_2 &=& diag\{B_1, P_2, ..., P_{q-2}, B_q\}, & K_2 &=& A - H_2 \end{array} \tag{12}$$

with

$$P_i = \begin{bmatrix} B_i & C_i \\ D_{i+1} & B_{i+1} \end{bmatrix} \quad i = 1, ..., q-1, \qquad P_q = \begin{bmatrix} B_q & 0 \\ 0 & B_1 \end{bmatrix}. \tag{13}$$

Then, for any real non negative number ρ for which $H_1 + \rho I$ and $H_2 + \rho I$ are non singular matrices, we define the following *Arithmetic Mean* (AM) method:

$$\begin{cases} (H_1 + \rho I)\tilde{\boldsymbol{x}}^{(1)} &=& \boldsymbol{b} + (\rho I - K_1)\boldsymbol{x}^{(k)} \\ (H_2 + \rho I)\tilde{\boldsymbol{x}}^{(2)} &=& \boldsymbol{b} + (\rho I - K_2)\boldsymbol{x}^{(k)} \\ \boldsymbol{x}^{(k+1)} &=& \frac{1}{2}(\tilde{\boldsymbol{x}}^{(1)} + \tilde{\boldsymbol{x}}^{(2)}) \end{cases} \tag{14}$$

$k = 0, 1, ...$ where $\boldsymbol{x}^{(0)}$ is an initial approximation to the solution $\boldsymbol{x}^*$ of (1).
Now, we define

$$M^{-1} = \frac{1}{2}((H_1 + \rho I)^{-1} + (H_2 + \rho I)^{-1}) \qquad \rho \geq 0 \tag{15}$$

If $N = M - A$, the matrix

$$Q = M^{-1}N = \frac{1}{2}((H_1 + \rho I)^{-1}(\rho I - K_1) + (H_2 + \rho I)^{-1}(\rho I - K_2)) \tag{16}$$

becomes the iteration matrix of the AM method. The following convergence theorems have been proved in [6].

Theorem 3. Let A of (2) be a real $n \times n$ symmetric positive definite matrix. Then, the iterative method (14) is convergent for all $\rho \geq 0$.

Theorem 4. Let A of (2) be an $n \times n$ irreducibly diagonally dominant real matrix (or strictly diagonally dominant real matrix), with $a_{ij} \leq 0$ for all $i \neq j$ and $a_{ii} > 0$, $i = 1, ..., n$. Then, the iterative method (14) is convergent for all $\rho > 0$ (or $\rho \geq 0$, respectively).

3 Parallel implementation

Methods (6) and (14) are characterized by having within their overall mathematical structure certain well-defined substructures that can be executed simultaneously. This feature makes the methods ideally suited for implementation on a multiprocessor system.
In order to make clear the intrinsec parallelism of both methods, we assume that any vector $\boldsymbol{z}$ with n components is partitioned commensurately with the block structure of the matrix A

$$\boldsymbol{z} = (\boldsymbol{z}_1^T, \boldsymbol{z}_2^T, ..., \boldsymbol{z}_q^T)^T$$

where $\boldsymbol{z}_i$ is a column vector with p components ($i = 1, ..., q$). Furthermore, we assume that any block structure of q elements (that is, any vector $\boldsymbol{z}$ or any block diagonal of matrix A) has a *ring structure*. Thus, we have $\boldsymbol{z}_{q+i} = \boldsymbol{z}_i$, $D_{q+i} = D_i$, $C_{q+i} = C_i$, $B_{q+i} = B_i$, $B'_{q+i} = B'_i$, for $i = 0, \ldots, q-1$.

In the case of the AGE method, the $(k+1)$th iteration consists of two sequential levels, the $(k+\frac{1}{2})$th and the $(k+1)$th level. At each level j ($j = k+\frac{1}{2}, k+1$), one must solve the following $q/2$ independent systems of order $2p$

$$(P'_i + \sigma I) \begin{bmatrix} \boldsymbol{y}_i^{(j)} \\ \boldsymbol{y}_{i+1}^{(j)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_i - (B'_i - \sigma I)\boldsymbol{y}_i^{(j-\frac{1}{2})} - D_i \boldsymbol{y}_{i-1}^{(j-\frac{1}{2})} \\ \boldsymbol{b}_{i+1} - (B'_{i+1} - \sigma I)\boldsymbol{y}_{i+1}^{(j-\frac{1}{2})} - C_{i+1}\boldsymbol{y}_{i+2}^{(j-\frac{1}{2})} \end{bmatrix} \tag{17}$$

where $i = 1, 3, ..., q-1$ if $j = k+\frac{1}{2}$ and $i = 2, 4, ..., q$ if $j = k+1$. Thus, at each iteration, the $q/2$ systems (17) of odd indices can be solved simultaneously on at most $q/2$ processors, obtaining the vector $\boldsymbol{y}^{(k+\frac{1}{2})}$, and subsequently the $q/2$ systems (17) of even indices can be solved also simultaneously on at most $q/2$ processors, obtaining the vector $\boldsymbol{y}^{(k+1)}$.

In the case of the AM method, at each iteration $k+1$ one must solve the following q independent systems of order $2p$

$$(P_i + \rho I) \begin{bmatrix} \tilde{\boldsymbol{x}}_i^{(j)} \\ \tilde{\boldsymbol{x}}_{i+1}^{(j)} \end{bmatrix} = \begin{bmatrix} \rho \boldsymbol{x}_i^{(k)} - D_i \boldsymbol{x}_{i-1}^{(k)} + \boldsymbol{b}_i \\ \rho \boldsymbol{x}_{i+1}^{(k)} - C_{i+1}\boldsymbol{x}_{i+2}^{(k)} + \boldsymbol{b}_{i+1} \end{bmatrix} \tag{18}$$

where $i = 1, ..., q$, $j = 1$ if i is odd and $j = 2$ if i is even. The q systems (18) can be solved simultaneously on at most q processors, obtaining the two vectors $\tilde{\boldsymbol{x}}^{(1)}$ and $\tilde{\boldsymbol{x}}^{(2)}$. Then, one must compute simultaneously on at most q processors the q block components $\boldsymbol{x}_i^{(k+1)}$ ($i = 1, ..., q$) of the vector $\boldsymbol{x}^{(k+1)}$:

$$\boldsymbol{x}_i^{(k+1)} = \frac{1}{2}(\tilde{\boldsymbol{x}}_i^{(1)} + \tilde{\boldsymbol{x}}_i^{(2)}) \tag{19}$$

On a system of π vector processors connected to a *shared memory* (no local memory within the processors is assumed), such as a Cray-like environment, the parallel implementation of both methods is easy. The data of problem (1) (the blocks $C_i, B_i, D_i, i = 1, ..., q$ and the vector $\boldsymbol{b}$) and the current vector approximation ($\boldsymbol{y}^{(k)}$ or $\boldsymbol{x}^{(k)}$) to the solution are stored in this shared memory.

In the case of the AM method, at each iteration the task of solving one of the q systems (18) is assigned to the first available processor by means of a dynamical scheduling. Since these concurrent tasks have the same complexity, some processors solve $\lceil q/\pi \rceil$ systems, other processors solve $\lfloor q/\pi \rfloor$ systems. Then, it is necessary a synchronization of all the processors (synchronization point) to make sure that the vectors $\tilde{\boldsymbol{x}}^{(1)}$ and $\tilde{\boldsymbol{x}}^{(2)}$ are available in the shared memory. After this synchronization point the q tasks (19) are executed on π processors with a similar dynamical way to assign the workload.

In the case of the AGE method, the $q/2$ systems of each one of the two sequential levels $k+\frac{1}{2}$ and $k+1$ are assigned to π processors by means of the same dynamical scheduling technique. In this case we have a synchronization point between the two levels and any processor solves about $\lfloor q/2\pi \rfloor$ or $\lceil q/2\pi \rceil$ systems at each level.

For both methods, we have a synchronization point at the end of each iteration.
If t_S is the time for solving a system of order $2p$ and t_F is the time for computing p sums, the elapsed times of the AGE and AM iteration on π processors have the following expressions:

$$\tau_\pi^{AGE} = 2\lceil q/2\pi \rceil t_S + \gamma_1 + \gamma_2; \qquad \tau_\pi^{AM} = \lceil q/\pi \rceil t_S + \lceil q/\pi \rceil t_F + \gamma_3 + \gamma_4 \tag{20}$$

where γ_i $(i = 1, 2, 3)$ is the delay of a synchronization point that arises after solving a set of systems (17) or (18) and γ_4 is the delay of the synchronization point after computing (19). The parameter γ_i $(i = 1, 2, 3)$ is at most equal to t_S while γ_4 is at most equal to t_F. Thus, when the workload is not well-balanced (π is not a divisor of $q/2$), $\gamma_i \geq \gamma_4$, $(i = 1, 2, 3)$. Furthermore $2\lceil q/2\pi \rceil t_S \geq \lceil q/\pi \rceil t_S$ (the equality holds when π is a divisor of $q/2$) and the time t_F (for computing p sums) can be considered negligible with respect to the time t_S (for solving a system of order $2p$). Then, when we use the same number of processors π for both methods, we have that the cost of the AM iteration does not exceed the cost of the AGE iteration. Also when π is a divisor of $q/2$, and in theory, the workload is perfectly balanced, τ_π^{AM} is not significantly different from τ_π^{AGE}.

On a *distributed memory* system with a local memory, message passing environment, such as a network of transputers, it must take particular attention to the data distribution and the data communication among the processors. The ring structure of the data suggests that the best topology for the interconnection scheme of the processors is a ring network. In this case, the processor T_l, $l = 1, ..., \pi$ is a node connected by bidirectional communication links to the neighbouring processors T_{l+1} and T_{l-1} (we put $T_{\pi+1} = T_1$ and $T_0 = T_\pi$).
For simplicity, we describe the implementation of the two methods on a ring network, assuming that π is a divisor of q, so that $\nu = q/\pi$. In order to minimize the communications among the nodes, we allocate for methods (6) and (14) in the local memory of the node T_l the diagonal blocks of matrix A with indices $j = (l-1)\nu + i$, $i = 1, ..., \nu + 1$ and the corresponding block components $\boldsymbol{b}_j$ of the vector $\boldsymbol{b}$. Furthermore, at the beginning of the $(k+1)$th iteration we assume that the block components of indices $j = (l-1)\nu + i$, $i = 1, ..., \nu + 1$ of the current vector approximation ($\boldsymbol{x}^{(k)}$ or $\boldsymbol{y}^{(k)}$) are allocated in T_l. Besides, at the beginning of each iteration (or level), also the block components of $\boldsymbol{x}^{(k)}$ (or $\boldsymbol{y}^{(k)}$) required to compute the right hand sides of the systems (17) (or (18) respectively) must be allocated in T_l. That is obtained by communications between neighbouring nodes.

For example, if we consider the problem arising from the discretisation of a two dimensional elliptic partial differential equation on a rectangular domain D and if the $p \times q$ interior points of the grid superimposed on D have a row-wise ordering, the data organization may be interpreted as a decomposition strategy in the following way : we decompose the domain in π subdomains D_l, $l = 1, ..., \pi$, having a common row with the neighbouring subdomains and we assign to node T_1 the rows $1, 2, ..., \nu + 1$, to node T_2 the rows $\nu + 1, \nu + 2, ..., 2\nu + 1$... and to the last node T_π the rows $(\pi - 1)\nu + 1, ..., q - 1, q, 1$. (Here, we assume $D_{\pi+1} = D_1$, $D_0 = D_\pi$.) In the AGE method, when ν is even, at the beginning of level $k + \frac{1}{2}$, T_l requires from T_{l-1} the approximation values on the last and the last but one rows of subdomain D_{l-1} (computed at iteration k) and, at the beginning of level $k + 1$, T_l requires from T_{l+1} the approximation values on the first and the second rows of subdomain D_{l+1} (computed at level $k + \frac{1}{2}$). When ν is even, at the beginning of

level $k+\frac{1}{2}$ (level $k+1$), only the nodes T_l with l odd (even respectively) require from T_{l-1} the approximation values on the last and the last but one rows of subdomain D_{l-1} and from T_{l+1} the approximation values on the first and the second rows of subdomain D_{l+1}. (computed at the previous level). In the AM method, at the beginning of iteration $k+1$, the node T_l requires from T_{l+1} the approximation values on the second row of subdomain D_{l+1} and from T_{l-1} the approximation values on the last but one row of subdomain D_{l-1} (computed at iteration k).

The two methods (6) and (14) require testing for convergence at any iteration. On a distributed memory system each node has to find out if any other node in the network has obtained the convergence. This global communication operation introduces a significant amount of overheads. The overheads increase as the size of the network increases. The overheads can be minimized if the convergence test is performed after a prefixed number of iterations, and it is performed at each iteration only when the error becomes lower than a weak tolerance.

In the case of the AM method, the work at each iteration $k+1$ is distributed among the π nodes ($\pi \leq q$) so that the node T_l executes the following steps:

1. *for* $i = (l-1)\nu+1, ..., l\nu$, compute the right hand side of system (18) of index i
2. *for* $i = (l-1)\nu+1, ..., l\nu$, solve system (18) of index i for the block components $\tilde{\boldsymbol{x}}_i^{(j)}$ and $\tilde{\boldsymbol{x}}_{i+1}^{(j)}$, where $j = mod(i-1,2)+1$ (with $mod(\alpha,\beta) = \alpha - \lfloor\frac{\alpha}{\beta}\rfloor\beta$)
3. send $\tilde{\boldsymbol{x}}_{(l-1)\nu+1}^{(mod((l-1)\nu,2)+1)}$ to T_{l-1}, receive $\tilde{\boldsymbol{x}}_{l\nu+1}^{(mod(l\nu,2)+1)}$ from T_{l+1}
4. send $\tilde{\boldsymbol{x}}_{l\nu+1}^{(mod(l\nu+1,2)+1)}$ to T_{l+1}, receive $\tilde{\boldsymbol{x}}_{(l-1)\nu+1}^{(mod((l-1)\nu+1,2)+1)}$ from T_{l-1}
5. *for* $i = (l-1)\nu+1, ..., l\nu+1$, compute $\boldsymbol{x}_i^{(k+1)} = \tilde{\boldsymbol{x}}_i^{(1)} + \tilde{\boldsymbol{x}}_i^{(2)}$
6. send $\boldsymbol{x}_{(l-1)\nu+2}^{(k+1)}$ to T_{l-1}, receive $\boldsymbol{x}_{l\nu+2}^{(k+1)}$ from T_{l+1}
7. send $\boldsymbol{x}_{l\nu}^{(k+1)}$ to T_{l+1}, receive $\boldsymbol{x}_{(l-1)\nu-1}^{(k+1)}$ from T_{l-1}
8. *test of convergence*

In the case of the AGE method, the work at each iteration $k+1$ is distributed among π nodes ($\pi \leq q/2$) so that the node T_l executes the following steps:

1. *for* $i = (l-1)\nu+1, ..., l\nu$, *if* i is odd *then* compute the right hand side of system (17) of index i and solve the system for the block components $\boldsymbol{x}_i^{(k+\frac{1}{2})}$ and $\boldsymbol{x}_{i+1}^{(k+\frac{1}{2})}$
2. $g1 = mod((l-1)\nu+1, 2)$, $g2 = mod(l\nu, 2)$
3. *if* $g1 = 1$ *then* send $\boldsymbol{x}_{(l-1)\nu+1}^{(k+\frac{1}{2})}$ and $\boldsymbol{x}_{(l-1)\nu+2}^{(k+\frac{1}{2})}$ to T_{l-1} *else* receive $\boldsymbol{x}_{(l-1)\nu}^{(k+\frac{1}{2})}$ and $\boldsymbol{x}_{(l-1)\nu+1}^{(k+\frac{1}{2})}$ from T_{l-1}
4. *if* $g2 = 1$ *then* send $\boldsymbol{x}_{l\nu}^{(k+\frac{1}{2})}$ and $\boldsymbol{x}_{l\nu+1}^{(k+\frac{1}{2})}$ to T_{l+1} receive $\boldsymbol{x}_{l\nu+1}^{(k+\frac{1}{2})}$ and $\boldsymbol{x}_{l\nu+2}^{(k+\frac{1}{2})}$ from T_{l+1} *else*

5. *for* $i = (l-1)\nu + 1, ..., l\nu$, *if* i is even *then* compute the right hand side of system (17) of index i and solve the system for the block components $\boldsymbol{x}_i^{(k+1)}$ and $\boldsymbol{x}_{i+1}^{(k+1)}$

6. *if* $g1 \neq 1$ *then* send $\boldsymbol{x}_{(l-1)\nu+1}^{(k+1)}$ and $\boldsymbol{x}_{(l-1)\nu+2}^{(k+1)}$ to T_{l-1} *else* receive $\boldsymbol{x}_{(l-1)\nu}^{(k+1)}$ and $\boldsymbol{x}_{(l-1)\nu+1}^{(k+1)}$ from T_{l-1}

7. *if* $g2 \neq 1$ *then* send $\boldsymbol{x}_{l\nu}^{(k+1)}$ and $\boldsymbol{x}_{l\nu+1}^{(k+1)}$ to T_{l+1} *else* receive $\boldsymbol{x}_{l\nu+1}^{(k+1)}$ and $\boldsymbol{x}_{l\nu+2}^{(k+1)}$ from T_{l+1}

8. *test of convergence*

The final approximation $\boldsymbol{x}^{(k^*)}$ or $\boldsymbol{y}^{(k^*)}$ is distributed among the π nodes so that the block components of indices j, $j = (l-1)\nu + 1, ..., l\nu$ is in T_l. All the data transfers arise between adjacent nodes.
In both methods, for any node the total number of communications per iteration is equal to $4p$ send operations and $4p$ receive operations; the total number of data transfers is equal to $4p$ for each node. The elapsed times for the AM iteration and the AGE iteration are:

$$\lambda_\pi^{AM} = \frac{q}{\pi}t_S + \frac{q}{\pi}t_F + 4pf + \xi_T(\pi); \qquad \lambda_\pi^{AGE} = 2\lceil q/2\pi \rceil t_S + (1 + mod(\nu, 2))4pf + \xi_T(\pi) \quad (21)$$

where f is the time to transfer (send and receive) one datum between adjacent nodes and $\xi_T(\pi)$ is the required time for convergence testing. When π is not a divisor of q (the workload is not well balanced for both methods), formulas (21) become:

$$\lambda_\pi^{AM} = \lceil q/\pi \rceil t_S + \lceil q/\pi \rceil t_F + 4pf + \xi_T(\pi); \qquad \lambda_\pi^{AGE} = 2\lceil q/2\pi \rceil t_S + 8pf + \xi_T(\pi) \quad (22)$$

Formulas (21) and (22) are similar to formulas (20). Thus, on a distributed memory system, we can repeat similar considerations of the case of a shared memory system.
We conclude that, if π and q are fixed, the effectiveness of the two methods depends on the number of the iterations of each method and on the synchronization/communication cost. Neverthless, if we use 2π processors ($2\pi \leq q$) for the AM method, we can take advantage of the higher degree of parallelism that the AM method has with respect to the AGE method. Thus, the AM method is convenient when we have at our disposal a large number of processors.

4 Computational Experiments

In this section we compare the effectiveness of the AGE method and the AM method on a set of test problems arising from the discretisation by five points finite difference approximation formula of a two dimensional elliptic partial differential equation on a rectangular domain D ([6], [7], [8]). The mesh spacings in both directions is equal to $h = 1/(p+1)$ with $p = q$. The systems (17) and (18) are solved by a cyclic reduction solver described in [9] and [6]. The convergence test used is $\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|_\infty \leq 10^{-14}\|A\|_\infty$ and the initial approximation is the null vector.
Some results of a large number of numerical experiments are shown in tables 1–2–3; these experiments are carried out on Cray Y-MP (tables 1 and 2) and on two Quintek FAST9

Table 1: Experiments on Cray Y-MP; $p = q = 128$.

	AGE [(8)]				AM		
Test problem	k^*	*error*	*time* $\pi = 2$	*time* $\pi = 4$	k^*	*error*	*time* $\pi = 4$
[6](quasi-symmetric)	11478	4.0-7	46.30	26.54	11481	4.0-7	24.51
[6](asymmetric)	99	1.2-10	0.40	0.23	107	2.6-10	0.23
[7]($\beta = 4$)	220	7.0-13	0.88	0.51	236	8.9-13	0.50
[7]($\beta = 8$)	126	1.1-12	0.50	0.29	142	2.0-12	0.30

boards plugged into a PC Intel 486 host computer (table 3). Each board is fitted with 9 transputers T805 running at 25Mhz with 4Mb of local storage. The communications among the nodes are handled by using the Express library [10]. In the tables, k is the number of iterations for an *error* $\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^*\|_\infty$, *time* is the elapsed time expressed in seconds for solving the problem with π processors. In tables 1-2-3 the value ρ is equal to 0 for the AM method and $\sigma = max(a_{ii}/2)$ for the AGE method corresponding to matrix (8). In table 2 we report also the results for the AGE method corresponding to matrix (10) where $\Sigma = diag\{\sigma_i\}$ with $\sigma_i = a_{ii}/2$; for the cases of table 2, the *error* assumes about the same values for all the methods.

Numerical experiments show that the two methods are particularly effective for strongly asymmetric matrices (see tables 1-2).

The AGE variant with the diagonal matrix $\Sigma = diag\{a_{ii}/2\}$ is more efficient than the AGE method with $\Sigma = \sigma I$ ($\sigma = max(a_{ii}/2)$).

In all the experiments, the difference $k^*_{AM} - k^*_{AGE}$ between the number of iterations of the AM method and the best choice of Σ in the AGE method is at most equal to 26% of the AGE iterations.

The experiments confirm the considerations in section 3 about the implementation of the two methods on shared and distributed memory systems. Indeed, on the Cray, the elapsed time of the AGE iteration is $\tau_2^{AGE} = .40 \cdot 10^{-2}$ and $\tau_4^{AGE} = .23 \cdot 10^{-2}$ seconds ($p = q = 128$) while the elapsed time of the AM iteration is $\tau_4^{AM} = .21 \cdot 10^{-2}$. On a network of π tansputers, when $p = q = 64$ we have $\lambda_2^{AGE} = .29$, $\lambda_4^{AGE} = .15$, $\lambda_8^{AGE} = .8 \cdot 10^{-1}$, $\lambda_{16}^{AGE} = .58 \cdot 10^{-1}$, while $\lambda_2^{AM} = .26$, $\lambda_4^{AM} = .13$, $\lambda_8^{AM} = .72 \cdot 10^{-1}$, $\lambda_{16}^{AM} = .46 \cdot 10^{-1}$. Then, in practice, even if the workload is well-balanced, we have $\tau_4^{AM} < \tau_4^{AGE}$ and $\lambda_\pi^{AM} < \lambda_\pi^{AGE}$. Moreover, when we double the number of processors on Cray, the speed-up ($\tau_2^{AGE}/\tau_4^{AGE}$) of the AGE iteration is about 1.7 while τ_2^{AGE}/τ_4^{AM} is about 1.9. Similar considerations hold on the network of transputers. Then, for a fixed number of processors, the efficiency of the AM method with respect to the AGE method depends on the difference of the iterations of the two methods. We observe experimentally that, when this difference is less than $10\% k^*_{AGE}$, the AM method is more convenient of the AGE method. When for the AM method we use 2π processors, the AM method is always more convenient of the AGE method. Besides, the efficiency ([1, p. 23]) of the AGE method varies from 0.99 ($\pi = 2$) to 0.63 ($\pi = 16$), while the same ratio of the AM method varies from 0.99 ($\pi = 2$) to 0.70 ($\pi = 16$).

Table 2: Experiments on Cray Y-MP; $p = q = 128$.

	AGE [(8)]			AGE [(10)]			AM		
Test problem	k^*	*time* $\pi = 2$	*time* $\pi = 4$	k^*	*time* $\pi = 2$	*time* $\pi = 4$	k^*	*time* $\pi = 4$	*error*
[7] ($\delta = 1$) (quasi-symmetric)	14162	56.65	32.57	9951	39.80	22.89	12292	25.81	5.2-5
[7] ($\delta = 100$) (asymmetric)	560	2.24	1.29	380	1.52	0.87	394	0.83	1.2-4
[8]	7179	8.94	5.08	5512	7.17	3.89	6070	4.06	4.4-7

Table 3: Experiments on a network of transputers; p=q=64.

Test problem	*Method*	k^*	*error*	*time* $\pi = 2$	*time* $\pi = 4$	*time* $\pi = 8$	*time* $\pi = 12$	*time* $\pi = 16$
[7]	AGE	135		38.73	19.99	11.18	9.64	7.82
($\beta = 1$)	AM	143		36.95	18.94	10.25	8.39	6.54
[6]	AGE	82	1.4-13	23.64	12.20	6.80	5.87	4.76
	AM	101	1.3-13	26.18	13.42	7.26	5.94	4.62

References

[1] Ortega J.M.: *Introduction to parallel and vector solution of linear systems*, Plenum Press, New York (1985).

[2] Evans D.J.: *Group explicit iterative methods for solving large linear systems*, Intern. J. Computer Math., 17 (1985), 81–108.

[3] Evans D.J., Yousif W.S.: *The solution of two-point boundary value problems by Alternating Group Explicit (AGE) method*, SIAM J. Sci. Stat. Comput., 9,3 (1988), 474–484.

[4] Varga R.S.: *Matrix Iterative Analysis*, Prentice Hall Inc., Englewood Cliffs, N. J. (1962).

[5] Ortega J.M.: *Numerical analysis*, Academic Press, New York (1972).

[6] Ruggiero V., Galligani E.: *A parallel algorithm for solving block tridiagonal linear systems*, Computers Math. Applic., 24,4 (1992), 15-21.

[7] Sonneveld P.: *CGS, a fast-type solver for non symmetric linear systems*, J. Sci. Stat. 10,1 (1989) 36-52.

[8] Smolarski D. C., Saylor P. E.: *An optimum iterative method for solving any linear system with a square matrix*, BIT, 28 (1988) 163–178.

[9] Galligani E., Ruggiero V.: *A parallel preconditioner for block tridiagonal matrices*, ParCo 93, Grenoble (1993).

[10] *Express User's Guide*, ParaSoft Corporation, Livingston (1990).

INVESTIGATION OF PARALLEL ALGORITHMS FOR SOLVING THE BOLTZMANN EQUATION AND THEIR EFFICIENCY

V.V.ARISTOV, I.G.MAMEDOVA

Computing Centre of the Russian Academy of Sciences
Moscow, 117967, Vavilova 40
e-mail: aristov@sms.ccas.msk.su
fax: (095) 135-61-59

Abstract

Possibilities of concurrent calculations based on the direct methods for solving the Boltzmann equation are studied. Simple transformation of the serial algorithms to the parallel ones is suggested. Various homogeneity features of the numerical schemes are analyzed. A number of problems of kinetic theory of gases are investigated which give high efficiency and speed-up. Special attention is paid to the value called serial fraction which is one of the important characteristic of parallel algorithms. It is noted that for the regular method of integration of the Boltzmann equation the efficiency is sometimes more than unity.

Keywords: the Boltzmann equation, homogeneous schemes, parallel program, serial fraction

1 Introduction

In [1] the principal possibility of applying the parallel algorithms in direct numerical methods for solving the Boltzmann kinetic equation has been shown. The present paper concerns the further study of the Boltzmann parallel schemes which exposed a number of interesting features.

For such a complex problem like the problem of solving the Boltzmann equation the use of parallel algorithms running on the new computer systems is one of the most important directions to speed-up calculations and increase the amount of data. Investigations on analysis, constructions and implementation of these algorithms have been begun recently. Several parallel algorithms were studied for the methods of direct statistical simulation in rarefied gas dynamics, e.g. [2, 3]. Though, as emphasized there, original statistical algorithms are more suitable for serial calculations, one can reach rather high efficiency in their parallel implementation.

For our purposes we took advantages of the features of homogeneity of the scheme used in the direct numerical solution of the Boltzmann equation [4]. The schemes for direct solving the Boltzmann equation, as mentioned in there, are much more suitable for parallel calculations by nature. (In statistical simulation methods the number of particles in different parts of the coordinate space are different, while in direct methods as usual the same number of discrete velocities is taken). Few attempts of parallel processing on vector computers demonstrated their high efficiency when running on the computers of such kind. Recently in [5] the first results of parallel implementation of the direct methods have been obtained. Here the conservative splitting method [6] was applied. The one-dimensional problem of evaporation/condensation was considered. The efficiency of concurrent calculations was found rather high.

In the present paper one-, two- and three-dimensional problems are under consideration. It is shown that the peculiarity of the kinetic equation gives good opportunities for parallelizing.

2 The general features of kinetic schemes for parallel processing

Let's investigate briefly the schemes of the direct method for solving the Boltzmann equation so as to expose the general features of those which enables to parallelise the algorithms in use efficiently. As well-known the Boltzmann equation is written as follows

$$\frac{\partial f}{\partial t} + \xi \frac{\partial f}{\partial x} = I.$$

Here $f = f(t, x, \xi)$ is the distribution function, x is the space vector, $\xi = (\xi_x, \xi_y, \xi_z)$ is the velocity vector, $I(f, f)$ is the integral collision operator.

For solving the kinetic equation the discrete velocity approach is used, so the grid in velocity space is fixed accordingly, finite domain of integration chosen *a priori* (cube or parallelepiped) which is the same for all the points of the physical space. The domain is divided into cubical cells of the same size and the value of the distribution function is set to the one in the center of according cube, i.e. piecewise-constant approximation is applied. Macroparameters of the distribution function are calculated as a simple sums over velocities in rectangular quadrature formula.

It is natural to use the quadrature square formula by average point in this case (using a formula of higher order will result in that the interpolation error on evaluating the distribution function gets more in value). As shown previously, such a formula is found useful as far as it allows to reduce the error appearing on evaluating the distribution function. In general form the α-moment evaluation of distribution function is represented as follows:

$$M^{(\alpha)} = \sum_r \xi^\alpha f_r (\Delta\xi)^3,$$

where r is the number of the cell in the velocity space, $\Delta\xi$ is the the size of a cubic cell in the velocity space. The strong point of this quadrature formula is that all the weights are equal, so it gets simpler to parallelise the calculation of macroparameters (it would be more difficult to do that for more complicated quadrature formulae).

The next point which is essential for parallel processing is homogeneity of integral evaluation schemes. The same number of trials is used in the Monte Carlo method (or quasi-Monte Carlo method). In the regular method the same set of quadratic nodes is applied according to the nodes in the velocity space.

For instance, let's put down the quasi-random quadrature formulae for evaluating five-fold collision integrals making use of so-called Korobov's sequences supplying uniform distribution of nodes (see [4]). Here the following quadrature formulae with equal positive weights are applied, (they are written for the molecular model of hard spheres):

$$\nu = \frac{V}{L_c} \sum_{\gamma=1}^{L_c} f(x, \xi_{1\gamma}) \sin\theta |\xi - \xi_{1\gamma}|$$

$$N = \frac{V}{L_c} \sum_{\gamma=1}^{L_c} f(x, \xi'_{\gamma}) f(x, \xi'_{1\gamma}) \sin\theta |\xi - \xi_{1\gamma}|,$$

where ν and N are frequency and inverse collision integrals accordingly, ξ_1 is the velocity vector of integration, θ is one of the impact angles, primed indices denote velocities after collisions, V is the volume of the five-dimensional cube of integration (three dimensions concern the velocity and two dimensions the impact angles), L_c is the number of nodes, γ numbers to points in the cube. The set of Korobov's numbers for each discrete velocity is the same. It is the condition for achieving good paralleling.

The approach of reducing dispersion sometimes used in Monte Carlo method means the number of trials increases for points with more essential values of distribution function for which reason it has not been used.

In the method of regular integration [4] for the discrete velocity approach, or piecewise-constant approximation in velocity space, one can succeed in exact integrating over angles (for hard spheres), and the right hand side of the Boltzmann equation is approximated as follows:

$$I_r = \sum_{p,q} A_{pqr} (f_p f_q - f_r f_{r_1})$$

Here the integer index r notes the velocity point, r_1 is represented in terms of indices p, q, r.

Hence, for all the points of the physical space one and the same matrix of constant coefficients is used (it can be regarded in a certain sense as a system of coefficients in quadrature formulae for evaluating collision integrals).

For collision integral evaluation used in Monte Carlo (or quasi-Monte Carlo) technique two variants of loop organization are possible:

a) outer loop is dealing with velocities in which case one and the same set of random vectors in each point is taken, and that essentially reduces the time required for calculations in the serial algorithm (compared with variant b)). The amount of calculations is turned out to be different then. In fact, for some values of velocities the hatched velocities are beyond the region controlled, so they do not make contributions in the sums calculated. It leads to less perfect load balancing between processors.

b) outer loop is dealing with physical variable in which case the number of calculations for each point is the same. In every point of physical space one and the same number of

velocities and, accordingly, the same set of velocities after collisions is involved in the region for which calculations are done (the number of velocities being beyond of this region is one and the same for every physical point too).

In what follows parallel algorithms only for variant b) are dealt with. For variant a) the efficiency of the parallel algorithm is turned out to be worse. By this reason when testing the two- and three-dimensional problems which are based on the same scheme of integral evaluation variant b) is chosen.

Another important feature of the algorithms which makes them well suitable for parallelizing is the conservative splitting method [6]. Here the main point is that no connection exists at the relaxation stage, therefore the simple decomposition technique can be adopted over the physical space. It is worth emphasizing that the relaxation stage is the most time-consuming (here the collision integrals are evaluated), while the influence of free-molecular stage where the communications between processors take place is not much essential. At this stage explicit (or further discussed explicit-implicit scheme) does not deteriorate the features supporting parallelizing. The schemes of all the stages of splitting method are represented in [1].

The scheme of free-molecular moving is explicit which is important for parallel processing as far as information is only sent to neighbors. This is the first order difference scheme by time and space steps Δt, Δx, Δy, Δz and it is monotonous. For stability of the difference scheme the numerical parameters should be bind by the Courant's condition. The requirement mentioned can be quite strong and slow down the process of converging to the solution of the stationary problem as far as the parameter δt is rather small. With the aim of accelerating of that iterative convergence to the stationary solution the following explicit-implicit scheme has been suggested which does not exercises influence on efficiency of parallel algorithm:

$$\frac{f^l_{ijk} - f^{l-1}_{ijk}}{\Delta t} + \xi_x \frac{f^l_{ijk} - f^{l-1}_{i-1jk}}{\Delta x} + \xi_y \frac{f^l_{ijk} - f^{l-1}_{ij-1k}}{\Delta y} + \xi_z \frac{f^l_{ijk} - f^{l-1}_{ijk-1}}{\Delta z} = 0$$

($\xi_x > 0, \xi_y > 0, \xi_z > 0$), where l denotes the number of time levels, i, j, k are spatial indices, for other set of signs of velocities the scheme is alike to that.

It is easy to see that the scheme is absolutely stable, hence it is possible to use large iterative parameter Δt. The scheme in use is converging to the steady solution of the Boltzmann equation.

3 Organization of parallel algorithms and programs

One of the strong points of the Boltzmann equation is that there exist two variables (three-dimensional), accordingly, in physical and velocity space and they behave differently in the right-hand side and the left-hand side. The left convective side represents transfer operator acting in physical space. The right-hand side is collision operator acting in velocity space for the fixed point in physical space. This, in principle, gives various ways for parallelizing.

The main stages of algorithm are the following: collisionless particle transport, uniform relaxation, the calculation of macro parameters, conservative correction. The last two stages could be successfully parallelized by both physical and velocity variables. In fact, both the calculation of moments and the distribution function after the conservative

correction is carried out in each point of the physical space when the loop the index of which is a spatial variable is parallelized.

For the conservative correction the calculation of directed distribution function is performed in each velocity point too. Full homogeneity of the calculation of macroparameters is characteristic of the calculation of moments as mentioned above. It is the only problem occurring on parallelizing by velocity in regular method that it is needed to calculate partial sums in each processor and gather them for further calculation.

Space-uniform relaxation stage is most effectively parallelized (for quasi-Monte Carlo method of collision integral evaluation). In case of regular method of calculation the parallel algorithm can be organized in various ways (both variants a) and b)) and each way is quite efficient.

At the free-molecular stage exchanges between processors are necessary while no communications are necessary at this stage on parallelizing by velocity variable.

4 Implementation of the algorithms

For implementation the considered problems of kinetic gas theory on the multi-processor computer system the approach called data parallelism, or geometric parallelism, is taken. It implies the dividing of calculation region into subregions which are in accordance with different processors of the system. That approach to parallelizing the problems mentioned was caused by the serial available algorithms for their solution adopted to multi-processor computer systems.

Taking the one-dimensional problem of wave shock as an example one can show how the dividing of calculation region (no matter physical or velocity) has been done (supposing linear topology of processor connection). The number of subregions is equal to the number of processors. The dividing has been done so as to place about the same number of calculation points in each processor. Some of processors treat m points from the calculation region and the others $m + 1$. In order to make parallel processing much faster it is appropriate to allocate one and the same number of points to each processor.

Calculations in every point of the interval $[1..M]$ where M is the number of points in the processor are carried out independently in each processor. And the boundary points of each processor are the only ones in which information from the neighbors is necessary for continuing the calculation. Those information exchanges are only necessary at the free-molecular stage. The values of the distribution function of a new layer in the boundary points of each processor are defined from either boundary conditions (in the first or the last processors) or by using information from neighbor points. Indeed, $M + 1$th point of ith processor is the first point of $i + 1$th processor, and it is send to processor $i + 1$ for further calculation.

Due to the fact that the amount of information being sent to one another is turned out not to be much it does not essentially affect efficiency of parallelized algorithms. In Figure 1 the process of interprocessor exchange is shown.

It is known that various interprocessor communication topology could be created on base of transputers. Two different topologies were under testing on parallelizing the one-dimensional problem of wave shock solved by regular method. The serial algorithm based on regular method is simple for implementation, but one of its points is the use of matrix of coefficients of large size. In parallel implementation of that problem only part

of this matrix is located to each processor which allows to increase the size of the matrix as much as few times. It is worth while to emphasize that for the problem which is spoken about this fact is more essential than to obtain sufficient values of efficiency and speed-up.

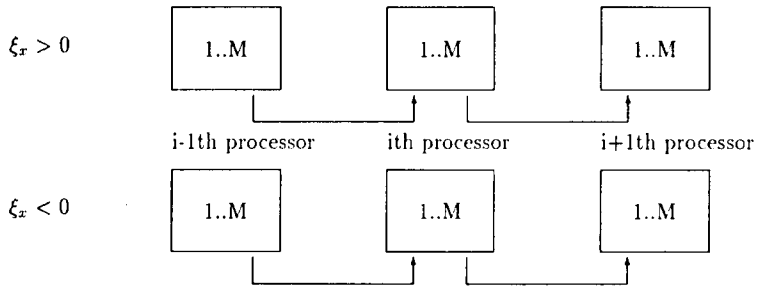


Figure 1

Note that all the stages of the program are parallelized by velocity space when using linear topology of connection of processors. At the stage of relaxation full parallelizing is achieved due to the same amount of calculation in each processor in performing the matrix A_{pqr}, here is the number of processor. No communication between processors is required at the free-molecular stage which is not the case when using another method of calculation. To avoid completely all the exchanges between processors is not possible in so far as macroparameter values must be gathered over all the velocity points while they are distributed over the processors. Besides, at the relaxation stage a value of distribution function in every point of velocity space should be known which leads to the necessity of saving the matrix representing the distribution function in each processor and, consequently, it leads to extra expenses for exchange between processors (after the matrix of distribution function gathers from each processor it is sent over all the processors linearly connected).

If using the topology of lattice all the stages of the program are parallelized by both velocity and physical space, in which case extra information exchanges between processors concerning the physical space in adding to those mentioned above (regarding the velocity space) appears. The way in which those exchanges are performed was considered early. This is illustrated schematically in Figure 2.

In this case it was natural to expect that the results in regard to efficiency and speed-up would be less sufficient than in case of linear topology of connection of processors. It is sure to result from the following: the information occupying a continuous area within computer memory is sent along the horizontal of processors while the information occupying piecewise-continuous area is sent along verticals, which results in increasing exchange expenses.

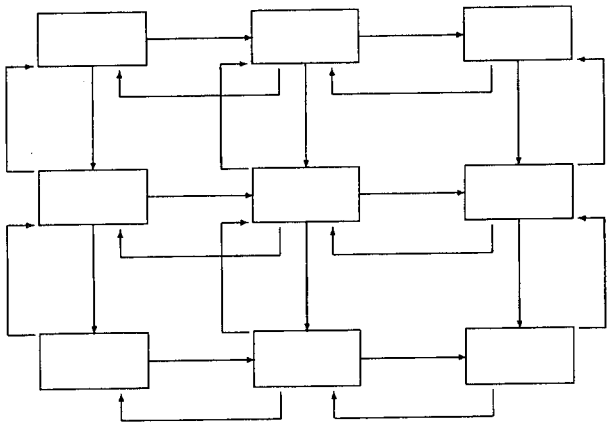


Figure 2: *horizontal arrows show exchange between processors by macroparameters and distribution function; vertical arrows show exchange by spatial coordinates*

Linear topology is under test on parallelizing the two- dimensional unsteady (reflection of shock wave from a wedge) and three-dimensional (freejet flows) problems. In both the problems parallelized loops are organized so that the loop by a physical coordinate is outer, e. i. like in the problem of shock wave. Distribution of points of calculation region over the processors and interprocessor information exchange are like to those in the problem of shock wave. It is worth emphasizing that for handling calculations in three-dimensional problem large amount of memory is required. The use of multi-processor machines make it possible to manage this problem successfully.

2 Results on concurrent calculations

The problems considered are analyzed not only in regard to efficiency and speed-up but for another value called serial fraction, this measurment has been introduced in [7]. That value permits to judge some advantages and disadvantages of parallel implementation which can not be exposed by looking at a value of efficiency and speed-up. for example, it is known that if speed-up is close to linear as new processors are being added that means a program is well-parallelized. But how close to linear is good enough? Then, the behavior of efficiency as the number of processors is changing tells us how close we are getting to the best our hardware can do. But if efficiency is not particularly high then what is the reason of it? That is where the value mentioned comes in.

Serial fraction is defined as $g = T_s/T(1)$ (see [7]), where T_s is the time taken by the part of the program that must be run serially in paralliling and $T(1)$ is the time needed on one processor.In terms of efficiency e that measurement could be represented as $g = (1/e - 1)/(p - 1)$, here p is the number of processors.Irregular change in g as p increases warns of load imbalansing of transputer machine. A smooth increase in the serial

	p	s	e	g
1D	*2*	*2.0*	*1.0*	*0.0025*
	7	*6.95*	*0.99*	*0.0012*
2D	*2*	*1.97*	*0.987*	*0.0132*
	4	*3.84*	*0.959*	*0.0142*
	7	*6.44*	*0.92*	*0.0144*
3D	*2*	*1.92*	*0.96*	*0.042*
	4	*3.76*	*0.94*	*0.0139*

Table 1: quosi-Monte Carlo

	p	s	e
pipeline	*2*	*2.04*	*1.02*
	3	*3.04*	*1.013*
	6	*5.9*	*0.983*
lattice	*4*	*2.628*	*0.657*
	6	*5.182*	*0.864*

Table 2: Regular method

fraction g as p increases warns of increasing the overheard of synchronizing processors. If the value of g is a constant (or nearly constant) then we have neither load imbalance nor increasing overheard.

In Table 1 speed-up s and efficiency e as well as the values of serial fraction evaluated for the one-dimensional shock wave problem (with the quasi-Monte Carlo method of integral evaluation), two-dimensional unsteady problem on reflection shock wave from a wedge and three-dimensional freejet flow problem are shown. It is important that the value of g should not increase (or it should be increasing smoothly at least). And it was the case in our experiences.

A great attention has been paid to the regular method of collision integral evaluation in our recent experiences for which all the features above are characteristic. The results are shown in Table 2. It is worth emphasizing that for this method the problem is not only to speed up computations but to save computer memory too, and more, saving computer memory is come out to be much more important. All the stages of the program were sufficiently effective parallelized by velocity in spite of exchanging between processors.

Indeed, the main stage (the uniform relaxation) is naturally divided into uniform parts as the matrix A_{pqr} is distributed by velocity index r through processors. So for each r_0 we have two-dimensional matrix A_{pqr_0} allocated to processor r_0.

It is founded that the efficiency of parallel computations for sufficiently large number of velocity points can be more than 1 in well-parallelized problems like the algorithm with the regular method of integral evaluation. This appears to be explained by transputer memory organization. An Inmos transputer has on-chip memory which is little in amount but very fast in data exchange. As the number of processors increases adding processors add that fast memory which reduced overheard and elapsed time for exchanging between processors.

The algorithm is also quite suitable for parallelizing using the matrix topology of processor connection (with velocity horizontals and spatial coordinate verticals). The explanation of it is given in 4.

For implementing the parallel program 3L PARALLEL FORTRAN77 language and 20 Mhz Inmos T800 Transputers hosted in a personal computer were applied.

References

[1] Aristov V.V., Mamedova I.G. Parallel implementation of numerical algorithms for solving the Boltzmann equation. Proceed. of Intern. Conf. on Parallel Comput. Technologies (PACT-93), 1993, Novosibirsk. v.1, p.103-109.

[2] Wilmoth R.G. Adaptive domain decomposition for Monte Carlo simulation on parallel processors. Proceed. 17th Intern. Symp. on Rarefied Gas Dynamics. 1991. ed. A.E.Beylich, Weinheim, P. 700-716.

[3] Long L.N., Wong B., Maczkowski J. Deterministic and non-deterministic algorithms for rarified gas dynamics on massively parallel computers. Book of Abstracts. 18th Intern. Symp. on Rarefied Gas Dynamics. 1992. Vancouver, GA1.

[4] Aristov V.V., Tcheremissine F.G. Direct numerical solution of the Boltzmann kinetic equations. Moscow. Computing Center of the Russian Academy of Sciences. 1992. 192p.

[5] Frezzotti A., Pavani R. Direct numerical solution of the Boltzmann equation on a parallel computer. Computers Fluids, 1993. V.22, N1, P.1-8.

[6] Aristov V.V., Tcheremissine F.G. Conservative splitting method for solving the Boltzmann equation. USSR Journal of Comp. Math. Math. Phys., 1980. V.20. N1. P.208-225.

[7] Karp A.H., Flatt H.P. Measuring parallel processor performance. Communications of the ACM, 1990, V.33, N.5. P.539-545.

A Multi-phase Gossip Procedure : Application to Matrices Factorization

Abdelhamid Benaini and David Laiymani
LIB, Université de Franche-Comté, 25030 Besançon Cedex, France
E-mail: [benaini,laiymani]@comte.univ-fcomte.fr

Abstract : The underlying algorithmic model for reconfigurable machines, is the *multi-phase model.* In this model, programs are divided into series of phases and each phase runs on the best suited physical topology. To show the interest of multi-phase algorithms, in terms of communication performances, we describe a multi-phase broadcast procedure and we generalize it to a gossip procedure. The performances of these operations are illustrated via two new multi-phase algorithms : QR factorization and Gauss-Jordan elimination which both require gossip procedures. Our experiments on a SuperNode machine confirm the superiority of these algorithms with respect to those presented in the literature.
Keywords : Multi-phase algorithms, Gossip, Broadcast, QR factorization, Gauss-Jordan elimination.

1 Introduction

For a parallel distributed memory machine, the communications are often a restrictive factor. Then, the performance of a parallel algorithm depends on how well its communication graph matchs the interconnection network of the target parallel machine. In this way, parallel systems with static interconnections require to adapt algorithms to the architecture. The conception of such algorithms is often difficult because there is no ideal topology for a set of algorithm and because of the intractable problem of mapping an algorithm onto a parallel system. The use of a router can remove the designing problems but in any case, the effects of the architecture limitation are an increase of the communication costs.
From an another point of view, reconfigurable machines in which the interconnection topology can be dynamically configured overcome this problem by allowing an adaptation of the architecture to the needs of a specific application. Such machines allow the development of variable topology programs. The underlying algorithmic model is the *multi-phase* model in which programs are designed so as to execute series of phases. Each phase runs on a topology which suits in the best way, the needs of data movements and the interconnection network is set before the beginning of each phase [1, 2].
To show how the *multi-phase* model provides an improvement in performances of parallel algorithms we present an efficient multi-phase gossip procedure. The use of this procedure is illustrated via new multi-phase algorithms for the QR factorization and for the Gauss-Jordan elimination. In fact the communication schemes of these algorithms show that is necessary to perform some gossip operations. Experimental tests have been performed on a T.Node machine, a transputer-based reconfigurable MIMD parallel system, under the C_NET programming environment.
Section 2 introduces the computation model and the multi-phase gossip procedure. The next two sections describe the multi-phase analysis of the QR factorization and of the Gauss-Jordan elimination. Finally we present some experimental tests which show better results for multi-

phase implementations than for static ones.

2 A multi-phase gossip procedure

In our study, each processor is assumed to have d bidirectionnal communication links and the communication protocol is a *point-to-point rendez-vous*. The time t_{com} needed to send a message of size L between two physically connected processors is $t_{com} = \alpha L + \beta$ where $\frac{1}{\alpha}$ is the throughput of the communication links and β is the start-up time. Moreover, each processor has independent units for communication and computation. It is therefore possible to perform in parallel on the same node data transfers on each link and arithmetic operations. With the multi-phase model, each phase runs on a particular topology. So, there is a reconfiguration of the interconnection network between two phases. It is shown that the reconfiguration cost t_{rec} only depends on the number of processors of the current phase and the number of processors of the next phase.
Now let a reconfigurable machine with a network of degree d and composed of p processors. We consider the problem of broadcasting m messages, each of size L, from m processors to all the other p processors. For $m = 1$ this is a *one-to-all* broadcast. For $m = p$ this is a *all-to-all* broadcast. These kinds of communications are frequently used and it would be interesting to study them in details.
A broadcast procedure on a reconfigurable machine for the particular case of $p = (d+1)^2$ and $m = 1$ is proposed in [3]. In this paper, we generalize it for any p, d and m. To underline the main ideas and results of this generalization, we illustrate our presentation with an example.

2.1 Procedure description

The procedure is articulated around three phases : a division of the messages in sub-blocks of size $\frac{L}{d+1}$, a broadcast of these sub-blocks and an aggregation of them. With this reconfigurable method the size of the communicated messages becomes $\frac{L}{d+1}$ instead of L. A theoretical and practical study of this procedure shows that, for long messages, the time gain is significant versus a static method. To help the reader to understand the following of this paper, we first present the procedure given in [3] which assumes that $p = (d+1)^2$ and $m = 1$.

Case $p = (d+1)^2$ and $m = 1$

In this case, the procedure is a broadcast operation. It consists in the transmission of a message originated from one node (root node) to all the other nodes of the network. The principles of the algorithm are as follows. In a first phase, the root node is connected to d nodes and it divides the message of size L into $d+1$ equal parts. The first part stays on the root and the d others are sent to the d remaining nodes. In the second phase every processor having received some data is connected to d other nodes and behaves as the root node did in the first phase. Then, there are $d+1$ groups of nodes, each holding a data of size $\frac{L}{d+1}$. If we take one node per groups we form $d+1$ new groups of fully connected processors and it allows each processor to get the whole message.
The figure 2.1 illustrates the different topologies of the procedure for $p = 25$ and $d = 4$.
During the three phases the size of the communicated messages is $\frac{L}{d+1}$. So the execution time is $T_{1\to p} = 3(t_{rec} + \beta) + \frac{3L}{d+1}\alpha$. On a d-ary tree the broadcasting algorithm has a communication time of $T_tree_{1\to p} = \lceil \log_d p \rceil (L\alpha + \beta)$.

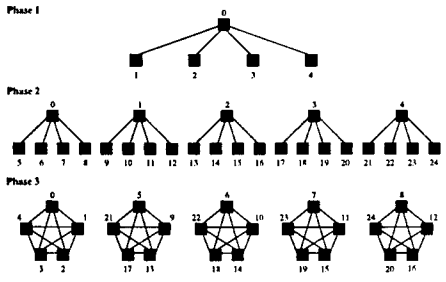


Figure 1: The broadcast procedure proposed in [2]

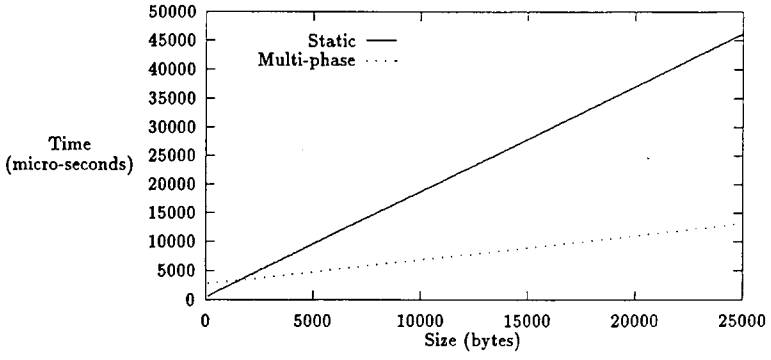


Figure 2: Static versus multi-phase algorithm for the broadcast procedure with $d = 3$ and $p = 16$

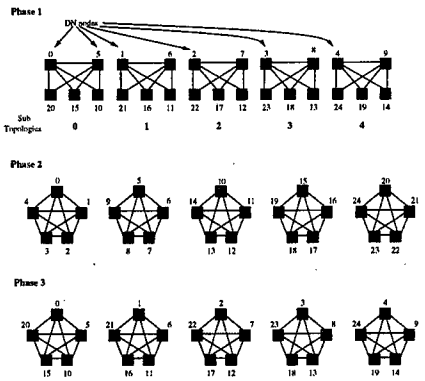


Figure 3: A multi-phase gossip procedure for $p = 25$, $m = 6$ and $d = 4$

The experimental results of the multi-phase broadcast operation with $d = 3$ and $p = 16$ are given in figure 2. After being less interesting than the static broadcast on a tree, the multi-phase algorithm becomes more efficient. In fact, for small messages the reconfiguration cost t_{rec} is too high to get good results.

Case $p = (d+1)^2$ and $1 \leq m \leq p$

Let L_i be the message that is sent from the processor P_i to all the other nodes. We divide L_i in sub-blocks of size $\frac{L}{d+1}$ called L_{ij} for $0 \leq i \leq m-1$ and $0 \leq j \leq d$. Futhermore, the m nodes $(P_0, P_1, \ldots P_{m-1})$ which have to send one message are called *distributing nodes* (DN).
For instance, let $p = 25$, $d = 4$ and $m = 6$ then, nodes P_i have to send L_i for $0 \leq i \leq m-1$, to all the other nodes. Figure 3 shows the different topologies for this example. In each phase there are $d+1$ sub-topologies each of $d+1$ nodes. Initially, P_0 holds $L_{00}, L_{01} \ldots L_{04}$, P_1 holds $L_{10}, L_{11} \ldots L_{14}$, ..., P_5 holds $L_{50}, L_{51} \ldots L_{54}$.
In phase 1, a sub-topology u contains the nodes $P_{k(d+1)+u}$ for $0 \leq k \leq d$. Each DN node sends its sub-blocks to all its neighbours. For instance, P_0 sends L_{01} to P_5, L_{02} to P_{10}, ..., L_{04} to P_{20}. The other DN nodes perform the same process. Also, at the end of this phase, let a sub-topology u which contains a distributing node P_x. Then, at the end of this phase, each node $P_{k(d+1)+u}$ of this sub-topology, holds L_{xk}. For example : $P_{10} = P_{2(5)+0}$ holds L_{02} and L_{52} and $P_6 = P_{1(5)+1}$ holds L_{11}.
In phase 2, we connect two nodes $P_{k_1(d+1)+u_1}$ and $P_{k_2(d+1)+u_2}$ iff $k_1 = k_2$. So, in this phase a sub-topology u contains the nodes $P_{u(d+1)+k}$ for $0 \leq k \leq d$. After an *all-to-all* procedure whithin each sub-topology, each node $P_{u(d+1)+k}$ holds L_{xu} where $0 \leq x \leq m-1$. For example : $P_0, P_1 \ldots P_4$ hold $L_{00}, L_{10} \ldots L_{50}$ $(u = 0)$ and $P_{10}, P_{11} \ldots P_{14}$ hold $L_{02}, L_{12} \ldots L_{52}$ $(u = 2)$.
In phase 3, we connect two nodes $P_{u_1(d+1)+k_1}$ and $P_{u_2(d+1)+k_2}$ iff $k_1 = k_2$. In this phase a sub-topology u contains the nodes $P_{k(d+1)+u}$ for $0 \leq k \leq d$. After an *all-to-all* procedure whithin each sub-topology, each node $P_{k(d+1)+u}$ holds L_{xy} with $0 \leq x \leq m-1$ and $0 \leq y \leq d$.
Let $r = \lceil \frac{m}{d+1} \rceil$. According to the selected model of communication, the communication cost t_i of a phase i is :

$$t_1 = t_{rec} + \frac{L}{d+1}\alpha + \beta,\ t_2 = t_{rec} + \frac{rL}{d+1}\alpha + \beta \text{ and } t_3 = t_{rec} + \frac{mL}{d+1}\alpha + \beta$$

Finally the total communication time when $p = (d+1)^2$ is $3(t_{rec} + \beta) + \frac{(m+r+1)L}{d+1}\alpha$.

Case $m < (d+1)^2 \leq p$

To broadcast m messages from m nodes to p nodes when $m \leq (d+1)^2 < p$, we iterate the multi-phase gossip procedure γ times. In fact, in a first step m nodes send a message to $(d+1)^2$ other nodes. These $(d+1)^2$ nodes are splitted into $\frac{(d+1)^2}{m}$ sub-sets with m nodes each, which run the multi-phase gossip procedure. So, $\frac{(d+1)^2(d+1)^2}{m}$ nodes are reached. And so on. In this way, $\gamma = \frac{\log(\frac{p}{m})}{\log(\frac{(d+1)^2}{m})}$.

3 Application to matrices factorization

3.1 QR factorization

The QR factorization is a common linear algebra computation, where an $m \times n$ matrix A, is expressed as the product of two matrices Q and R, with Q an orthonormal matrix ($Q^TQ = I$) and R an upper triangular one. For the sake of simplicity we assume that $m = n$. The modified Gram-Schmidt method allows a QR decomposition and the correponding sequential algorithm is as follows. Let a_j, r_j and q_j be the j-th column of respectively A, R and Q.

$d_j = ||a_j||^2, 1 \leq j \leq n$
For $k = 1$ to n do
    $d_p = min(d_i)_{k<i\leq n}$
    Swap(d_k,d_p) Swap(a_k,a_p)
    Swap(r_k,r_p)
    $r_{kk} = \sqrt{d_k}$
    $a_k = \frac{a_k}{r_{kk}}$
    For $j = k+1$ to n do
        $r_{kj} = r_{kj} + < a_k, a_j >$
        $d_j = d_j - r_{kj}.r_{kj}$
        $a_{ij} = a_{ij} - a_{ik}.r_{kj}$, $1 \leq i \leq n$
    Done
Done

The parallelization of the modified Gram-Schmidt method has provided some interesting results [4]. Jeesup [5] proposes an algorithm which uses a row decomposition, O'Leary and Whitman [6] present experimental results on the BBN Butterfly machine, Zapata et al [7] give an algorithm for Hypercube SIMD computers, finally Snyder and Lin [8] propose new algorithms which use row decomposition and particularly the Cached Rows algorithm. The communication scheme of this last algorithm is illustrated in figure 4 (see also [9]). This figure shows that in a first time, the algorithm requires a broadcasting of the vector d_j, and at each step $1 \leq k \leq n$, the values r_{kj}, $1 \leq j \leq n$ have to be send from one processor to all the others.
This algorithm can be implemented on a d-ary tree static topology. Nevertheless, we underline that $n+1$ broadcasts of messages of size n are neccessary. So, to perform them and according to the theoretical and practical valuations of the broadcast algorithm described above, it would be interesting to use the multi-phase strategy for $m = 1$.
Experimental tests presented at the end of this paper confirm this analysis.

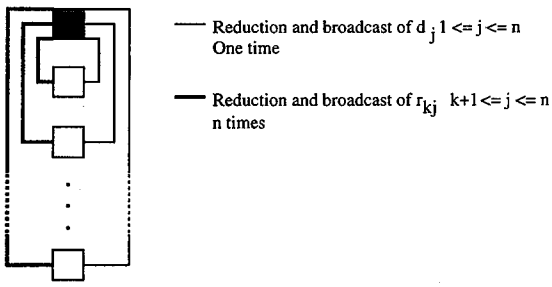


Figure 4: Communication scheme of the Cached Rows algorithm

3.2 Gauss-Jordan elimination

Let $B.w = c$ a linear system, where B, w and c have respective size $n \times n$, $n \times m$ and $n \times m$. We consider the matrix $\overline{B}$ augmented with c, that is we let $\overline{B} := (B, c)$ be a $n \times (n+m)$ matrix. Let $(b_{ij})_{1 \leq i \leq n, 1 \leq j \leq (n+m)}$ be the components of $\overline{B}$. Then, the sequential Gauss-Jordan algorithm is the following :

for $k = 1$ to $n - 1$ do
  < step k {Gaussian elimination }
  for $i = k + 1$ to n do
    $pivot := b_{ik}/b_{kk}$
    for $j = k$ to $(n + m)$ do
      $b_{ij} := b_{ij} - pivot.b_{kj}$
  { Jordan elimination }
  for $i = 1$ to k do
    $pivot = b_{ik}/b_{kk}$
    for $j = k$ to $(n + m)$ do
      $b_{ij} := b_{ij} - pivot.b_{kj}$ >

The Gaussian elimination and the Gauss-Jordan algorithm have been extensively studied in the literature [10]. To our knowledge, no multi-phase parallelization of this method is proposed except for tridiagonal matrices [11]. In order to refer a node, let a reconfigurable network of p^2 processors viewed as a square array. A processor located at (i, j), $1 \leq i, j \leq p$ is labeled P_{ij}. Now, the $\overline{B}$ matrix is naturally decomposed into blocks B_{ij}, $1 \leq i, j \leq p$ each of size $\frac{n}{p} \times \frac{n+m}{p}$. So $B_{ij} = (b_{(i-1)\frac{n}{p}+t,(j-1)\frac{n+m}{p}+t'})_{0 \leq t < \frac{n}{p}, 0 \leq t' < \frac{n+m}{p}}$ and a processor P_{ij} holds the block B_{ij}. We assume that the B matrix is entirely distributed among the processors P_{i1} for $1 \leq i \leq p$. According to the Gauss-Jordan algorithm described before, at step k we define the *pivot value* as b_{kk}, the *pivot line* as $(b_{kj})_{1 \leq j \leq n+m}$, the *pivot column* as $(b_{ik})_{1 \leq i \leq n}$ and the processor which holds the pivot value is called *pivot processor*. Let a step k for which P_{x1} is the pivot processor. Then the pivot column $(b_{ik})_{1 \leq i \leq n}$ is allocated to the processors P_{i1}, $1 \leq i \leq p$. The part of the pivot column holding by a processor P_{i1} is called *partial pivot column.* In the same way, the pivot line $(b_{kj})_{1 \leq j \leq n+m}$ is distributed among the processors P_{xj}, $1 \leq j \leq p$ and the part of the pivot line presents onto a processor P_{xj} is called *partial pivot line.* A such distribution involve the communication scheme shown in figure 5.
Now, let P_{x1} be the pivot processor. Recall that the interconnection network is of degree d. According to the figure 5, the different broadcast procedures could be achieved in two phases.

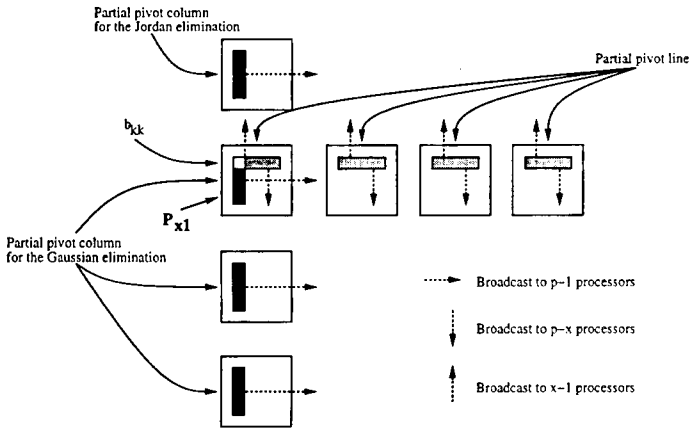


Figure 5: Communication scheme of the Gauss-Jordan elimination

First, we configure each row as a $(d-1)$-ary tree rooted at P_{i1} for $1 \leq i \leq p$ and second, we configure each column as a $(d-1)$-ary tree rooted at P_{xj} for $1 \leq j \leq p$.
This strategy seems to be costly, because $2n$ broadcasts are needed. So, in order to speed-up the algorithm we use, instead of $(d-1)$-ary tree static topologies, the multi-phase broadcast strategy described in section 2. According to the theoretical and practical valuations of this strategy this approach is relevant.

4 Experimental results

We have implemented the Cached Rows algorithm on T.Node machine (reconfigurable parallel system) with an interconnection network configured as a d-ary tree. The results of the Hypercube (static parallel system) implementation are given in [8]. Recall that the communication graph of the cached rows algorithm requires $n+1$ broadcasts. Then, regarding to an Hypercube implementation, a d-ary tree implementation is more advantageous because the communication graph is mapped onto a better topology. The speed-up results of the QR factorization with $p = 16$, $d = 4$ and $n = 256$ shown in figure 6, corroborates this prediction.
Nevertheless and according to section 2, a broadcast on a tree is less efficient than a multi-phase broadcast, especially for long messages. Figure 7 confirms this analysis. For great problem sizes, the use of the multi-phase broadcast procedure is relevant and allows a decrease of the communication time of the Cached Rows algorithm, with respect to a d-ary tree implementation.
The multi-phase Gauss-Jordan algorithm assumes p^2 processors viewed as a square array and requires broadcasts along each row and column of the array. So, the number of processors involved in each broadcast is p. That's why we don't present experimental tests for the Gauss-Jordan elimination. In fact our T.Node machine own $p^2 = 16$ processors, implying broadcasts on $p = 4$ processors which is not very significant. Nevertheless, the communication complexity of the multi-phase Gauss-Jordan elimination is $O(n)$ which is similar to the multi-phase QR factorization. So, for a multi-phase implementation, we can expect a significant gain in performances regarding to a static implementation.

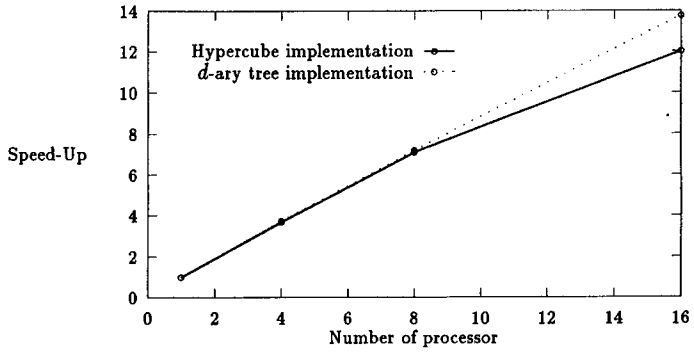


Figure 6: Hypercube versus d-ary tree implementation for the QR factorization with $p = 16$, $d = 4$ and $n = 256$

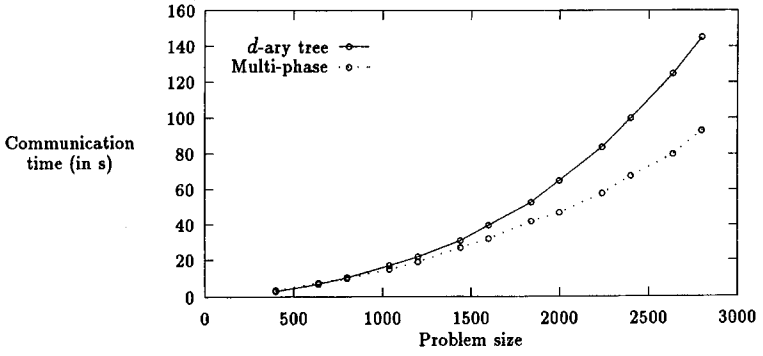


Figure 7: d-ary tree versus multi-phase communication time for the QR factorization with $p = 16$ and $d = 4$

5 Conclusion

In this paper, we have shown how a multi-phase gossip procedure improves the performance of some classical parallel numerical problems. Generally speaking, the simplicity of the multi-phase algorithms and their performances lead to a comfortable development and to an efficient execution.
We have also to note that reconfigurations of the interconnection network could be overlaped with computations. In this way, the reconfiguration cost which is sometimes penalizing, becomes non-existent for some applications.

Aknowledgements

We gratefully acknowlege C. Bonello for his precious help for programming with the C_NET parallel environment.

References

[1] J.M. Adamo, C. Bonello, and L. Trejo. The c_net programming environment : An overview. CONPAR 92, Lyon, September 1-4 1992.

[2] A. Benaini and D. Laiymani. Generalized wz factorization on a reconfigurable machine. *Journal of Parallel Algorithms and Applications*, 3(1):1–10, 1994.

[3] C. Bonello, F. Desprez, and B. Tourancheau. Parallel blas and blacs for numerical algorithms on a reconfigurable network. Rr, LIP-ENS Lyon, 1992.

[4] M. Cosnard, J.M. Muller, and Y. Robert. Parallel qr decomposition of rectangular matrices. *Numerishe Mathematik*, 48:239–249, 1986.

[5] E.R. Jessup. *Parallel Solution of the Symmetric Tridiagonal Eigen Problem.* PhD thesis, Yale University, Department of Computer Science, 1989.

[6] D.P. O'Leary and P. Whitman. Parallel qr factorization by householder and modified gram-schmidt algorithms. *Journal of Parallel Computing*, 16:99–112, 1990.

[7] E.L. Zapata, J.A. Lamas, F.F. Rivera, and O.G. Plata. Modified gram-schmidt qr factorization on hypercube simd computers. *Journal of Parallel and Distributed Computing*, 12:60–69, 1991.

[8] C. Lin and L. Snyder. An algorithm of choice for solving qr factorization. Rr, Department of Computer Science and Engineering - University of Washington, Seattle, 1992.

[9] A. Benaini and D. Laiymani. Parallel qr factorization on a reconfigurable machine. pages 317–327. 7-th Inter. Conf. on Computer Science, Tunis, 1994.

[10] Y. Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm.* Manchester University Press, 1990.

[11] J.C. Browne and R.N. Kapur. Techniques for solving block tridiagonal systems on reconfigurable array computer. *SIAM J. Sci. Stat. Comput.*, 5(3):701–719, 1984.

A substructuring method for solving the Image Restoration Problem on a multiprocessor system

I. Galligani*, E. Loli Piccolomini*, V. Ruggiero**, F. Zama*,
* Department of Mathematics, University of Bologna, Italy
e-mail: zama@dm.unibo.it
** Department of Mathematics, University of Modena, Italy

This paper is concerned with the development, on a multiprocessor system, of a substructuring method for solving the problem of image restoration when the degradation function has no specific form.

1 Statement of the problem

The image degradation process can be modeled as a system H which operates, with an additive noise term $\epsilon(x,y)$, on an input image $f(x,y)$ to produce a degraded image $g(x,y)$. The result is an equation of the following form:

$$g(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(x,\xi,y,\eta)f(\xi,\eta)d\xi d\eta + \epsilon(x,y) \qquad (1)$$

The problem of *image restoration* is the determination of the original object distribution f given the recorded image g, the degradation function h and the noise ϵ.

The image restoration problem is ill-conditioned. This ill-conditioned behavior can be demonstrated by means of the Riemann-Lebesgue lemma ([1]). A well known and highly regarded method for dealing with such problems is the *method of regularization* by Phillips and Tikhonov ([3]).

If we suppose that the functions f,g,h and ϵ are sampled uniformly over a finite rectangular network to form arrays of size $\mu \times \nu$ and use the "rectangular rule" for the approximation of the integral in (1), then the Tikhonov regularization method consists in solving the following minimization problem ([1], p. 149)

$$min_f J(f,\gamma) \qquad (2)$$

where

$$J(f,\gamma) = \|Hf - g\|^2 + \gamma\|Cf\|^2 \qquad (3)$$

The matrix C is the regularization operator (usually chosen equal to the discretization of the Laplace or biharmonic operator) and γ is the regularization parameter ($\gamma > 0$).

The solution of problem (2) satisfies the normal equations

$$(H^T H + \gamma C^T C) f = H^T g \tag{4}$$

Here, $A = H^T H + \gamma C^T C$ is a symmetric and positive definite matrix of order $n = \mu \times \nu$ and f and g are vectors of n components.

For "modest-sized" problems (e.g. $n < 1000$) an efficient and numerically stable method for solving this problem is based on the singular value decomposition theorem. When the degradation function is "space-invariant", the solution of (4) may be efficiently obtained on a parallel computer by using the fast Fourier transform (FFT) algorithm, which also requires a limited amount of storage.

In general, the dimension of system (4) is quite large. For a low resolution image of 512×512 pixels, the matrix A takes on a size $2^{18} \times 2^{18}$, which is difficult to operate upon in the computer; obviously storage alone is a monumental task.

For example, in many image restoration problems, the degradation function h has the following *general* form, i.e., h is neither "space-invariant" nor "separable":

$$h(x,\xi,y,\eta) = \frac{N}{2\pi\sigma(x,y)} exp\left(\frac{(\xi - x)^2}{2\sigma^2(x,y)}\right) exp\left(\frac{(\eta - y)^2}{2\sigma^2(x,y)}\right) \tag{5}$$

where

$$\sigma(x,y) = \begin{cases} |k sin(x+y)|, & \text{if } \sin(x+y) \not\equiv 0 \quad k \in [1,2] \\ k, & \text{if } \sin(x+y) = 0 \end{cases} \tag{6}$$

or

$$\sigma(x,y) = k(x+y); \quad k \in [0.01, 0.02] \tag{7}$$

where k is a positive number and N is a normalization constant for h. The magnitude of k will determine the extent of blur in the image, in the sense that for growing values of k, each image point is obtained with the contribution of a larger number of object points (*superposition* principle, [1]).

An interesting proposal for reducing the dimension of system (4) has been presented in [2]. However, for many images of practical interest, the dimension of (4) is still too large.

The advent of high performance parallel computers makes tractable the general case in which the degradation function has no specific form. Indeed, a natural approach to solve the problem (2) on these computing systems is based on the domain decomposition principle. The basic idea is to decompose the image-domain into image-subdomains and restore the image-patch in each image-subdomain separately in parallel. The approach may be different according to the parallel architecture we have at our disposal.

2 A substructuring technique for a shared memory multiprocessor system

On a shared memory multiprocessor system with a few high performance vector processors, such as the Cray Y-MP, we propose the following substructuring technique.

Let $\mathcal{R} = [1, \mu] \times [1, \nu]$ be the domain on which the function $f(x, y)$ is sampled in $\mu \times \nu$ regular grid points. We decompose R into τ disjoint rectangular subdomains R_t with interfaces parallel to the sides of $\mathcal{R}$. We *assume that an approximation of the function f(x,y) is known on these interfaces.* For example, an approximation to $f(x, y)$ on each boundary line may be obtained by solving an image restoration problem (1) in a strip centered on the boundary of two adjacent domains. These restoration problems may be solved efficiently in parallel.

With this assumption, the original image restoration problem on $\mathcal{R}$ may be reduced to τ image-patch restoration problems on the subdomains $R_t (t = 1, 2..., \tau)$ with the constraint that the function is continuous on the common boundaries. In each subdomain R_t this continuity condition is expressed by a system of linear equations

$$E^T f - s = 0 \tag{8}$$

where E is a sparse $n \times m$ matrix and s is a vector of m known values of the image $f(x, y)$ on the boundaries of R_t.

Then, in each subdomain R_t it is required to solve a medium-size minimization problem of the form (2) with the equality constraints (8) and the τ subproblems (2)-(8), related to the τ subdomains $R_1, R_2, ...R_\tau$, may be solved in parallel.

A suitable method for solving efficiently on a vector computer in each subdomain R_t the medium-size equality constrained quadratic programming problem (2)-(8) is the Hestenes method of multipliers. This method is defined by the following formulas ($c > 0$):

$$\lambda_{i+1} = \lambda_i + c(E^T f_i - s) \quad i = 0, 1, ... \tag{9}$$

where f_i is the solution of the system

$$(A + cEE^T) f = -E\lambda_i + r + cEs \tag{10}$$

and $r = H^T g$. For moderately large c, starting with $\lambda_0 = 0$, a few total iterations will usually be sufficient for the convergence of method (9)-(10).

At each iteration i the system (10) is solved with the Preconditioned Conjugate Gradient method (PCG) with a diagonal preconditioner. A Fortran program for a parallel implementation of the method on CRAY Y-MP (with four vector processors) with CMIC$ microtasking directives has been developed.

With $\tilde{f}(x, y)$ we indicate the restored image; the solution of (9)-(10) is the restriction on R_t of $\tilde{f}(x, y)$.

3 Computational experiments on CRAY Y–MP

Two images have been used as test problems for the domain decomposition method described in section 2: a CAT (63×109 pixels) and a VENUS (105×105 pixels), with 32 gray levels each. Three blurred CAT images ($k = 2$ in formula (6)(TP1); $k = 0.01$ and $k = 0.02$ in formula (7)(TP2 and TP3)) have been decomposed into four image–patches, each of 32×55 sample values. The values of the image on the boundaries of $\mathcal{R}$ and on the interfaces are determined by solving six restoration image problems (1) in six strips, of width varying from 14 to 24 pixels, centered on the four sides of $\mathcal{R}$ and on the two interfaces of the four subdomains; these interfaces are parallel to the coordinate axes.



Figure 1: Original CAT image



Figure 2: TP1 blurred image



Figure 3: Restored TP1 image

In our second restoration experiment, the input to the domain decomposition method is a severly blurred image ($k = 2$ in formula (6)) derived from the VENUS image. It has been decomposed into two image–patches of 105×53 sample values and four image–patches of 53×53 sample values (TP4 and TP5 respectively).



Figure 4: Original VENUS image

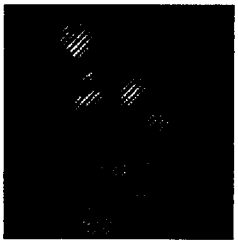


Figure 5: TP5 blurred image



Figure 6: Restored TP5 image

We can also compare the original image $f(x,y)$ and the restored image $\tilde{f}(x,y)$ by comparing the gray level in these images at the mesh–points of the network that covers $\mathcal{R}$. It is interesting to plot the *absolute error* $|f(x,y) - \tilde{f}(x,y)|$ and to give the percentage of the area of $\mathcal{R}$ with absolute error less or equal to the gray level l.

We have also calculated the three *picture distance measures* $D(f,\tilde{f})$,$R(f,\tilde{f})$ and $E(f,\tilde{f})$

given by Herman in [4]:

$$D = \sqrt{\frac{\sum_{u=1}^{\mu}\sum_{u=1}^{\nu}(f_{u,v} - \tilde{f}_{u,v})^2}{\sum_{u=1}^{\mu}\sum_{u=1}^{\nu}(f_{u,v} - \bar{t})^2}} \tag{11}$$

$$R = \frac{\sum_{u=1}^{\mu}\sum_{u=1}^{\nu}|f_{u,v} - \tilde{f}_{u,v}|}{\sum_{u=1}^{\mu}\sum_{u=1}^{\nu}|f_{u,v}|} \tag{12}$$

$$E = \max_{i,j} \; (|T_{i,j} - R_{i,j}|) \qquad \begin{matrix} 1 \le i \le \mu/2, \\ 1 \le j \le \nu/2 \end{matrix}$$

where

$$T_{i,j} = \frac{1}{4}(f_{2i,2j} + f_{2i+1,2j} + f_{2i,2j+1} + f_{2i+1,2j+1})$$

$$R_{i,j} = \frac{1}{4}(\tilde{f}_{2i,2j} + \tilde{f}_{2i+1,2j} + \tilde{f}_{2i,2j+1} + \tilde{f}_{2i+1,2j+1})$$

and $\bar{t}$ denote the average density of the digitized test image.

These three measures have been calculated also before restoration (D_0, R_0, E_0) in order to have a quantitative evaluation of the blurring effect on the original image.

Table 1:

Restored image	D_0	R_0	E_0	D	R	E	$Perc.of error < 1$	$and < 8$
TP1	1.63	0.83	38.1	0.41	0.18	28.8	28%	94%
TP2	1.57	0.79	29.0	0.39	0.15	22.0	24%	95%
TP3	1.34	0.66	28.6	0.57	0.25	26.6	12%	87%
TP4	1.13	0.27	27.6	0.74	0.18	12.9	76.3%	
TP5	1.17	0.28	27.2	0.74	0.18	12.9	77%	

Tables 1 and 2 summarize the results obtained in the experiments of restoring the CAT and VENUS images. In table 2, k^* indicates the number of "outer" iterations required to make $\|E^T f_i - s\| < 10^{-3}$ and j^* indicates the total number of CG iterations for the solution of system (10) with the CG method. Different values of k^* and j^* are obtained for the different subdomains R_t, $t = 1, ...\tau$; in table 2 we report maximum and minimum values of k^* and j^*. The termination criterion for the CG method is taken as $\|(-E\lambda_i + r + \sigma ES) - Mf\| < min\{10^{-3}, 10^{-8}\|M\|_T\}$ where $\|M\|_T$ is the Turing norm of M. For all these test problems the values of γ and c are 10^{-4} and 10^6 respectively.

Table 2:

Restored image	n	m	$k^*(max/min)$	$j^*(max/min)$
TP1: CAT(k=2)	2418 $(32+7)\times(55+7)$	170	2/1	624/430
TP2: CAT(k=0.01)	2730 $(32+10)\times(55+10)$	170	2/1	109/53
TP3: CAT(k=0.02)	2948 $(32+12)\times(55+12)$	170	2/1	148/98
TP4: VENUS(k=2)	6300 $105\times(53+7)$	312	2/2	824/561
TP5: VENUS(k=2)	3844 $(53+9)\times(53+9)$	208	2/2	757/468

The computer time on CRAY Y-MP with two vector processors for the complete restoration of the CAT image with the proposed substructuring method is 29.3 seconds in the case $k = 0.01$ and 67.0 seconds in the case $k = 0.02$.

4 A substructuring technique for a message passing distributed memory multiprocessor system

On a message passing distributed memory multiprocessor system we have many processors with a local memory. If we want to obtain a satisfactory performance, it is necessary to reduce the interprocessor communications as much as possible. Thus, we propose to decompose the image domain $\mathcal{R}$ into a *large* number τ of rectangular subdomains $\tilde{R}_t$ $(t = 1, 2, \ldots, \tau)$ with rectangular overlapping strips between adjacent subdomains. More precisely, the image $\mathcal{R}$ is subdivided into $\tau = \nu r \times \mu r$ disjoint rectangular subdomains R_t, On each subdomain we consider the problem (1). In order to preserve continuity of the solution across the interfaces of two adjacent subdomains, an overlapping strip is added to each subdomain side. Thus, the image-subdomain $\tilde{R}_t$ is obtained by enlarging each side of R_t with u pixels, so that each subdomain $\tilde{R}_t$ holds approximately the same number of pixels. In each subdomain $\tilde{R}_t$ we solve the medium-size minimization problem (2) or (4), where, the matrix $A = H^tH + \gamma C^tC$ has order $dimen = (\mu/\mu r + 2u)(\nu/\nu r + 2u)$.

In order to take advantage of the sparse structure of the matrix A, the solution is computed with the Preconditioned Conjugate Gradient method (PCG) with a diagonal preconditioner. A C language program has been developed to implement the Domain Decomposition algorithm on a transputer network constituted by two Quintek FAST9 boards plugged into a PC Intel 486 host computer. Each board is fitted with nine T805 transputers running at 25Mhz with 4Mbyte of local storage. The transputers may be

connected in many different topology networks and better results, in terms of communication times, are obtained using connections that match the communication paths of the algorithm. In our implementation, there are two communication phases: the *data decomposition* step, where each transputer reads part of the blurred image from a file and the *results collection* step, where each transputer writes its own results to a file. The communication phases have been accomplished by primitives of Arnia DD_lib library [7] as follows:

DD_read_domain_ol (NDIM,ldim,offset,sizeof(int), fname,gdim,edim,f,u);
...
... */* Subproblem Solution */*
...
DD_write_domain_ol (NDIM,ldim,offset,sizeof(int), fname,gdim,edim,f,u);

where the primitive *DD_read_domain_ol* distributes the data of file *fname* into local arrays f according to a *NDIM*–dimensional grid of processors with an overlapping area defined by u. The primitive *DD_write_domain_ol* implements the *results collection* process. These routines take the user specification domain (the blurred image) and perform a mapping to the underlying processor topology, so that the user has not to know the location of the processes or which nodes have to communicate. In this way it is possible to change the physical connection topology without modifying the code.

5 Computational experiments on transputer networks

Computer experiments have been carried out blurring some test images with the degradation function (5)–(6) or (5)–(7). We report the results obtained with the test problems

Table 3: Herman's parameters for the blurred test images.

TEST	k	D	R	E
TP6	1.8	0.52	0.29	62.5
TP7	2	0.73	0.3	54
	0.018	0.44	0.2	68

TP6, (a 117×103 radiographic image, figure 8) and TP7, (a 64×64 photographic image, figure 11), with 256 gray levels each. The quality of the restored and blurred images have been measured by the Herman's parameters (11). In table 3 are reported the parameters measured in the blurred test images. Table 4 reports errors and total computation time

Table 4: Image restorations errors and timings.

TEST	k	γ	u	D	R	E	τ	sec	*dimen*
TP6	1.8	$3.e-3$	5	0.26	0.18	37	16	437	34×37
				0.26	0.18	37	15	571	41×31
				0.26	0.18	37	12	654	41×37
TP7	2	$1.e-3$	6	0.51	0.25	65	16	164	25×25
				0.51	0.25	65	12	224	25×29
				0.51	0.25	65	8	334	25×38
	0.018	$8.e-4$	8	0.32	0.13	68	16	341	28×28
				0.31	0.13	66	15	374	26×32
				0.31	0.13	68	12	412	28×32

(elapsed time) obtained varying the number of subdomains. Comparing the values of column D, R and E in tables 3 and 4 it is possible to have a quantitative evaluation of the restoration process. Furthermore, observing the values of table 4, it is clear that errors are not affected by the increasing number of subdomains (column τ). Particular attention should be payed to the case $k = 0.018$. The maximum error E seems not to be reduced by the restoration process as much as the other error parameters (D,R). Besides, the percentage of absolute error less than 16 gray levels passes from 49% in the blurred image, to 73% in the restored image. This means that the highest errors concentrate in few points while most of the restored image has a better quality. Since the program hardly can run on a single processor (because of storage limitations), we evaluate the efficiency as the ratio $(T_{\tau^*}/\tau)/(T_\tau/\tau^*)$, where T_τ and T_{τ^*} represent the computation time with τ and τ^* processors respectively and $\tau^*(< \tau)$ is the minimum number of processors that can be used in the solution of a problem (in our case, $\tau^* = 6$). In Figure 7 the values of

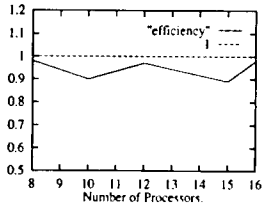


Figure 7: Efficiency values for test image TP7 ($k = 2$)

efficiency obtained with test problem TP7 ($k = 2$) are plotted. Finally, table 5 reports the

Table 5: Communication time for test problem TP7 case $k = 2$.

τ	Comm. Time	Total Time	%
6	4.7	433	1.%
8	5.0	334	1.5%
10	6.3	289	2.1%
12	4.9	224	2.1%
15	6.0	193	3.0%
16	4.9	164	2.9%

time spent in the communication phases and total computation time when the number of subdomains increases. Though the communication time increases with the number of subdomains, it is always a small percentage of the total computation time.

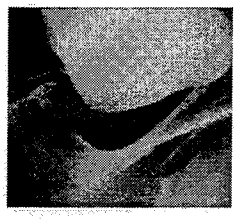


Figure 8: 117 × 103 Original image (TP6)

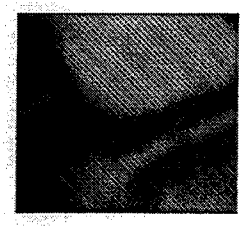


Figure 9: TP6 image (blurred with k=1.8)

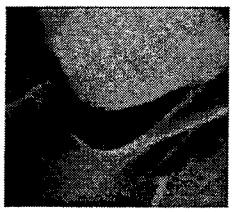


Figure 10: Restored Image



Figure 11: 647 × 643 Original image (TP7)

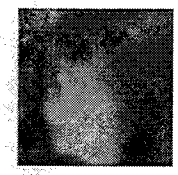


Figure 12: Blurred image (k=0.18)



Figure 13: Restored Image

References

[1] H. C. Andrews, B. R. Hunt, Digital Image Restoration Prentice Hall, INC., Englewood Press, New Jersey.

[2] H. S. Hou, H. C. Andrews, Least Square Image Restoration Using Spline basis Functions, IEEE Transaction on computers, vol. C-26, NO. 9 September 1977

[3] A.N. Tikhonov, V.Y. Arsenin, Solutions of Ill–Posed Problems, John Wiley & Sons (1977) John Wiley & Sons, Inc., New York (1981).

[4] Gabor T. Herman, Image Reconstruction from Projections, Academic Press, (1980).

[5] M.G. Cox, G.T. Anthony, The fitting of extremely large data sets by bivariate splines, Algorithms for Approximation (J.C. Mason, M.G. Cox eds.), Clarendon Press, Oxford(1987).

[6] I. Galligani, V. Ruggiero, Numerical Solution of Equality–Constrained Quadratic Programming Problems on Vector–Parallel Computers, Optimization Methods and Software, vol. 2 (1993), 233–247.

[7] Arnia 1.0, Advanced Computing Systems, Milano (1991)

Selected SMS TPE'93 Papers

Specification and Compilation of Distributed Algorithms for the ArMen Machine *

Philippe Dhaussy, Stéphane Rubini
LIBr-Télécom Bretagne, Université de bretagne occidentale
BP. 832, Kernevent Plouzané, 29285 Brest
e-mail : dhaussy@enstb.enst-bretagne.fr
rubini@ubolib.cicb.fr

February 12, 1993

Abstract

The execution of an application on distributed memory multiprocessors machines requires a superposition of computation tasks and control tasks. This is harmful if machines do not have a specific support for control services. The ArMen architecture, composed of a processor network and a shared programmable logic layer, have such a support.

We propose to use the formalism UNITY to express algorithms as a union of a program *user* and a program *control*. The program *user* is compiled for a MIMD processor network and the program *control* is transformed into boolean expressions which can be mapped on the reconfigurable logic layer. This layer constitutes an integrated and autonomous support of computation and communication, assigned to the control of distributed parallel programs.

This strategy is illustrated for the compilation of a mutual exclusion algorithm.

Introduction

On the distributed memory multiprocessor machines, the cooperation of a processus set requires control services to ensure system coherence. If a machine do not have a specific support, computation and control operations use the same computation and communication resources alternately. This sharing reduces the output and harms to the clarity of programs. Furthermore, the impossibility of observing instantaneously a global state of the processor network often implies a total end to computation during the global control phases.

To consider these problems, several machines offer communication and processing supports dedicated to the control tasks. A few machines have two processors per node. One assumes the production tasks, the other is assigned to controls and communication. The ArMen machine

*This work is supported by the GDR/PRC ANM and C^3, the Bretagne region and the Brest city. The ArMen machine development has been mainly supported by ANVAR

provides an alternative. Its configurable logic layer can be used to implante centralized control and global processor models [10].

On the other hand, the parallelism and often non-determinism make programming these machines difficult. The state of global behaviour through local interactions and local behaviour is a delicate problem for programmers. An informal specification is then insufficient to ensure programs accuracy and reliability.

Thus, several formalisms, like CCS, UNITY or GAMMA, have been proposed to develop parallel algorithms. They bear mechanisms to prove the accuracy of algorithms using mathematical arguments. Furthermore, these formalisms allow deduce local behaviour to be deduced from global specifications of systems.

In this article, we will describe a parallel program with UNITY formalism, as a union of a program *user* and a program *control*. We apply this transformation to compile programs on the ArMen machine.

First, we explain the ArMen architecture and its caracteristics. Next, we show how a UNITY description is compiled to produce on the one hand executable programs on a distributed memory multiprocessor machine, and on the other hand implantable logic circuits on a reconfigurable logic layer. Then, we end with a complete example applied to a mutual exclusion algorithm.

1 The ArMen machine

The architecture of the ArMen machine [15] is based on a superposition of a processor network and logic programmable circuits. Current implementation involves modules with a T800 transputer, a Xilinx 3090 Logic Cell Array (LCA) and a 1 Mbyte static memory (figure 1).

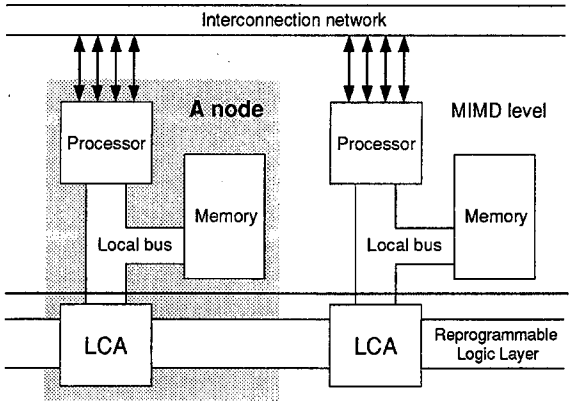


Figure 1: ArMen Architecture

Each configurable logic circuit is connected on the one hand with the local bus of the associated processor, and on the other hand with the two neighbouring circuits to form a linear

ring throughout the machine. The topology of the transputer link interconnection network is not imposed.

One LCA is composed of a logic block matrix (boolean functions on five variables) and of programmable interconnection resources. Each node is in charge of the local LCA configuration. This stage lasts 150 ms.

The ideas of a reconfigurable logic layer may be easily applied to other parallel machines. Furthermore, the reconfigurable logic circuits will benefit fully from integration technical progress.

2 Characteristics and requirements of the ArMen machine

The ArMen machine with its reconfiguration possibilities constitutes an efficient support which is adapted to several programming and execution models.

On most existing machines, the material architecture is fixed. Possible configuration capabilities are limited to the level of networks and communication modes. The machine supernode proposes, for example, a configurable interconnection *crossbar* network. Communications on a iWarp processors network may be synchronous or asynchronous.

But, the flow control and the execution model remains fixed. Moreover, machine characterization is often carried out in proportion to these features by Flynn's classification [11]. The operations are stepped by local clocks on each node in the case of MIMD machines, or by a global clock common to the processor set in the case of SIMD or systolic machines. Thus, the software of these machines is designed and compiled in relation with a determined architecture, according to imposed execution schemes.

On the other hand, the ArMen architecture has more avanced configuration capabilities because the unicity of the execution model is not implied. An application may be executed according to several models alternately or concurrently. An application description is composed during that time of software and associated hardware for the configurable layer, to define fonctions and the choice of execution models.

- MIMD model : the machine processor network naturally controls this fonctionning mode. Communication is carried out in the form of message passing. Each processor may make use of independent local accelerators implanted on the reconfigurable layer. This fonctionning mode is adapted to irregular programs.

- SPMD model : each node executes the same program and synchronizes itself with its neigthbours via access to a computation and/or control operator shared by all nodes. Communications may be synchronous at the level of the reconfigurable layer. This model is of interest particulary in order to employ the machine in a *data flow* context [1]. Futhermore, shared global operators can be used to implant massively parallel circuits [3].

- Systolic circuits : regular networks of simple processors can be synthesized in the reconfigurable layer. They have a proper sequencing independant of thw MIMD processor network, and they communicate synchronously. Data is supplied by the MIMD processor network.

- Global processors : These circuits are composed of regular pipelines of operators throughout the machine. A particular node sequences pipelines independently of the MIMD network. This architecture constitutes an autonomous integrated machine, able to apprehend efficently global states of the MIMD machine and to undertake distributed regular computation.

Thus, the versatile nature of the ArMen machine implies a particular need of tools, intended to describe the circuit structure to implant in the reconfigurable logic layer. This description must be written in a high level language.

With this end in view, two approaches are necessary :

1. This first one is to propose predefined execution and computation schemes, which potentially have a interest for many applications. Description languages enable high level logic synthesis according to architectural models easy to define. For that, we have developed :
   - The language CCEL allows global operators to be synthesized using a cellular automata model. These cicuits are used in SPMD mode to do regular computations on a large data array.
   - Another model will be defined to describe global processors in the form of a set of distributed finite state automata communicating through operators. They send and receive data from processor local interfaces.
2. The second approach is to use a unified language allowing global behaviour of a system to be specified. The aim is to provide a general tool to test and use various parallel algorithms. With this end in view, we propose to use UNITY formalim to describe essentially the control on the ArMen machine.

3 A unified approach of description

3.1 Generalities

UNITY (Unbounded Non deterministic Iterative Transformation) [7] is a description formalism of concurrent program. Its purpose is to provide a support in order to reason and develop, appropriate for a wide variety of architectures (shared and distributed memory architectures, systolic arrays, ...) and problems (control, reactive program, sorting program ...).

A UNITY program is characterized by :

- a set of simple or multiple assignments,
- non-determinism and the absence of control flow. There is no hypothesis about the assignment execution order. Non-determinism selection is only constrained by the following *fairness* rule: every statement is executed infinitely often in a infinite time interval.
- the possibility of describing asynchronous or synchronous behaviour with the notion of multiple assignments.

UNITY gives a high level programming style in order to describe the solution, independently of the target architecture. This is the design task. Next, this first program is derived and refined successively to obtain an adapted version for the target architecture. This is the mapping task. Temporal logic associated with the language, allows proof to be built on the transformation. Furthermore, a UNITY program can be structured as a union or a superposition of a set of components. With a few rules, global properties can be deducted from component's properties. This abstraction mechanism allows a parallel algorithm to be describe as a union or a superposition of an operative component (program *user*) and a control component (program *control*).

3.2 UNITY program structure

Here, we present in brief the UNITY formalism syntax. For a full description, the reader may referred to [7].

The general structure of a UNITY program is :

Program < program name >
Declare < variable declarations >
Always < macro definitions >
Initially < initial values >
Assign < set of assignments >
End

1. The section **declare** holds variable declarations and their associated type.
2. The section **Always** is composed of a equational set, allowing transparent variables or macro definitions to be defined according to other variables.
3. The section **Initially** defines initial values of variables.
4. The section **Assign** holds a non empty set of multiple or simple assignments separated by the symbol []. This non deterministic choice of evaluated assignments at a given time allows asynchronous execution models to be translated.

   A multiple assignment is a synchronous evaluation of values of a group of variables (separated by the symbol ||).

   Assigments may be guarded (clause if) or quantified.

Examples

- Guarded assignments :
  $x, y := x + 1, x$ if b
  is identical to :
  $(x := x + 1 \parallel y := x)$ if b
  and signifies that x takes the value of x+1 and y takes the value of x. If, initially, x is 1, x takes the value 2 and y the value 1. *simultaneously* if the condition b is true.

- Quantified assignments :
  $\langle \| \ i : 0 \leq i \leq N :: A[i] := 0 \rangle$
  Simultaneous reset of an array $A[0..N]$

- A sort program :

Program sort
  Declare A:array[0..N] of integer
  Assign
    $\langle [] i : 0 \leq i < N :: A[i], A[i+1] := A[i+1], A[i] \text{ if } A[i] > A[i+1] \rangle$
End

The variables $A[i]$ and $A[i+1]$ are exchanged if the guard $A[i] < A[i+1]$ is true. When all guards are false, the program reaches a *fixed point* corresponding to a sorted state of array.

3.3 A unified description for ArMen

We use the UNITY formalism to specify applications and especially their control on the ArMen machine. The absence of flow control and the ability to describe synchronous and asynchronous models, constitute a characteristic adapted for this architecture.

First, a complete description of the application, without distinction between software and hardware aspects, serves as a raisonning support. This description is then refined to obtain, on the one hand, an implantable program on the processor networks, and on the other hand a synthesizable circuit on the reconfigurable layer.

A interesting transformation strategy consistes in separating the control part and the operative part of the application. The reconfigurable layer then gives a specific and efficient support for the control of distributed parallel programs.

Example : This example presents a program to model a synchronizer. Its purpose is to produce pulses at each site of the asynchronous network as if the latter had been generated by a global clock. Each site has a instantaneous value of this clock [8].

The program *user* positions local variables l_sure_i , which represent the state suree of each process of the processor network. It manages local variables $puls_i$ which are incremented when the global condition *system_sure* is true.

The program control scans the l_sure_i states on each process and set global condition *system_sure* when all sites are surees.

Program *synchro* : *user*
Initially $\langle [] i : 0 \leq i \leq N-1 :: l_sure_i, puls_i = false, 0 \rangle$
Assign $\langle [] i : 0 \leq i \leq N-1 :: l_sure_i := true \ if \ application_condition \rangle$
  [] $\langle \| \ i : 0 \leq i \leq N-1 :: puls_i, l_sure_i := puls_i + 1, false \ if \ system_sure \rangle$
End

Program *synchro* : *control*
Assign $system_sure := \langle \wedge i : 0 \leq i \leq N-1 :: l_sure_i \rangle$
End

application = (synchro:user[]synchro:control)

This program cannot be implanted directly on the processor network and on the logic circuits because of the global variable *system_sure*. This program must be transformed into another program where this variable will be distributed on all sites. These derivations are at present the programmers responsibility.

4 Compilation for the ArMen machine

For the distribution of UNITY programs, the distribution of the data is used on the processors network of the ArMen machine.

Each variable of the program is located on the single node. With annotations, the programmer says where he wishes each variable to be.

A program P is distributed on the ArMen network of nodes S_i with generating programs P_i .

Programs P_i come from program P in which the affectations of only the left member variable of the S_i node are kept.

It is the local writing principle [16]. It provides communications between processes. A process reads remote data to execute a statement which modifies one of its variables.

The code of each process is particularized with its local variables. Also, each variable can be located in the processor of a node, or in the logic cell.

So, the communication strategy of the modified values of variables during the distributed program execution is different in both cases.

In this paper, we use the word node to name the processor or the logic cell of the node. Implementation in the logic layer is made by a logical synthesis of circuits with the XILINX development tools.

4.1 The compilation tools

The compilation tools which are under development, allow a executable code for the T800 processor and a LCA configuration to be generated for each node.

Figure 4.1 presents the current process of UNITY compilation for ArMen.

Compilation tools are created with the transformation tools of a UNITY program.

The *UC* compiler transforms a UNITY program and distributes it on the network by generating files which hold the intermediate codes corresponding to the program parts which are implemented on each logic cell and processor.

A *UC* compilation option allows the C program of a simulation of the specified application to be generated on a sequential machine.

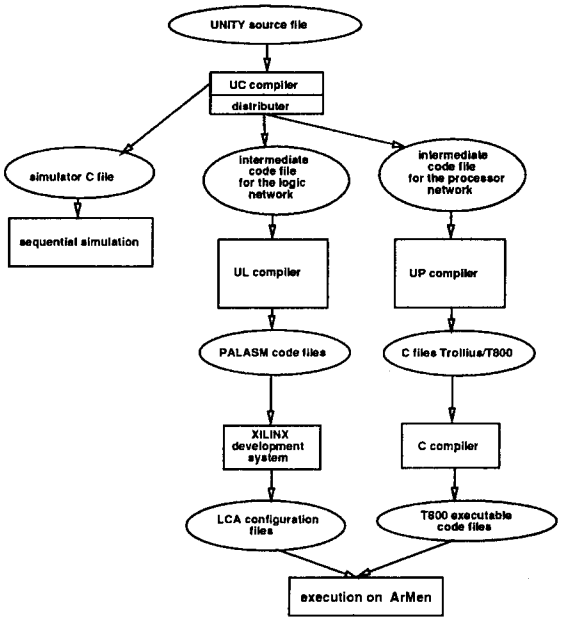


Figure 2: The compilation tools

The UL compiler transforms the intermediate code corresponding to a logic circuit into boolean equations.

Then, the XILINX tools transform the PALASM code into a configuration for each LCA.

The UP compiler transforms the intermediate code corresponding to a processor into a C program for the T800 processor of the Armen machine.

Currently, the implementation of the program in the processor network is made easier by the Trollius system [2].

This system installs a communication layer between the nodes by unifying the host processor with the others.

4.2 Data distribution

The data manipulated by the programmer are boolean arrays ou integer arrays. This kind of data is easily implementable. Firstly, we consider static data.

Data partition : Each variable has a single owner who is authorized to modify it. A scalar array is distributed on all the nodes. In the *declare* section of the UNITY program, the programmer indicates the type of desired distribution (modulo distribution, aliasing) with

the keyword *block*. This data distribution is inspired by the IRISA project for the Pandore language compilation for distributed memory computers [16].

Example :

declare

$$state : array\ [100]\ of\ integer\ block\ (4)$$

The array of integers *state*, with one dimension, contains 100 elements and is divided into four integer blocks. The integers of the same block are implemented in the same node. The blocks are implemented on the nodes modulo the number of nodes.

With the declaration *block* (k), and with NBS nodes, the number of the i^{th} element of a one-dimension array is given by :

$$i\ div\ k\ modulo\ NBS$$

The computation rule for the node number can be extended for the two dimension arrays. The distribution depends on the programmer choice in order to privilege a dimension to be distributed.

For example, the computation model implementation for cellular automata [13] on the ArMen machine requires a distribution of the array on the nodes by column.

The implementation of a variable on the processor or the logic circuit of a node is managed by the compiler and depends on the occurrence in the *user* code or *control* code.

4.3 Computation model

For reasons of simplification, the process is identified with the processor on which it is executed. Each P_i code on the S_i node contains a copy of the original P code.

P_i only executes the left side assignments which have a variable of the S_i node. Before the multiple assignment execution, S_i must have received a copy of the left side variables which it does not own. The communication channels are considered reliable and FIFO.

To respect the semantics of the UNITY program, all the left side variables of an assignment are owned by one node and are to be allocated to the processor or the logic circuit. Bellow, we describe the compilation for the processor and logic circuit network.

4.3.1 The processor network compilation

This part is not very detailed because it is still under development. The computation model is based on the asynchonous node communication, with queues implemented on each node.

The original program distribution P generates an intermediate program which is executed on each processor. This program contains the local variable assignments and the send and receive statements for the variables between the nodes.

Then, it is transformed by the UP compiler to a C code which is interfaced with the system communication functions. The computation model on a S_i processor comprises several steps :

- **step 0:** Initialisation of the local variables for each processor. For all the v left side variables (which are owned by S_i) of the assignments, sending out of v to all the processors which need it. Synchronization between all the nodes.

- **step 1:** Reading of the queue values and refreshing of the variables.
- **step 2:** Choice and execution of a multiple assignment which has a true guard.
- **step 3:** For all the v left side variables of the executed multiple assignment, sending out of v to all the processors which require it. Return to step 1.

The optimization of this compilation model concerns the deletion of the redundant updating of the copies of the variables which are not owned by the node or do not generate a communication operation.

4.3.2 The logic network compilation

The compilation model for the logic layer is based on the reading of the state variables on the neighbouring node i.e. a node which uses a variable reads it on the neighbouring node owning it.

For this, the program copy, P_i, of the original program holds only assignments which have left side variables owned by the S_i node and have right side variables owned by the S_i node or the neighbouring nodes, S_{i-1} or S_{i+1}.

The original program distribution, P, generates an intermediate program which runs on each logic circuit.

This program holds the local variable assignments with a functional form.

Then, it is transformed by the UL compiler into boolean equations.

Subsequently, we explain the assignment transformations which hold only boolean values.

Next, we study the case of integer values. Then, we approach the question of the transformation of arithmetical and relational operators.

The boolean variables : Notation : $Comp(n)$ means the result of compilation of n assignments. The assignment with the boolean variables, V, A and C

$$V := A \; if \; C$$

is transformed into the expression $Comp(1)$:

$$Comp(1) \vdash V := (\neg C \wedge V) \vee (C \wedge A)$$

If the variable, V, is in several assignments of the program, P_i, for example :

$$\begin{array}{l} V := A_1 \; if \; C_1 \\ [] \\ V := A_2 \; if \; C_2 \end{array}$$

The two assignments are transformed into :

$$Comp(2) \vdash V := (\neg C_2 \wedge \neg C_1 \wedge V) \;\; \vee \;\; (C_1 \wedge A_1 \;\vee\; C_2 \wedge A_2)$$

For the UNITY program to have a determistic behaviour, we have to be sure the C_1 and C_2 conditions are exclusive when A_1 and A_2 are different. Static control has to undertaken at compilation time.

The transformation of n occurences of the left side variable, V, in n assignments can be generalized.

$$Comp(n) \vdash V \ := \ ((\bigwedge_{i=1...n} \neg C_i) \wedge V) \ \vee \ \bigvee_{i=1...n} (C_i \wedge A_i)$$

The semantics of the UNITY program is respected if the assignment of two variables on the left side of the same assignment are simultaneous.

The integer variables : The assignments with integer variables are transformed into an assignment system of dimension equal to the width of the assigned variable. The programmer has to give the type of variable and the size.

The assignment, $V := A \ if \ C$, with V and A which are integers of k bit size, is transformed into an assignment system :

$$\forall i \in [0 \,..\, k-1], V_i \ := \ A_i \ if \ C$$

with V_i and A_i, variables associated with the i^{th} bit value of the variables, V et A, respectively. Now, this equation set can be transformed as in the boolean variable case.

Now, we have to specify the compilation of the guard , C which will be explained in the next paragraph with the relational operator study.

The arithmetical operators : Here, we do the transformations of expressions which hold only the arithmetical operators, + and -.

The compilation of the expressions which hold arithmetical operators, * and /, is considered currently, because of the logic circuit capacity, only for the integer variables of small size. The transformation of these expressions depends on the format chosen to implement the integers in the logic circuits.

The relational operators : In the guard expressions, the relational operators are $=, <>$ $, <, >, <=, >=$. As an example, the conditional expression :

$$A <> B$$

with the integers, A et B, of k bit width, is tranformed into :

$$\bigvee_{i=0...k-1} (\neg A_i \wedge B_i \ \vee \ A_i \wedge \neg B_i)$$

with the boolean variable A_i (resp. B_i), associated at the i^{th} bit value of A (resp. B). The transformation of conditional expressions holding the operators, $<, >, <=, >=$, depends on the format chosen for the integers into the logic circuit.

5 Synthesis of circuits

The equation generated during UL compilation time, is used to produce configuration files for the reconfigurable logic layer. The main steps are the following :

- minimization and distribution. Basic equations are minimized in resource terms. Next, they are distributed in the logic circuit blocks.
- assembling with interface components. These components are fixed and, in practice, they interface the reconfigurable layer and the other components of the node.
- location and routing. The logic blocks are located and interconnected through the physical structure of LCAs.

This synthesis phase lasts at present from 2 to 3 hours. A configuration file is obtained for each machine node and is loaded locally during program execution.

6 A example : Mutual exclusion

One of the important contributions with the ArMen machine with regards to standard MIMD machines is the control possibilities of the parallel programs. The programmable logic layer provides a flexible and specific support to implement control acitivities. So, the computation and control resources are distinct and communicate from time to time.

This section describes an implementation of the mutual-exclusion algorithm. The implemented algorithm is here Dijkstra's [5].

This algorithm presents the advantage of handling only a small number of variables. But it does not have the FIFO property (*i.e.* the requests are processed in order of their occurrence) unlike the Lamport algorithm [12], using more variables.

This example shows the way in which an application which has an application and control part can be structured. The strategy consists in implementing the application part into the processor network and the control part into the logic layer.

The application which uses the mutual-exclusion service is composed of NBS nodes or processes. The state of each process, S_i, is represented by a local variable, $access[i]$.

This variable can take three values : HUNGRY (the process requests the critical resource), EAT (the process has the resource), or THINK (the process releases the resource).

Here, we express the formal specifications of the application, then the model of these specifications in a UNITY program. In the appendix, the reader can find the results of the compilation of the control part for a LCA circuit of the logic layer, and also the diagram of the logic circuit synthesized into a LCA circuit.

6.1 The specifications

The specifications of the mutual-exclusion program are expressed with the following properties :

- The safety property indicates that only one process has the resource :

$$\forall i, j \in [0 \,..\, NBS-1] \neg(access_i = EAT \wedge access_j = EAT \wedge i \neq j)$$

- The progess property ensures not-deadlock and fairness : This property is based on the follow hypothesis : At the application level, a process which has the critical resource releases it eventually. It is expressed by :

$$(\forall i \in [0 \,..\, NBS-1] \quad access_i = EAT \longmapsto access_i = THINK) \Rightarrow$$

$$(\forall i \in [0 \,..\, NBS-1], \quad access_i = HUNGRY \longmapsto access_i = EAT)$$

With two predicates, p and q, $p \longmapsto q$ (p *leads* $-$ *to* q) asserts that once p becomes true, q is or will be true eventually.

6.2 The model

Here, we express a UNITY program which models the specifications of the application. The program is broken down into two parts : *user* and *control*. The *user* part corresponds to the application part of the program, the *control* part with the control which provides the mutual-exclusion service.

The variables, *eat* and *flag*, are implemented in the logic layer because they are in the program, *control*. The variable, *access*, is in the program, *user* et *control*, so they are implemented in the processor-logic circuit interface.

We specify the two bit format of the integers of the distributed array, *acces*. Here, the arrays are distributed in an elementary way : one element by node.

Program *mutex* : *user*
Declare

NBS	: *constant* 8
$THINK$	: *constant* 1
$HUNGRY$	: *constant* 2
EAT	: *constant* 3
access	: $array[NBS]$ *of integer* 2 *block*(1)
eat	: $array[NBS]$ *of boolean* *block*(1)
flag	: $array[NBS]$ *of boolean* *block*(1)

Initially

$\langle[]i : 0 \leq i \leq NBS-1 :: \quad access[i] = THINK\rangle$

Assign

$\langle[]i : 0 \leq i \leq NBS-1 :: \quad access[i] := HUNGRY \;\; if \; access[i] = THINK \;\rangle$

[] $\langle[]i : 0 \leq i \leq NBS-1 :: \quad access[i] := THINK \;\; if \; access[i] = EAT \;\rangle$

End

Program *mutex* : *control*
Initially

$\langle []i : \ 0 \leq i \leq NBS - 1 :: \quad flag[i], eat[i] = false, false \rangle$

Assign

—— Authorizes the resource ———

[] $access[0], eat[0] := EAT, true$
$\quad if \ access[0] = HUNGRY \wedge flag[0] = flag[NBS - 1]$

[] $\langle []i : \ 1 \leq i \leq NBS - 1 :: \quad access[i], eat[i] := EAT, true$
$\quad if \ access[i] = HUNGRY \wedge \neg(flag[i] = flag[i - 1]) \rangle$

——- Releases the resource and transmits the privilege ——-

[] $eat[0], flag[0] := false, \neg flag[0]$
$\quad if \ access[0] = THINK \wedge eat[0]$

[] $\langle []i : \ 1 \leq i \leq NBS - 1 :: \quad eat[i], flag[i] := false, flag[i - 1]$
$\quad if \ acccess[i] = THINK \wedge eat[i] \rangle$

—— Transmits the privilege ——

[] $flag[0] := \neg flag[0]$
$\quad if \ access[0] = THINK \wedge \neg eat[0] \wedge flag[0] = flag[NBS - 1]$

[] $\langle []i : \ 1 \leq i \leq NBS - 1 :: \quad flag[i] := flag[i - 1]$
$\quad if \ access[i] = THINK \wedge \neg eat[i] \wedge \neg(flag[i] = flag[i - 1]) \rangle$

End

mutex = (mutex:user []mutex:control)

The *flag* array implements a wave which distributes the privilege on the row of nodes.

6.3 Circuit generation

The compilation tools take this program at the input and generate the boolean equations corresponding to the configuration of each logic circuit.

In the appendix, the reader can find for example, the boolean equations generated for the node 0.

The semantics of the UNITY multiple assignments is respected because a global time is distributed over the set of LCA registers. So the left side variables of a multiple assignment are modified synchronously. We admit that the maximal propagation time in the combinational logic section is lower than the system clock period.

6.4 Performance

The synthesized circuit for this mutual-exclusion algorithm take 5% of the available logic resources. Other computation or control operators can be implemented in the programmable logic layer

The wave propagation time (variable *flag*) in the programmable logic layer is 75 ns per node for the 20 Mhz clock. If the shared resource is free, a request will be processed in a 1.2 μs in the worst case for a 16 node machine.

The processor network exploits the control circuit with the simple read-write access to one memory address. Also, the mutual-exclusion control does not involve overheads on the computation and communication resources of the processors.

Conclusion

We have shown how parallel algorithms may be expressed with the UNITY formalism to be compiled on ArMen architecture. So, such architecture proposes an adapted support for the control of distributed programs.

Research is in progress, for example, on how to manage the interleaving of MIMD-SPMD phases, how to implant dynamic routers or to control industrial systems.

Our present work consists in finishing and expanding the proposed UNITY compiler. It must allow the use of specific abilities of control, because the reconfigurable layer may act on the processor network by interruption or by physical locking.

Therefore, we shall have a tool to test and use varied control algorithms on the ArMen machine, synthesized through a formal specification.

References

[1] L. Bougé, JL. Levaire. Control structures for data-parallel SIMD languages : semantics and implementation March 17,1992.

[2] G. Burns, V. Radiya, Raja Daoud, et Raghu Machiraju. All about trollius. *Occam User Group Newsletter*, pages 55–70, July 1990.

[3] K. Bouazza, J. Champeau, P. Ng, B. Pottier et S. Rubini Experiment with Cellular Automata on ArMen. *Workshop "Parallel Architectures and Algoritms for VLSI II", Bonas, June 91.*

[4] K. Chandy et L. Lamport. Distributed snapshots : Determining global state of distributed systems. *ACM Transactions on Computer Systems*, 1(2):63–75, February 1985.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. ACM*, Vol. 17,11 643–644, November 1974.

[6] E. W. Dijkstra, W.H. Feijen, et A.J. Van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, February 1983.

[7] K. Chandy et J. Misra. *Parallel Program Design, a foundation.* Addison-Wesley, 1988.

[8] M. Raynal et J.M. Hélary. *Synchronisation et contrôle des systèmes et programmes répartis.* Eyrolles, September 1988.

[9] JM. Filloque. Spécifications de synchroniseurs matériels sur une architecture à couche logique reconfigurable. PhD Thesis, to be published in 1992, Université de Rennes I, France.

[10] J.M. Filloque, E. Gautrin, et B. Pottier. Efficient global computation on a processor network with programmable logic. In *Proceedings of PARLE'91, in LNCS Number 505*, pages 55–63, Eindhoven, NL, June 1991.

[11] M.J. Flynn. Some computer organizations and their effectiveness. IEEE Transaction on Computer, 21(9), 1972.

[12] L. Lamport. A new solution of Dijsktra's concurrent programming problem. *Comm. ACM*, Vol. 17,8 453-455, Aug. 1974.

[13] N. Margolus et T. Toffoli. *Cellular Automata Machines.* MIT Press, 1987.

[14] J. Misra. Detecting termination of distributed computations using markers. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 290–294, Montréal, Canada, 1983.

[15] B. Pottier. *ArMen, une machine parallèle intégrant un réseau de circuits logiques reconfigurables.* PhD Thesis, Université de Rennes I, France, June 1991.

[16] H.Thomas. *Une approche de la compilation de programmes séquentiels pour la machine à mémoire distribuée.* PhD Thesis, Université de Rennes I, France, June 1991.

[17] XILINX. *Reference guide.* Xilinx, 1990.

A The generated circuit for a mutual exclusion service

The diagram of synthesized circuit is shown in figure 3. The numbers in brackets refer to equations.

The PALASM format of generated boolean equations for the node 0 are the following :

The variables $accessIn_0$ and $accessIn_1$ are the 0 and 1 bits of the memorized data in the transputer/LCA interface, the variable $flagG$ is equal to the value of the variable $flag$ in the left neighbouring node, $accessOut_0$ and $accessOut_1$ constitute the 0 and 1 bits of the word read by the transputer in the transputer/LCA interface.

(1) $accessOut_0 := /accessIn_0 * accessIn_1 * (/flag * /flagG + flag * flagG)$
(2) $accessOut_1 := /accessIn_0 * accessIn_1 * (/flag * /flagG + flag * flagG)$
(3) $flag := /flag * accessIn_0 * /accessIn_1 * eat$
$+ /flag * accessIn_0 * /accessIn_1 * /eat *$

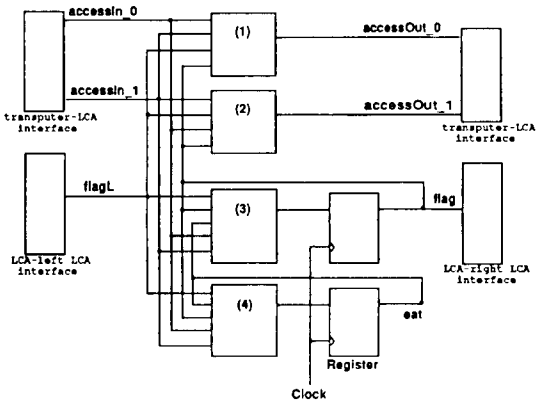


Figure 3: The synthesized circuit

$$
\begin{aligned}
& (/flag * /flagG + flag * flagG) \\
& + flag * /(accessIn_0 * /accessIn_1 * eat) * \\
& /(accessIn_0 * /accessIn_1 * /eat * (/flag * /flagG + flag * flagG)) \\
(4)\ eat \quad & := /eat * \\
& /accessIn_0 * accessIn_1 * (/flag * /flagG + flag * flagG) * \\
& /(accessIn_0 * /accessIn_1 * eat\ ) \\
& + eat * \\
& /(/accessIn_0 * accessIn_1 * (/flag * /flagG + flag * flagG)) * \\
& /(accessIn_0 * /accessIn_1 * eat) \\
& + eat * \\
& /accessIn_0 * accessIn_1 * (/flag * /flagG + flag * flagG) * \\
& /(accessIn_0 * /accessIn_1 * eat)
\end{aligned}
$$

The Programming Language Modula-P*

Jürgen Vollmer
GMD Research Group at the University of Karlsruhe†

February 12, 1993

Abstract

The programming language *Modula-P* extends *Modula-2* with parallel language constructs. It is based on the *Communicating Sequential Processes* (CSP) and the concept of the *conditional critical regions*.

This paper extends the previous language definitions of *Modula-P*, it defines the third release of the language. Some typical program examples are shown.

1 Introduction

Nowadays, parallel computers with various parallel architectures have become available on market, and a higher computational power is expected from their use. However the problem arises of how to program such computers in the best manner. There are at least two answers: either the compiler should automatically generate a parallel program from the sequential program text, or the programmer should make explicit the parallelism inherent to the problem. With *Modula-P* we have chosen the second approach.

Modula-P [Vollmer 89a, Vollmer *et al* 92] is a superset of the programming language *Modula-2* [Wirth 85], extented by language constructs based on the *Communicating Sequential Processes* (CSP) [Hoare 85], and the concept of the *conditional critical regions* [Andrews *et al* 83].

Modula-P is base on the MIMD[1] machine model with a distributed and/or shared memory architecture. Such a computer system consists of several processors which are able to perform the usual logical and arithmetic operations. They communicate using either shared memory or special hardware or both.

This language definition extends the old definitions [Vollmer 89b, Vollmer *et al* 92]. Besides some syntax changes, the concepts of the conditional critical regions and units and unit procedures are introduced.

The next sections present the language and show solutions for some typical problems.

2 The Language

All language constructs (except coroutines) known from *Modula-2* [Wirth 85] are part of *Modula-P*. *Modula-P* allows to execute statements in parallel (`PAR` statement) on one or several processors (`AT` statement). This placement can be done "network independent" by using relative positions like `north`, `east`, etc. The number of created processes can be determined during runtime.

Processes may communicate in two ways: they may access common shared variables or exchange messages. There is no distinction between the access of common shared variables and ordinary non-shared variables. Messages are exchanged synchronously via channels (variables of type `CHANNEL OF ...`) with the send statement (`!`) and the receive statement (`?`). Messages are automatically routed from the

*This work has been partially supported by the ESPRIT Project COMPARE (#5399).

†email: vollmer@karlsruhe.gmd.de

[1] Multiple Instruction Multiple Data

sender to the receiver through the communication network of the computer system without specifying its way by the programmer.
Processes which use shared variables as communication medium, may be executed on different processors only, if these processors have access to a shared memory. Otherwise, these processes are executed on a single processor. Processes using only messages passing for communication may always be executed distributedly.
A process may wait simultaneously for several events (**ALT** statement). Conditional critical regions are supported by the **LOCK** and **TRY** statement and the new variable type **GATE**.
Another compilation unit called *unit* is introduced. It defines a new kind of live range for variables.
The syntax of the extensions of statements is[2]:

```
statement  ::=  ... | ParStatement | AtStatement | ChannelStatement |
                CCRStatement | AltStatement .
```

2.1 Parallel Execution

The execution of a statement sequence is called a *process*. **PAR** p_1 | $\cdots$ | p_n **END** specifies the concurrent execution of its components $p_1 \cdots p_n$ The processes created by the **PAR** statement are called *child process* of the process executing the **PAR** statement, which is called the *parent process*.
Executing a **PAR** statement suspends the parent process until all of its child processes have terminated.
A process terminates after the last statement of its statement sequence has terminated its execution.
The component p_i is a statement sequence preceded by an optional *replicator*. A replicator creates several processes all executing the same statements, see section 2.7.
A statement of statement sequence of p_i must not contain a **RETURN** statement or an **EXIT** statement related to a **LOOP** statement not contained in p_i.
The syntax is:

```
ParStatement  ::=  PAR Process {"|" Process} END .
Process       ::=  [[replicator] StatementSequence] .
```

2.2 Distributed Execution

The syntax of the **AT** statement is:

```
AtStatement  ::=  AT Expression DO StatementSequence END .
```

The **AT** statement allows the execution of statements on another processor[3]. The meaning of the program is independent of the fact that statements are executed on another processor or not. An exception are those differences which are caused by possible different execution speeds.
The **Expression** must be assignment compatible to **INTEGER**. The intended meaning of the value of **Expression** is to specify a processor where the **StatmentSequence** is executed. The **Expression**'s value is interpreted by the runtime system[4]. The runtime system defines the set of valid values, and the behaviour if illegal values are passed to it. The library **net** offers functions to determine valid values for the processor numbers[5].
The runtime system may restrict the kind of statements which are allowed to be executed on another processor. It is not a programming error to specify restricted statements to be executed remotely. The **AT** statement then causes these statements to be executed on the same processor which executes the **AT** statment. A typical restriction for a system without shared memory would be that the statements are not allowed to access shared variables.

[2]The syntax is given in Backus-Naur-Form according to [Wirth 85].
[3]This is an extension of the *remote procedure call* concept.
[4]Runtime system stands for both, the actual computer and the operating system supporting the execution of a *Modula-P*-program on that computer.
[5]For example: **north()**, **east()**, etc. which specify direct neighbors of a processor. These functions also result in valid numbers, if there is no such direct neighbor processor.

2.3 Message Passing via Channels

Processes may exchange messages over channels synchronously. The term synchronous specifies that the process which reaches its communication statement with a channel *chn* first will wait until its partner process reaches the corresponding communication statement with channel *chn*. Then the communication takes place, i.e. the message is passed, and both processes continue independently.
There are n:m and 1:1 channels. For the n:m channels, n sender and m receiver processes may use a single channel *chn*. But at a given time only one sender and one receiver are exchanging a single message over a n:m channel *chn*. If there are more than one sender and one receiver ready to communicate over *chn*, an arbitrary sender and an arbitrary receiver is chosen out of the ready ones to communicate over *chn*. The processes not chosen continue to wait for exchanging their messages. It is guaranteed that this selection is fair, i.e. each sender or receiver will be chosen for communication at some time.
For 1:1 channels, only one sender and one receiver are allowed at a given time. If more than two processes try to communicate at a given time over a 1:1 channel, an error is raised.
Channels are treated in the same manner as *Modula-2* variables, i.e. they have a type (`CHANNEL OF` *MessageType*) and must be declared. Channels may be passed to procedures as parameters or may be assigned to variables of the same type. Messages (i.e. values of type *MessageType*) may be of any other *Modula-P* type[6].
Before the first communication over a channel can take place, the channel must be initialized once with the standard procedure `INIT`. The signature of `INIT` is:

```
PROCEDURE INIT (VAR channel :ChannelType; info : BITSET);
```

where `channel` is a channel variable, `info` is a set of informations which are interpreted by the runtime system. It is used for example to specify the property of a channel to be n:m or 1:1. Values for `info` are provided by the module `Channel`[7]. `info` is an optional parameter. If it is missing, the channel becomes a 1:1 channel.
By calling `INIT` for the variable `channel`, an identification is assigned which is unique for all processes created while running this program. This identification is used when a channel is passed as a parameter or assigned to another variable. Two channel variables with the same message type are equal, if they have the same identification. They may be compared using `=` and `#`, respectively. Using a non-initialized channel variable in a communication statement results in an undefined behaviour.
The statement `channel ! expression` sends the value of `expression` over the `channel`. The statement `channel ? variable` receives a message from `channel` and assigns it to the variable `variable`. `expression` must be compatible to the message type of the channel `channel`. The message type of `channel` must be assignment compatible to the type of `variable`.
Any type of message may be sent over a `CHANNEL OF ANY`[8] using only the size of the message and the address of the source / destination.

```
VAR chn : CHANNEL OF ANY; chn ! src_adr, size; chn ? dest_adr, size;
```

`chn` must be of type `CHANNEL OF ANY`. `src_adr` and `dest_adr` are expressions compatible to `ADDRESS`. `src_adr` is the address of the memory, where the message to be sent is stored. `dest_adr` is the address of the memory the received message has to be stored in. `size` is an expression which must be assignment compatible to `CARDINAL`. `size` specifies the length of the message in bytes. The message sizes of the sender and receiver must be equal. The receiver process has to allocate enough storage for the message before the communication takes place. The message size may change from one communication to the next.
Function procedures may have a channel as return type.

[6] This includes channels and gates.
[7] For example `one2one` or `n2m`.
[8] Note: `ANY` is not a language type like `WORD`; it is a keyword.

The syntax is:

```
type             ::=  ... | ChannelType .
ChannelType      ::=  CHANNEL OF MessageType | CHANNEL OF ANY .
MessageType      ::=  type.
ChannelStatement ::=  SendStatement | ReceiveStatement .
SendStatement    ::=  channel "!" expression | channel "!" adr, size .
ReceiveStatement ::=  channel "?" designator | channel "?" adr, size .
channel          ::=  designator .
adr, size        ::=  expression .
```

2.4 Shared Variables

Several processes may read or write (simultaneously) a *shared* variable. There is no syntactic difference in the use of shared and non-shared variables. To avoid the well-known problems introduced by accessing shared variables, the statements presented in section 2.5 should be used.

2.5 Conditional Critical Regions

Critical regions are needed, if resources (like variables, files, data structures, etc.) are allowed to be used at a given time only by a single or a specific maximal number of processes. The *protected statements* of a conditional critical region may be executed by a process if and only if an additional condition is fulfilled.
A conditional critical region in Modula-P implements the structured use of *binary* and *counting semaphores*.

2.5.1 The Type GATE

Modula-P has a scalar type called GATE, whose values are used in the conditions of conditional critical regions. Before using a gate variable in a critical region statement for the first time, it must be initialized once with the standard procedure INIT. The signature of INIT is:

```
PROCEDURE INIT (VAR gate : GATE; max : CARDINAL);
```

Each gate variable has two counters gate.cur and gate.max. which are invisible to the programmer. INIT initializes them to gate.cur = 0, gate.max = max. If the optional parameter max is present, its value must be $\geq$ 1, otherwise max = 1 is assumed. By calling INIT for the variable gate, an identification is assigned which is unique for all processes created while running this program. This identification is used when a gate is passed as parameter or assigned to another variable. Two gate variables are equal, if they have the same identification. They may be compared using = and #, respectively. Using a non-initialized gate variable in a critical region statement results in an undefined behaviour.

2.5.2 The LOCK and Try Statements

Conditional critical regions are expressed in *Modula-P* by means of the LOCK and TRY statements. The LOCK alternative of the ALT statement (see section 2.6) is treated as the LOCK statement. Each of these statements use a gate variable. Several conditional region statements for which their gate variables have the same unique identification number, form a single conditional region.

```
LOCK gate, nr DO stmts END
TRY gate, nr DO stmts1 ELSE stmts2 END
```

stmts and stmts_1 are called the *protected statements* of LOCK and TRY. nr is an expression having the type CARDINAL. nr is an optional argument with the default value 1.
The protected statements can be executed by a process if and only if the following condition holds:

$$\text{nr} + \text{gate.cur} \leq \text{gate.max} \qquad (1)$$

Before the execution of the protected statements, `gate.cur` is incremented by `nr`, after finishing their execution `gate.cur` is decremented by `nr`. With other words: only `gate.max` processes may execute statements, protected by `gate`, i.e. enter the critical region
A process may count more than one by using a value of `nr > 1`.
Executing a `LOCK` statement causes the process to be suspended until the condition (1) becomes true. If several processes wait for entering a critical region, fairness is obeyed.
If the condition (1) is false when starting the execution of a `TRY` statement, the statements of the optional `ELSE` part are executed immediately. Omitting the `ELSE` part is treated as the "empty" statement sequence. If the condition holds, the protected statements are executed as described for the `LOCK` statement.
The standard procedure `LOCKED (gate : GATE; nr : CARDINAL) : BOOLEAN` results `TRUE`, if and only if condition (1) does not hold. The optional parameter `nr` has the default value `1`.
The syntax is:

```
type          ::=  ... | GATE .
CCRStatement  ::=  LOCK gate ["," expression] DO StatementSequence END |
                   TRY gate ["," expression] DO StatementSequence
                         [ELSE StatementSequence] END .
gate          ::=  designator .
```

`LOCK` and `TRY` statements may be nested, but it is easy to program deadlocks. The following example shows such a situation: If both processes execute the statements `(* 1 *)` and `(* 2 *)` (fig. 1), respectively, each has entered a critical section. A deadlock occurs now, since each process tries to enter another critical section which is already protected by a gate used by the other process.

```
PAR
  LOCK a DO (* 1 *) LOCK b DO ... END END;
 |
  LOCK b DO (* 2 *) LOCK a DO ... END END;
END
```

Figure 1: Example program: Deadlock with `LOCK` statements.

2.6 Selective Waiting

A process may wait simultaneously for several events by means of the `ALT` statement. If an expected event happens, the corresponding alternative is selected and the statements of that alternative are executed.

`ALT` $\mathtt{event}_1$`:`$\mathtt{stmts}_1$ `|` $\mathtt{event}_2$`:`$\mathtt{stmts}_2$ `| ... |` $\mathtt{event}_n$`:`$\mathtt{stmts}_n$ `ELSE` $\mathtt{stmts}_0$ `END`

The process executing the `ALT` statement is suspended until at least one event out of the set $\{\mathtt{event}_1, \cdots, \mathtt{event}_n\}$ happens. If one event or several events occurred at the same time, an event $\mathtt{event}_i$ is selected out of the set of occurred events, and the $\mathtt{stmt}_i$ of the alternative `i > 0` are executed. If several events occurred simultaneously, the textual first is selected.
Four kinds of events may be expected:

(e1) Waiting for communication with another process:
`expression , channel ? variable` or
`expression , channel ? adr, size`
(e2) A process waits for entering a conditional critical region:
`expression , LOCK gate , nr`
(e3) A process waits until some time has passed:
`expression , WAIT (ticks)`
(e4) This event happens "always" and immediately:
`expression`

The expression `expression` must result in a boolean value. In the cases (e1), (e2), and (e3) it is optional and has the default value `TRUE`. The same type rules as for communication and conditional

critical regions statements must be applied. nr is an optional CARDINAL expression having the default value 1. ticks must result an INTEGER value.
An event occurs, if expression[9] evaluates to TRUE and if for event type (e1) to (e4) the corresponding condition (c1) to (c4) holds:

(c1) Another process is ready to send a message over the channel channel.
(c2) Condition (1) becomes true.
(c3) Since starting the execution of the ALT statement, the time specified by ticks has passed.
(c4) This event happens always and immediately.

If an alternative i based on an event of type (e1) to (e4) is selected the corresponding action (a1) to (a4) is performed:

(a1) first the message is passed and then $\texttt{stmts}_i$ are executed.
(a2) $\texttt{stmts}_i$ are treated like the protected statements of a conditional critical region of a LOCK statement and executed immediately as described in section 2.5.2.
(a3) and **(a4)** $\texttt{stmts}_i$ are executed.

An alternative is discarded if its boolean "mask" expression evaluates to FALSE. The ELSE part of the ALT statement may be omitted. If it is present and all alternatives are discarded, the statements $\texttt{stmts}_0$ are executed. If the ELSE part is omitted in this case, a runtime error is raised.
The syntax is:

```
AltStatement  ::=  ALT alternative { "|" alternative } [ELSE StatementSequence] END .
alternative   ::=  [[replicator] event ":" StatementSequence] .
event         ::=  [expression ","] ReceiveStatement |
                   [expression ","] LOCK gate ["," expression] |
                   [expression ","] WAIT "(" ticks ")" |
                   expression .
ticks         ::=  expression.
```

2.7 Replicators

The components of a PAR or ALT statement may be replicated. A replicated process component has the form:

```
[ident :  lower TO upper] p
```

(ORD(upper) - ORD(lower) + 1) processes are started all executing p in parallel. Each process gets a unique value of ident in the range [lower ··· upper], which is accessed using ident.
Similarly, a replicated alternative has the form:

```
[ident :  lower TO upper] event :  stmts
```

(ORD(upper) - ORD(lower) + 1) alternatives are set up, waiting for the guards. Each alternative gets a unique value of ident in the range [lower ··· upper].
For ident, lower, upper the same rules as for the FOR statement applies. The replicator variable may be used inside of the child process or the alternative only as a constant.
If (ORD(upper) - ORD(lower) + 1) ≤ 0 then no component (process or alternative) is created.

```
replicator  ::=  "[" ident ":" lower TO upper "]".
lower       ::=  expression.
upper       ::=  expression.
```

2.8 INOUT Parameter Passing

In *Modula-2* parameters may be passed in two ways either *passed by value* or *passed by reference* (also called *var parameters*). Reference parameters are specified by the VAR keyword in the formal parameter list.

[9] expression is evaluated only once.

In *Modula-P* is a third way to pass parameters: *passing by copy and result* (also called *in-out parameter*). Parameters which are passed using this method are specified by the `INOUT` keyword in the formal parameter list. As actual in-out parameter only variables are allowed, and the same rules as for passing var parameters are applied.
When calling a procedure with an in-out parameter, the value of the actual parameter is copied into the storage location specified by the corresponding formal parameter. On exit of the procedure, the value of the formal parameter is assigned to the actual parameter.
The syntax for the formal parameter declaration in procedure declarations and procedure types is:

```
FPSection       ::=  [pKind] IdentList ":" FormalType .
FormalTypeList  ::=  "(" [[pKind] FormalType {"," [pKind] FormalType}] ")"
                              [":" qualident] .
pKind           ::=  VAR | INOUT .
```

2.9 Units and Unit Procedures

Modula-P has another kind of module, so-called *units*. There are definition and implementation units. In a unit constants, types and procedures may be declared, but neither variables nor local modules. An `IMPLEMENTATION UNIT` has no body. All other rules for `DEFINITION MODULE`s and `IMPLEMENTATION MODULE`s are applied accordingly. `DEFINITION MODULE`s, `IMPLEMENTATION MODULE`s and local `MODULE`s are called *ordinary modules*. Procedures declared in a `DEFINITION UNIT` are called *unit procedures*, procedures declared in ordinary modules or in `IMPLEMENTATION UNIT`s and bodies of `IMPLEMENTATION MODULES` are called *ordinary procedures*. There is one exception: the body of the program `MODULE` is viewed as a unit procedure, too. Parameters may be passed to unit procedures only by value or as in-out parameter. All other rules of procedures apply accordingly to unit procedures.
The purpose of units is to redefine the meaning of a global variable's life time. *Modula-2* knows three kinds of life times for variables:

Global Variable A variable is *global* if it is declared in a module[10]. The variable exists as long as the program is executed.

Local Variable It comes into existence when the procedure declaring it, is called. It is removed when the procedure returns.

Heap Variable Its life time is determined by the programmer dynamically. It is created by calling a procedure `ALLOCATE`, it exists until is is released by calling `DEALLOCATE` (or similar routines).

In *Modula-P* the life time of variables declared in an ordinary module is connected to the call of the unit procedures importing this module. With other words: *all variables declared in an ordinary module which is imported (transitively) in a unit procedure, come into existence when the unit procedure is called, and are removed when the unit procedure returns.* Each instance (call) of a unit procedure has its own instance of these global variables[11].
The bodies of the modules imported by a unit are executed each time a unit procedure of these units is called. They are executed in the usual order.
Note:

- A unit procedure may always be executed on another processor using the `AT` statement. Problems may occur, if pointers to (heap) variables are passed. It is not always possible to detect this at compile time or runtime. An undefined behaviour may be the result.
- If one or several unit procedures of one unit (or several units which import the same modules) are called[12] no instance of this unit/these units share variables with other instances.

[10] which is not declared local to a procedure
[11] This may be viewed as if all global variables are allocated local to the unit procedure.
[12] e.g. recursively or in parallel

The syntax is:

```
DefUnit     ::=  DEFINITION UNIT ident ";" {import}{unit_defs} END ident ".".
ImpUnit     ::=  IMPLEMENTATION UNIT ident ";" {import}{unit_decls} END ident ".".
unit_defs   ::=  CONST {ConstantDeclaration ";"} | TYPE {TypeDeclaration ";"} |
                 ProcedureHeading ";" .
unit_decls  ::=  CONST {ConstantDeclaration ";"} | TYPE {TypeDeclaration ";"} |
                 ProcedureDeclaration ";" .
```

2.10 The Notion of Time

The notion of time is introduced by the standard library module `time`. It declares the scalar type `TIME` as

```
TYPE TIME = INTEGER
```

The semantics of values of type `TIME` is specified in the runtime system. `TIME` values may be compared with the usual relational operators, where `a < b` has to be read as "a before b", etc.
Note: Parallel executed processes running on different processors the do not need same `TIME` value at a given "real" time. The `time` module looks like:

```
DEFINITION MODULE time;
TYPE TIME  = INTEGER;
TYPE TICKS = INTEGER;
CONST sec = ...;              (* number of TICKS per second, depends on hardware *)
PROCEDURE Time (): TIME;      (* returns the actual time                         *)
PROCEDURE Delay (t : TICKS); (* suspends the process for t TICKS from execution *)
(* from an algebraic view the following should be observed:
 *  -   : TIME  x TIME    -> TICKS
 *  +   : TIME  x TICKS   -> TIME
 *  *   : TICKS x INTEGER -> TICKS
*)
END time.
```

2.11 Other Extensions/Restrictions

Modula-P has additional long and short integer, cardinal and real types `LONGINT`, `SHORTINT`, `LONGCARD`,`SHORTCARD` and `LONGREAL`.
There is another module kind called `FOREIGN MODULE`, which is like a `DEFINITION MODULE`, but the procedures specified there may be implemented in other languages, for example assembler or *C*. A `FOREIGN MODULE` has no Modula `IMPLEMENTATION MODULE`. It may declare only constants, types and procedures.
Modula-2 allows the assignment of priorities to program modules which is not supported by *Modula-P*.

3 Program Examples

3.1 Nested Conditional Critical Regions

To avoid the problems using nested `LOCK` statements, i.e. acquiring several different resources, the `TRY` statement may be used. Several strategies may be programmed to resolve the conflicts. The solution presented in figure 2(a) priorizes the first process. The solution of figure 2(b) may cause starvation, if the parameter of the call of `time.Delay(..)` is not well randomized, or if the call is missing entirely.

3.2 Conditional Critical Regions: Input/Output

Figure 3 shows how input/output to a terminal may be ordered, so that only one process uses it at a given time. The usual style of *Modula* I/O programming (i.e. calling procedures) is used, together with the `LOCK` statement. Here only one process is allowed to use a shared resource (terminal) at a given time.

```
PAR
  LOCK a DO LOCK b DO ... END END;
 |
  LOOP
    TRY b DO TRY a DO ... END END;
  END
END
```

2(a)

```
PAR
  LOOP
    TRY a DO TRY b DO ... END END;
    time.Delay (..);
  END
 |
  LOOP
    TRY b DO TRY a DO ... END END;
    time.Delay (..);
  END
END
```

2(b)

Figure 2: Example program: Nested acquiring of several resources

```
MODULE io;
FROM InOut IMPORT
     WriteString,WriteLn,ReadInt;
PROCEDURE square(i:INTEGER;g:GATE);
VAR k : INTEGER;
BEGIN
 FOR k := 1 TO i DO
  LOCK g DO
   WriteString("square");
   WriteInt(i,0); WriteInt(i*i,0);
   WriteLn;
  END;
 END;
END square;
PROCEDURE root(i:INTEGER;g:GATE);
...

VAR io_gate:GATE; i,j:INTEGER;
```

```
BEGIN (* main *)
 INIT (io_gate);
 PAR
  REPEAT
   LOCK io_gate DO
    WriteString ("squares");ReadInt(i);
   END;
   squares (i, io_gate);
  UNTIL i = 0;
 |
  REPEAT
   LOCK io_gate DO
    WriteString ("roots");ReadInt (j);
   END;
   root (j, io_gate);
  UNTIL j = 0;
 END
END io.
```

Figure 3: Example program for conditional critical regions: Input/Output

3.3 Reader/Writer Problem

The program fragment in figure 4 presents a solution to the *reader-writer problem*: several processes (readers) are allowed to read a shared resource (data structure), but only one process (writer) may modify it. Modification is only possible if no reader reads the resource while it is changed.
The data structure is viewed as if it consists of `nr_readers` single resources. A reader process may access exactly one of these resources. The exclusive access of the entire data structure is given the writer by using all `nr_readers` resources in the `LOCK` statement.
`nr_readers` `reader` processes are started in the replicated `PAR` statement concurrently to one `writer` process.

```
PROCEDURE reader (gate:GATE);
BEGIN
 LOOP
  ..; LOCK gate DO (* read *) END; ..;
 END;
END;
PROCEDURE writer(gate:GATE;nr:CARDINAL);
BEGIN
 LOOP
  ..; LOCK gate,nr DO (* write *) END; ..;
 END;
END;
```

```
VAR gate : GATE;
VAR i, nr_readers : CARDINAL;

BEGIN (* main *)
 nr_readers := ...;
 INIT (gate, nr_readers);
 PAR
   [i : 1 TO nr_readers]
      reader(gate,nr_readers)
 | writer (gate)
 END.
END
```

Figure 4: Example program for conditional critical regions: Reader/Writer

3.4 Client - Server

The program fragment in figure 5 presents the implementation of a client server model. Each `server` reads data from its channel `in`, solves the problem and sends the result over channel `out`, until the

command **quit** is read. To solve the problem, each instance of the server needs its own stack which is implemented by the ordinary module **stack** (not shown). The procedure **new** creates new input data for the problem to be solved by the server. If all nr_jobs data sets are created, it produces the command **quit**. There are two n:m channels: job is used to send the problem data to the servers, and **res** is used for the results. The main program starts **max** servers which are distributed over all processors. The client process distributes at the beginning one data set to all servers. Then it reads the result from one server and sends it a new data set. If the computation lasts too long, the procedure **report_timeout** is called.

```
DEFINITION UNIT Server;
TYPE tData=RECORD cmd:tCmd;... END;
     tDataChn = CHANNEL OF tData;
PROCEDURE server (in,out:tDataChn);
END Server.
```

```
IMPLEMENTATION UNIT Server;
IMPORT stack;
PROCEDURE solve (VAR data:tData);...
PROCEDURE server (in,out:tDataChn);
VAR data : tData;
BEGIN
 LOOP
  in ? data;
  IF data.cmd=quit THEN EXIT END;
  solve (data); out ! data
 END
END server;
END Server.
```

```
MODULE client_server;
IMPORT Channel, net, Time;
FROM Server IMPORT tData,...;
VAR job,res:tDataChn;d:tData;
    max, nr_jobs,cur_job,i,j:INTEGER;
PROCEDURE new (VAR data:tData);
BEGIN
 IF cur_job=nr_jobs THEN data.cmd:=quit
 ELSE INC (cur_job); ...
 END;
END new;
BEGIN (* main *)
 max:=...;nr_jobs:=max+...;cur_job:=1;
 OPEN(job,Channel.n2m);
 OPEN(res,Channel.n2m);
 PAR
   [i : 1 TO max]
    AT net.UseAll DO server(job,res)END
 | FOR j:=1 TO max DO new(d);job!d END;
   FOR j := 1 TO nr_jobs+max DO
     ALT
       res?d : print(d);new(d);job!d;
     | WAIT(10*Time.sec):report_timeout
     END
   END;
 END
END client_server.
```

Figure 5: Example program using units: client - server

3.5 Pipeline Processing

The example in figure 6 shows a parallel program that computes prime numbers[13]. It seems silly to compute prime numbers in this way, but this example shows how to construct an arbitrary large pipeline of processes using recursion. There are two kinds of processes: a **generator** and several **worker** processes chained by channels. The generator produces odd numbers starting with 3 up to the maximal number, and sends them to the first worker (which has number 2). Each worker process represents an already computed prime number. After starting a **worker**, it reads from its input channel **in** its prime number. After that it reads the numbers it should test for being prime relative to itself. If a read number is divisible by the worker's own prime number, this number is not prime; hence, it is discarded. Otherwise it must be sent to the next worker process for further examination over the output channel **out**. Initially two processes are created: one **generator** and one **worker**. When the **worker** process is called, two further processes are started. One of them checks the numbers read. The other recursively starts the next worker, which waits for input of its prime number or the termination signal **quit**.

[13]The original Ada program can be found in: *An Ada Tasking Demo*, Dean W.Gonzalez, Ada letters, Volume VIII, Nr. 5 , Sept,Oct 1988, page 87 ff.

```
DEFINITION UNIT worker_unit;
TYPE IntChannel = CHANNEL OF INTEGER;
CONST quit = -1;
PROCEDURE worker (in : IntChannel);
END worker_unit.
```

```
MODULE p_prime;
FROM worker_unit IMPORT IntChannel;
FROM InOut IMPORT ReadInt;
FROM worker_unit IMPORT worker,quit;
VAR max,i:INTEGER;out:IntChannel;
BEGIN (* main *)
 InOut.ReadInt (max);
 OPEN (out);
 PAR
   worker(out) (*starts 1. worker*)
 | (* test number generator *)
   out ! 2; (*send 1. prime number*)
   FOR i:=3 TO max BY 2 DO out!i END;
   out ! quit;
 END
END p_prime.
```

```
IMPLEMENTATION UNIT worker_unit;
FROM InOut IMPORT WriteInt;
FROM net IMPORT north;
PROCEDURE worker (in:IntChannel);
VAR out:IntChannel;prime,x:INTEGER;
BEGIN
 in ? prime; OPEN (out);
 IF prime = quit
 THEN RETURN ELSE WriteInt(prime,0)END;
 PAR
   LOOP  (* reads numbers to test *)
    in?x;
    IF x=quit THEN out!quit;EXIT END;
    IF x MOD prime#0 THEN out!x; END;
   END;
 | AT north () DO worker (out) END;
   (* worker on northern processor *)
 END;
END worker;
END worker_unit.
```

Figure 6: Example program using units and recursion: prime number pipeline

4 The Standard Library

There is a set of standard modules supporting the *Modula-P* language features. The important ones are:

net The **net** module provides information about the computer network. Procedures like **north**, **south**, etc.[14] allow the specification of a process topology independently of the actual network topology[15]. Constants like **UseAll** specify a strategy which is used when distributing a process over processors instead of a specific processor.

Storage is used to dynamically allocate and deallocate storage.

InOut Formatted terminal input/output. Any process executed on any processor may use it.

Files File input/output. Any process executed on any processor may use it.

time The **time** module offers procedures to get the system time, etc.

channels The **channels** module specifies the kind of additional information to be passed to a call to the **INIT** standard procedure.

sys_xxx These modules offer access to the underlying operating system of the host computer. These modules depend on the actual host system. Any process executed on any processor may use it.

Acknowledgements

Thanks to Markus Armbruster, Ulrich Drepper, Helmut Emmelmann, Ralf Hoffart and Hartmut Nebelung for their implementation support and discussions.

A Syntax Summary

```
statement     ::=  ... | ParStatement | AtStatement | ChannelStatement |
                   CCRStatement | AltStatement .
ParStatement  ::=  PAR Process {"|" Process} END .
Process       ::=  [[replicator] StatementSequence] .
AtStatement   ::=  AT Expression DO StatementSequence END .
```

[14] If there is such a neighbor processor, these procedures return the number of that neighbor, otherwise another neighbor number is returned.

[15] A network of this topology may only increase the execution speed.

```
type              ::=  ... | ChannelType .
ChannelType       ::=  CHANNEL OF MessageType | CHANNEL OF ANY .
MessageType       ::=  type.
ChannelStatement  ::=  SendStatement | ReceiveStatement .
SendStatement     ::=  channel "!" expression | channel "!" adr, size .
ReceiveStatement  ::=  channel "?" designator | channel "?" adr, size .
channel           ::=  designator .
adr, size         ::=  expression .
type              ::=  ... | GATE .
CCRStatement      ::=  LOCK gate ["," expression] DO StatementSequence END |
                       TRY gate ["," expression] DO StatementSequence
                              [ELSE StatementSequence] END .
gate              ::=  designator .
AltStatement      ::=  ALT alternative { "|" alternative } [ELSE StatementSequence] END .
alternative       ::=  [[replicator] event ":" StatementSequence] .
event             ::=  [expression ","] ReceiveStatement |
                       [expression ","] LOCK gate ["," expression] |
                       [expression ","] WAIT "(" ticks ")" |
                       expression .
ticks             ::=  expression.
replicator        ::=  "[" ident ":" lower TO upper "]".
lower             ::=  expression.
upper             ::=  expression.
FPSection         ::=  [pKind] IdentList ":" FormalType .
FormalTypeList    ::=  "(" [[pKind] FormalType {"," [pKind] FormalType}] ")"
                              [":" qualident] .
pKind             ::=  VAR | INOUT .
DefUnit           ::=  DEFINITION UNIT ident ";" {import}{unit_defs} END ident ".".
ImpUnit           ::=  IMPLEMENTATION UNIT ident ";" {import}{unit_decls} END ident ".".
unit_defs         ::=  CONST {ConstantDeclaration ";"} | TYPE {TypeDeclaration ";"} |
                       ProcedureHeading ";" .
unit_decls        ::=  CONST {ConstantDeclaration ";"} | TYPE {TypeDeclaration ";"} |
                       ProcedureDeclaration ";" .
```

References

[Andrews et al 83] G. Andrews and F. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3-43, March 1983.

[Hoare 85] C.A.R Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, Inc., 1985.

[Vollmer 89a] Jürgen Vollmer. Kommunizierende sequentielle Prozesse in Modula-2; Entwurf und Implementierung eines Transputer - Entwicklungssystems. Master's thesis, Universität Karlsruhe, May 1989.

[Vollmer 89b] Jürgen Vollmer. Modula-P, a language for parallel programming. *Proceedings of the First International Modula-2 Conference October 11-13, 1989, Bled, Yugoslavia*, pages 75-79, 1989.

[Vollmer et al 92] Jürgen Vollmer and Ralf Hoffart. Modula-P, a language for parallel programming; definition and implementation on a transputer network. In *Proceedings of the 1992 International Conference on Computer Languages ICCL'92, Oakland, California*, pages 54-64. IEEE, IEEE Computer Society Press, Los Alamitos, California, April 1992.

[Wirth 85] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, Heidelberg, New York, third, corrected edition, 1985.

Pact – A Fault Tolerant Parallel Programming Environment

Joachim Maier
IPVR
Stuttgart University
joachim.maier@informatik.uni-stuttgart.de

Pact is a parallel programming environment relieving the programmer from the burdens of parallel programming which are not really necessary to write efficient parallel programs. This is done by providing a simple synchronization model and virtual shared data with user-defined granularity and automatic consistency control. Pact guarantees user-transparent fault-tolerance with low overhead by using atomic actions as basic units of parallel execution. Additionally, the runtime system maps parallel actions to server processes using dynamic load-balancing. An included on-line visualization tool helps tuning and debugging parallel programs.

1 Introduction

Since operating systems for parallel computers have become commercially available (e.g. Tandem's Guardian, Sequent's Dynix, Intel's and Kendall Square's OSF/1), the main obstacle to broader acceptance is the absence of an easy-to-use parallel programming environment. There is an ongoing debate on what a programming language for parallel computers should look like without loosing too much efficiency. A lot of different languages have been proposed in the last years,[1] where most approaches directly reflect the hardware architecture. This allows efficient programming of the underlying hardware, but prevents from portability.

The *shared memory* programming model [Han75] is originally developed for single-processor computers where many processes share data-structures like in operating systems or database management systems. But this model can be used for shared memory multiprocessor systems in the same way [Ost86]. The fact that the shared memory model provides shared data-structures which can be concurrently used by all tasks of the parallel program is very important for parallel programming, because many parallel programs need common data. To guarantee consistency, access to shared data-structures has to be synchronized by explicitly setting semaphores which are used for other synchronization purposes, too.

The *message passing* model [Hoa78] is oriented towards *distributed memory* architectures. In this model synchronization has to be done by sending messages. Because there is no notion of common data, all data needed by the parts of the parallel program, running on different nodes, have to be distributed explicitly by sending messages.

Distributed memory seems more suitable for building massively parallel computers, because it provides better scalability, lower costs and less access conflicts than centralized shared memory. Because of its support of common data from the point of view of program-

[1] Refer to [BST89] for an overview on programming languages for distributed computing systems.

ming ease, shared memory seems to be the better choice. The approach of *virtual shared memory*[1] simulating logically shared memory on a distributed memory computer is obviously a solution of this conflict.

Li and Hudak called it *shared virtual memory* [LH86], [LH89] and showed the practicability and efficiency of their (distributed and centralized) algorithms on selected applications. Like in the well-known *virtual memory* algorithm references on non-local pages (page-faults) trigger the page's migration into local processor's memory. The algorithm is easy to implement in the operating system, but has the drawback that it cannot use application-specific information. Li and Hudak showed that the application-dependent granularity of the memory units (pages) has a significant influence on the efficiency of the implementation. Based on shared memory semantics it produces the same unpleasant network traffic like cache coherency algorithms for shared memory computers, but in a distributed memory system communication is much more expensive.

Some approaches to reduce message traffic use modified consistency semantics. Cheriton's *problem-oriented shared memory* [Che86] defines consistency depending on the requirements of the applications. They can eventually use an outdated version of the date and therefore do not need an actual copy, or are capable of computing it by themselves.

In *distributed database management systems* literature (e.g. [CP84] or [OV91]) topics like replication of data, different consistency models and object migration are discussed, too. But for database systems it is much easier to decide whether the operation has to be executed at the node where the object is located or to migrate the object to the node from where the operation is called, because the query-optimizer can decide on actual statistics about a few well-known (relational database) operations and data structures.

Since typical compute-intensive "grand-challenge" applications are running for a few days or hours, and today's massively parallel computers reliability is even lower than that of workstations, the programming environment should support *fault-tolerance*. Unfortunately most systems do not even have a checkpointing library. In this case the programmer has only two possibilities to deal with system-failures, either he restarts the program from the beginning and loses all preceding work, or he accepts the additional work to write checkpoints and recovery by himself. But even if there is such a library the complexity of parallel programming is increased. Therefore, fault-tolerance should be fully user-transparent. This can be done by *checkpointing of messages* like in [BBG83], [PP83], [JZ88], [SY85] or by using *transactions* [Gra78], [GR92].

In programming languages like Aeolus [WL86] and Argus [Lis88] fault tolerance is provided by using *atomic actions* and *reliable objects*. Atomic in this context means that actions have all-or-nothing semantics – either they run successfully and all updates on objects will become visible, or elsewhere all updates are rolled back to their previous state. Although in both languages actions can be distributed over different nodes, they were rather built for implementing system software for distributed systems than for parallel programming. Therefore, it is not necessary to migrate objects dynamically. This fits well if the objects are e.g. file-systems, but in parallel systems this makes dynamic load-balancing very difficult [Sch91].

Like in Aeolus and Argus the basic unit of execution in Pact[2] is the action. This helps providing user-transparent fault-tolerance. The actions' common data are organized in shared variables which are dynamically distributed by Pact's runtime system. This is done by using a virtual shared memory algorithm adapted from Li's and Hudak's distributed page ownership

[1] Some call it distributed shared memory.

[2] Pact is a synonym for parallel actions.

algorithm [LH89]. By embedding Pact in familiar procedural programming languages like ANSI-C, a parallel programming language is provided which is easy to learn.

In the next section the programming model is described in more detail. In section 3, Pact's extensions to procedural languages are presented. In section 4, there is a short presentation of the runtime system and the visualization tool *pactview*. Section 5 shows conclusions and further work in the Pact project.

2 Pact's Programming Model

The basic concept of Pact is to give programmers an environment which is almost as simple as that of sequential machines. This means that supplementary work for using parallelism, like synchronization of accesses to shared data, scheduling and load balancing is done by Pact's runtime system. Most people with practical experience in writing parallel programs will agree that the explicit synchronization of accesses to shared data and the coordination of parallel execution with barriers, semaphores, message passing, etc. are difficult. (As an example how difficult it can be, refer to the buffer manager code in [GR92]. In this code the access to a database buffer is synchronized with shared and exclusive semaphores.) Often complex parallel programs have completely unexpected effects or run into deadlocks. Synchronization failures are usually very hard to find, because they occur only in a special execution order. If the program is debugged using a source level debugger or print-statements the failure often cannot be found, because the execution order is modified.

Pact's approach is to avoid everything that is not really necessary to write efficient parallel programs and offer everything that helps doing this. This results in the following requirements to the parallel programming model:

- There is a need for a notion for *expressing parallelism* and to *coordinate* the execution of parallel actions.
- Actions[1] running in parallel should not deal with parallelism. They should only contain sequential code. This makes programming simpler and therefore helps avoiding programming faults.[2]
  Note that on this level it is not useful to think about which node should execute which part of the program. Therefore, typical constructions like `if (mynode() == k) do_something()` create programs which are difficult to understand and often cannot be executed on varying numbers of processors.
- Parallel actions need the possibility to *share data*. To comply with the above requirement this should be fully user-transparent.
- For providing this transparency it is necessary to guarantee *consistency* on shared data in the same transparent way by the runtime system. Therefore, there is no need for semaphores, locks, monitors, etc. in the programming language.

Similar to Aeolus and Argus, Pact's basic unit of execution is an *action*. Actions in Pact are written like usual procedures or functions. This means that they contain exclusively sequential code and that they are called using reference or value-parameters like in Pascal or ANSI-C.

[1] Some call it *tasks*, others *threads of control*.

[2] In [GR92] "KISS: keep it simple, stupid" is one of the general principles for building fault-tolerant software.

Similar to transactions, Pact's actions have the property of *atomicity*. They have all-or-nothing semantics – either an action runs up to its successful termination, and all its results and its changes to global variables get visible, or otherwise all its effects are rolled back to their previous state. If for example an action runs on an internal error (e.g. division by zero), this is reported to the main program. Now the main program may react on this error by executing the action once again with modified parameters. This is possible, because no effects of the faulty action are visible to other actions – the runtime system undoes them. If a processor fails, the runtime system is recovered and all actions which are lost – because of the processor failure – can simply be done again. This is the mechanism to provide user-transparent fault-tolerance. It is so easy here because of the atomicity of actions.

Actions can use *shared-variables* for data needed concurrently by some actions. Because consistency on shared data is provided by the runtime system, it is not necessary to think about concurrent accesses of other actions. This avoids situations like that in Figure 1. where two tasks concurrently updating shared variables compute two different results, depending on their execution order. In case 2, the update of task t_2 gets lost (this effect is called "lost updates"). In the absence of explicit synchronization on these variables the order of the updates is not in the user's hand but in that of the operating systems scheduler. This is definitively a synchronization failure in the programs, both should acquire locks on x, but the argumentation here is that these kinds of failures are frequent and therefore synchronization should better be automated. Note that in order of doing automatic concurrency control by the runtime system shared-variables do not have shared memory semantics. Therefore, it is not possible to update the same variable at the same time from two different actions. Shared variables are used exclusively to provide global data, communication and synchronization between actions are done by other mechanisms.

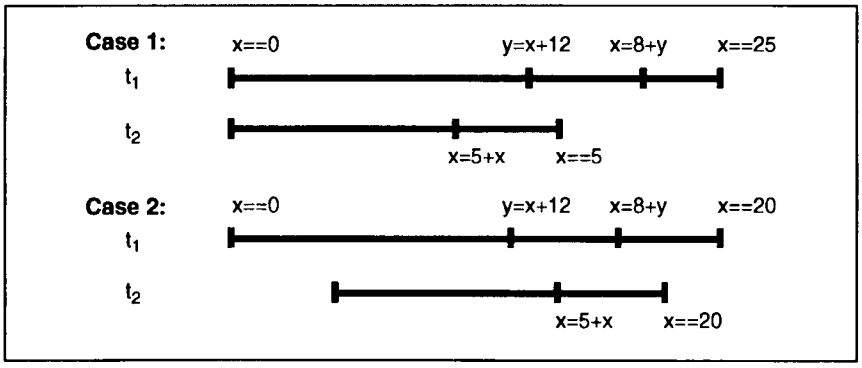


Figure 1. Lost updates caused by synchronization faults.

Following the requirements above, coordination of parallelism in Pact is completely done in the main program by defining the execution order. For simple parallel execution orders this can be done by using forall or pardo statements. In Figure 2., an execution graph for data-parallelism and function-parallelism is shown. On the left side the function f() is executed N times in parallel. Each function call is done with a different parameter. Because the same function works on different data, this is called *data-parallelism*. On the right side the functions f(), g(), and h() are executed in parallel. As in this case different functions are running in parallel, this is called *function-parallelism*. To define more complex execution orders like in Figure 4. a model similar to the precedence-graphs in [Sch91] is used.

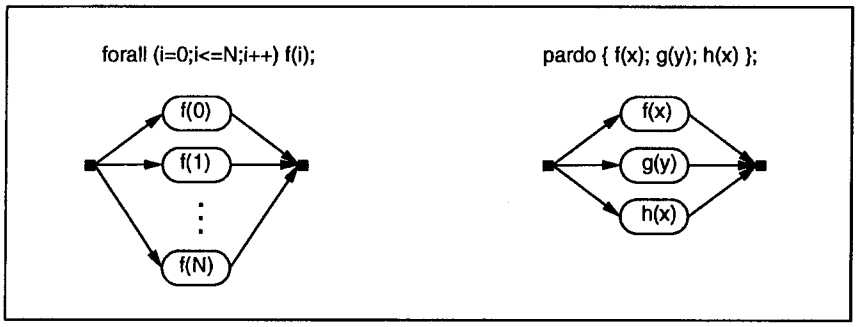


Figure 2. Data-parallelism vs. function-parallelism.

The ideas of these graphs are explained in the graph in Figure 3. The rounded rectangles symbolize the actions with the names in them corresponding to the actions' names. Rectangles which are drawn above each other represent actions which are running in parallel. Each rectangle has some arrows pointing into it and some arrows pointing out. These arrows reflect the control flow of the execution. The small squares with names besides them are called *events*. The idea is that if an event has been triggered by the termination of all preceding actions the following actions can be executed. In this example, after the action f(x) has terminated the event ParBegin is triggered. This starts the parallel execution of f(y), g(y) and h(y). After the termination of all of them the event ParEnd is triggered which causes the start of e(a,b,c).

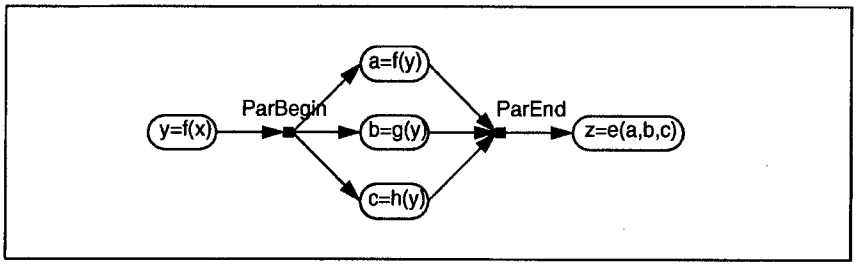


Figure 3. An example precedence-graph.

3 Pact's Extensions to Procedural Programming Languages

Pact is completely embedded in a procedural programming language. This makes it easier to learn parallel programming with Pact and simplifies its implementation, too. Hence, it is only necessary to write a precompiler which replaces Pact's language extensions by Pact's intermediate language library calls (ImPact). The extensions to C are described here in a BNF and by giving simple examples to help the reader's imagination. Pact's keywords are underlined to show the difference to C.

As described above the basic unit of execution in Pact is the action. Following the requirements of the programming model, action-definitions look like usual function-definitions containing only sequential code. The slight difference between usual C function-definitions and the definition of actions is the keyword action in front of the function's head.

```
<definition>          ::=   <action-definition> | <function-definition> | ...
<action-definition>   ::=   action <function-definition>
```

In the following example an action is defined which computes x^y.

```
action double mypower( double x, double y);
{ return( pow(x,y) ); }
```

As mentioned above, Pact has two different possibilities to define the parallel execution order of actions. The simpler way is to use well-known forall and pardo statements. The more complex one uses special actions-calls and events. Events are defined like usual variables with the predefined type event. This looks like a variable definition and it is possible to define vectors and arrays of events, too. In the LU-factorization on page 8 an array of events is used. Events are initialized with 1 by default and the event triggers subsequent execution if its value has been set to 0 by explicitly setting it with the Pact-function preset() or by the termination of a preceding action. To implement execution-plans as in Figure 3., where the execution of action e() has to be started not before all actions f(), g(), and h() have terminated, it is necessary to await all three actions' termination-events. This is done by initializing ParEnd with 3 (with preset() or like in the subsequent example in the event-definition). Each action's termination decrements the event value by one, until it equals 0. Then the event triggers the start of other actions.

Actions are called using an action-call which looks like a call to the "function" action. An action-call has four parameters: Three event-expressions and the action-expression. The first event-expression specifies the start-events (conjunction of events) which are necessary to start the execution of the action. To give more flexibility in defining execution-plans (e.g. non-deterministic execution) disjunctions of events are allowed for start-events, too. The second parameter gives the events which are triggered when the action has terminated successfully. The third event is triggered when the action runs on an internal error (e.g. division by zero or segmentation fault). All event-expressions can be omitted. If the start-event is omitted, the action runs immediately. If the termination or error-event is omitted, nothing special happens, but there is no connection possible to other actions.

```
<action-call>                  ::=   action( [<start-event-expression>],
                                           [<termination-event-expression>],
                                           [<error-event>],
                                           <action-expression> );
<start-event-expression>       ::=   <event-id> && <start-event-expression> |
                                     <event-id> || <start-event-expression>|
                                     <event-id>
<termination-event-expression  ::=   <event-id> && <termination-event-expression> |
                                     <event-id>
<error-event>                  ::=   <event-id>
<action-expression>            ::=   [<variable-identifier> =]
                                     <action-identifier>(<parameters>)
```

The following example defines the execution-order in the example in Figure 3. by using pardo on the left side and action-calls on the right side.

```
y = f(x);                                   event ParBegin, ParEnd = 3;
pardo                                       action( , ParBegin, , y = f(x));
    {
        a = f(y);                           action( ParBegin, ParEnd, , a = f(y));
        b = g(y);                           action( ParBegin, ParEnd, , b = g(y));
        c = h(y);                           action( ParBegin, ParEnd, , c = h(y));
    }
z = e(a,b,c);                               action( ParEnd, , , z = e(a,b,c));
```

To share data between some actions shared variables can be defined in the main program and in the actions using it. This is done by a special keyword shared before the variable definition, like in [Ost86]. For better efficiency it is possible to partition these shared variables into *dataunits*. With this mechanism the programmer has the ability to define the granularity optimal to the requirements of its program. Dataunits define the unit of concurrency control and transport. This means that locks are acquired for a complete dataunit. If the size of the dataunit fits the access, this reduces the number of lock conflicts to its minimum. Since the unit of transportation for the virtual shared memory algorithm it also a dataunit, problems with the granularity like reported in [LH89] are avoided. This improves the performance of virtual shared memory. The declaration of shared variables is embedded in variable declarations by defining a special storage-class:

```
<declaration>       ::=   ... | <class-or-type> <initialized-declarator-list> ;
<class-or-type>     ::=   [<storage-class>] <type-name>
<storage-class>     ::=   ... | shared
```

In the subsequent example a shared variable for a two-dimensional array is declared. Because the actions using this array are working on small sub-arrays of size 9x9 the size of the dataunit is 9x9, too.

```
shared double Array [N][N] unit [9][9];
```

An example of a parallel program with a corresponding precedence graph illustrating the parallel execution order of the actions is presented in Figure 4. The program implements a parallelized version of the Gaussian elimination algorithm.

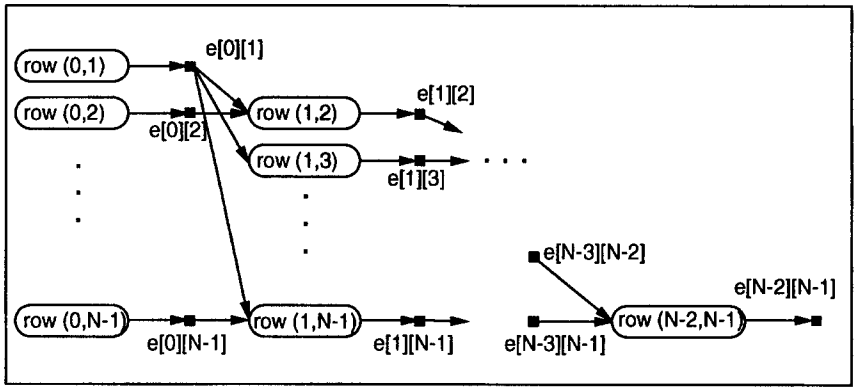


Figure 4. An precedence graph for the parallel LU-factorization using the Gaussian elimination algorithm.

Parallel LU-factorization in Pact:

```
shared double A[N][N+1] unit A[1];   /* dataunit is a row of the extended matrix A      */

action row( int i, int j)            /* computes a new row j from the old row j and row i */
{                                    /* read access to row i, write access to row j     */
    int k;
    A[j][i] = A[j][i] / A[i][i];
    for (k=j+1; k<=N; k++)
        A[j][k] = A[j][k] - A[j][i] * A[i][k];
    A[j][N] = A[j][N] - A[j][i] * A[i][N];      /* computes the vector b                 */
}

main()
{
    int i, j;
    event e[N-1]N];                  /* event matrix                                    */
                                     /* by default initialized for single events        */
    initialize_system(A);            /* equation system is loaded into A                */
    for (j=1; j<=N-1; j++)
        action( , e[0][j], , row(0,j); /* action-call without a start-event starts immediately*/
    for (i=1; i<=N-2; i++){
        for (j=i+1; j<=N-1; j++)
            action( e[i-1][i] && e[i-1][j], e[i][j], , row(i,j));
    }
    await( e[N-2][N-1]);             /* wait on the termination of the last action      */
    store_results(A);                /* store the resulting matrix somewhere            */
}
```

4 Pact's Runtime System

The prototype's runtime system which is currently implemented on a Tandem shared-nothing multiprocessor system is a process system as illustrated in Figure 5. The architecture is based on two major decisions:

- Shared data is emulated by a *virtual shared memory* algorithm.
- No dynamic process creation, but preallocated *server-processes* are used.

The decision to use server processes is made, because in absence of *threads* [ea86] process creation is too expensive for dynamic process creations. Instead of this, a preallocated number of server-processes for each type of action is started at the start-up of the runtime system. During execution of the parallel program new processes are started only as part of recovery. A virtual shared memory algorithm is necessary for an implementation of logical shared data on a distributed memory architecture. Note that more than one server process of each server type is running on one processor node. This is necessary because during the time a server process is waiting on a dataunit another server can execute on this node.

After the runtime system has been started the execution plan, which is defined in the Pact main program using actions and events, is passed to the Supervisor which controls the execution on all involved processor nodes and maps the actions to the server processes. The Node Manager controls the accesses to virtual shared data which is physically shared between the

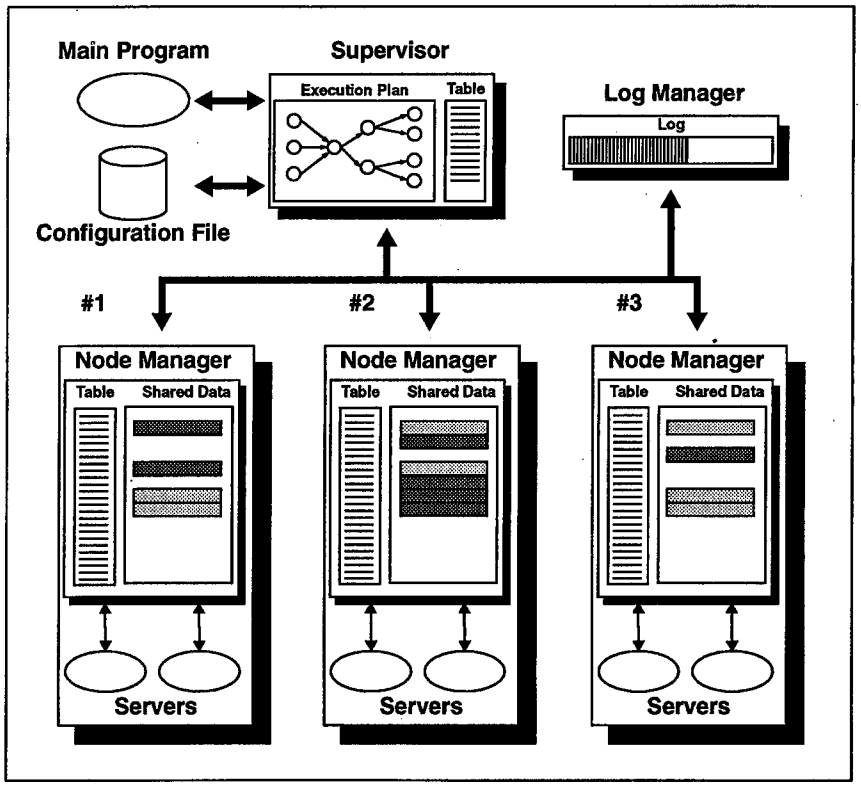


Figure 5. Architecture of the *Pact* runtime system

server processes on the processor. The Log Manager logs the actions in the runtime system in order to enable recovery.

The *Supervisor* distributes the actions to the corresponding servers on the processor nodes. This is done depending on the execution plan defined by the main program and the occurrence of start-events. For reasons of efficiency the Supervisor uses a data-dependent load balancing strategy. This means that it uses compiler generated or user given information about the dataunits an action wants to use, in order to decide if the action should run on the node which already has the necessary dataunits. A problem here is that depending on the virtual shared memory algorithm, it is impossible for the Supervisor to have actual information about the dataunits' locations. For reasons of efficiency the Supervisor cannot get involved in every data transfer between the nodes, but it gets actual information whenever an action terminates. At this time, the action server sends its results together with the identifiers of the dataunits, it has used, to the Supervisor in one message. Additional to this, the Supervisor's duties are those of a recovery manager. Whenever a component of the Pact runtime system fails, the Supervisor recovers all lost data and actions as described in more detail in [Mai93].

In order to recover system failures it is essential to hold an actual copy of all dataunits on the Log. The *Log Manager* coordinates this logging activity. Whenever an action gets ready during its *commit phase*, it has to send all changed dataunits and its results to the Log Man-

ager. When this is done successfully and all locks are released, the action's results are sent to the Supervisor. If a node crashes after this commit, the Supervisor can recover the actual version of all lost data from the Log. Another situation in which the Log is needed is when actions get into a deadlock situation or when the action fails because of a software fault. If the system detects a deadlock, it *aborts* one of the participated actions and restores the unchanged data on the node. This is equal to a rollback of all of the action's updates. Although the Log Manager is centralized in this prototype, it is easy to distribute it like it is done in some database management systems.

The *Node Managers* administrate in a fully distributed way the system's virtual shared data. This is emulated by an algorithm adapted from Li's and Hudak's *dynamic distributed page ownership* algorithm [LH89]. For this purpose each Node Manager uses a table in which the actual *probable owner* of every dataunit is stored. In this table the locks are managed, too. Whereby each Node Manager manages exactly the locks of those dataunits which are owned by itself. If a server requests a dataunit, it sends a message to its Node Manager. The latter gives it a pointer to the dataunit in the nodes' local memory which is shared between the Node Manager and the servers on the node. If the Node Manager cannot satisfy the request from its local server, it asks the probable owner of the requested unit. If this is not possible because of lock conflicts, the request is queued. Tandem's communication protocol which notes the sender, if the recipient of a message does not exist, makes it possible for the Node Manager to detect system failures. Whenever it discovers a failure, it informs the Supervisor which coordinates all necessary recovery activities. Note that in opposite to Li's and Hudak's algorithm user-defined *dataunits* instead of virtual memory *pages* are passed through the system. This reduces the number of messages and the message length, because the granularity of the dataunits fits exactly with the granularity of access. If the program needs only a few bytes, only these few bytes are sent to (and locked for) the server. If the program needs thousands of bytes which need many pages, the dataunit is sent in only one message with only one message latency.

For a more detailed description of the runtime system and the virtual shared memory algorithm refer to [Mai92].

To help the programmer debugging and controlling the parallel execution of his Pact program a X-Window based visualization tool *Pactview* [Has92] is included in the programming environment. By turning the runtime system in "trace" mode a collector process is started on the parallel computer which collects control messages from the Supervisor and the Node Managers and sends them via TCP/IP to a workstation, where the visualization is done. Pactview supports two major modes: on-line and off-line. In on-line mode the ongoing execution is visualized. In this mode it is necessary to compute dynamically the layout of the execution graph which can be very compute intensive for large graphs. Therefore, it is possible to vary some layout parameters e.g. to switch off the centralization of the graph, as it is done in Figure 6. and Figure 7. In off-line mode it is possible to replay or to step forward and backward through an execution previously recorded.

Figure 6. shows the execution of a parallel LU-factorization. Similar to the graphs before rounded rectangles show actions with their parameters and squares and arcs show events connecting them. Actions can be in four different states. A solid black rectangle shows that the action has already terminated. White shows that the action has not executed yet. Striped actions are running and actions with plaids are waiting on dataunits. Since graphs can get very big it is necessary to zoom in and out in a very flexible way. Figure 7. shows the global view of an execution graph using the "world button".

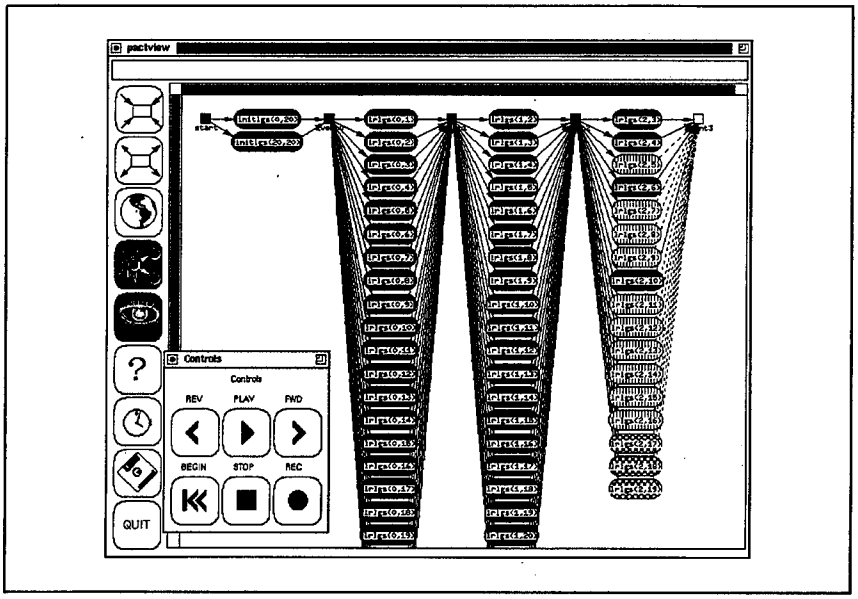


Figure 6. Screen-dump showing Pactview's visualization of a parallel LU-factorization.

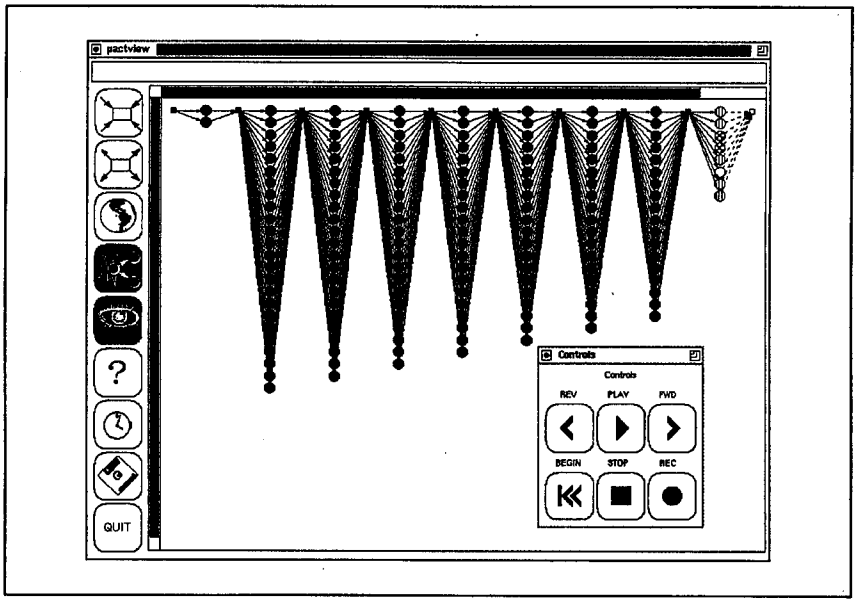


Figure 7. Global view of an execution graph for parallel LU-factorization.

5 Conclusions and Further Work

In this article, a new approach to parallel programming is presented which not only makes parallel programming very easy, but also provides user-transparent fault-tolerance. Programming ease in Pact is obtained by the programming model. To reduce the complexity of parallel programming the only place where it is necessary to deal with parallelism in Pact is the main program, where the parallel execution order is defined. This can be done because synchronization between parallel actions is either done by defining the execution order, or if it results from the coordination of concurrent accesses to shared data, it is completely automated.

User-transparent fault-tolerance in Pact is based on the atomicity of actions, which enables the repetition of failed actions. But atomicity helps not only against system failures but also gives Pact the possibility to handle programming faults in actions. This enables Pact to overcome transient software faults.

To provide shared data even on a parallel computer with distributed memory, a virtual shared memory implementation in the runtime system controls the distribution and replication of shared data which is partitioned into user-defined dataunits for better efficiency. To ensure highest performance the runtime system schedules the actions to server-processes using a dynamic data-dependant load-balancing strategy.

Talking about performance, it is necessary to note that last but not least the performance of the parallel program depends on the programmer. He has to choose the best parallel algorithm for his problem. Although the goal of Pact is to give the programmer a view to the parallel machine which abstract from most hardware details (distributed memory, number of processors, etc.), to get highest performance the programmer has to choose a granularity of parallelism which fits with the underlying hardware and he has to distinguish between real shared memory and virtual shared memory. The latter is necessary because the access to real shared memory is a magnitude less expensive than the access to virtual shared memory.

As mentioned above, the actual prototype of the Pact programming environment is implemented on a Tandem multi-processor system, but the architecture is designed to run on any other distributed memory computer as well. Only the recognition of failures, which is a precondition for implementing fault-tolerance, is much easier on a Tandem system than on most other parallel computers because it is provided by Tandem's Guardian operating system.

In the current prototype concurrency control is done by simple shared and exclusive locks. To keep the protocols simple these locks are held up to the actions' end (commit). This does not result in performance problems if the parallelism in the program is higher than the number of nodes and if lock conflicts are rare – in most applications these conditions are fulfilled – but if there is a "hot spot" dataunit there is a danger of lock contention which results in serialization. To reduce this danger, the implementation of additional concurrency control mechanisms (e.g. optimistic concurrency control [KR81], [RT90]) with weaker – user defined – consistency are planned.

The Pact prototype is currently implemented on a system with moderate parallelism. The next version will be implemented on a massively parallel Intel Paragon supercomputer under the OSF/1 operating system.

6 References

[BBG83] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault-Tolerance. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 90–99, Atlanta, October 1983.

[BST89] Henri E. Bal, Jennifer E. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Che86] D.R. Cheriton. Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design. In *6th International Conference on Distributed Computing Systems*, pages 190–197, Los Angeles, CA, May 1986. IEEE Computer Society.

[CP84] Stefano Ceri and Guiseppe Pelagatti. *Distributed Databases*. McGraw-Hill, 1984.

[ea86] Mike Accetta et al. Mach: A New Kernel Foundation for UNIX Development. In *USENIX 1986 Technical Conference*, pages 93–112, June 1986.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[Gra78] Jim Gray. Notes on Database Operating Systems. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, pages 393–481. Springer Verlag, 1978.

[Han75] P.B. Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, pages 199–207, June 1975.

[Has92] Michael Hasan. Visualisierung von Pact Programmen unter X Windows. Master's thesis, IPVR, Stuttgart University, Stuttgart, September 1992.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, pages 666–677, August 1978.

[JZ88] David B. Johnson and Willy Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of Seventh ACM Symposium on Principles of Distributed Computing*. ACM, August 1988.

[KR81] H. T. Kung and J.T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[LH86] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, Calgary, Alberta, Canada, August 1986.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lis88] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Mai92] Joachim Maier. Pact: A Fault Tolerant Parallel Programming System with Virtual Shared Data. Technical report, Fakultät Informatik, Universität Stuttgart, February 1992.

[Mai93] Joachim Maier. Fault Tolerance Lessons Applied to Parallel Computing. In *Proceedings of the Compcon '93*, San Franzisco, February 1993.

[Ost86] A. Osterhaug. Guide to Parallel Programming. Published by Sequent Computer Systems, Inc., Beaverton, Oregon, 1986.

[OV91] M. Tamer Oezsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice-Hall, 1991.

[PP83] Michael L. Powell and David L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 100–109. ACM, October 1983.

[RT90] Erhard Rahm and Alexander Thomasian. Distributed Optimistic Concurrency Control for High Performance Transaction Processing. In *Proceedings of Parbase*, pages 490–495. IEEE, 1990.

[Sch91] Gerhard Schiele. *Kontrollstrukturen zur Ablaufsteuerung massiv paralleler Datenbankanwendungen in Multiprozessorsystemen.* PhD thesis, IPVR, Universität Stuttgart, Germany, December 1991.

[SY85] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[WL86] C.T. Wilkes and R.J. LeBlanc. Rationale for the Design of Aeolus: A system programming language for an action/object system. In *International Conference on Computer Languages*, pages 107–122, New York, 1986. IEEE.

Parallel Processing for Real-Time Image Analysis of Aircraft Engines

N.W. Campbell A.G. Chalmers B.T. Thomas

Advanced Computer Research Centre
University of Bristol,
Bristol, UK.

Abstract

In this paper we describe the Rolls-Royce VIMS2 project which allows the thermal expansion of aircraft engines to be determined by tracking features of interest within X-ray images. This information is used in the analysis of engine design. A parallel implementation of the tracking algorithm achieves real-time rates using a demand driven computational model on a network of Inmos T805 transputers.

1 Introduction

Modern aircraft engines have reached a level of sophistication and performance such that advanced methods of design and monitoring are required to validate and improve designs. In particular, the response of the engine when running at full speed is extremely important, but traditionally, collecting such data has been difficult. The use of probes or sensors inside the engine obviously affects the performance of the engine, and hence the results obtained. Therefore, an X-ray imaging technique has been developed which allows the inside of the engine to be monitored without the use of intrusive sensors.

The method uses a high-energy X-ray source passed through a small part of the engine, and projected onto a screen. This screen is then imaged by a video camera connected to an image processing system, as shown in Figure 1. Computer vision techniques are used to analyse the images produced and parts of the engine are tracked over a period of time as their positions change due to thermal expansion, thereby providing design engineers with invaluable and hereto unavailable data.

The engine components to be tracked, typically tips of turbine blades and gas seals, are defined by the user drawing templates on the engine image. These templates are then located as the part moves by use of template matching. However, since the image is very noisy, the templates are typically very large and hence tracking them is computationally expensive. The parallel implementation of this problem on a network of processors forms the basis of this paper.

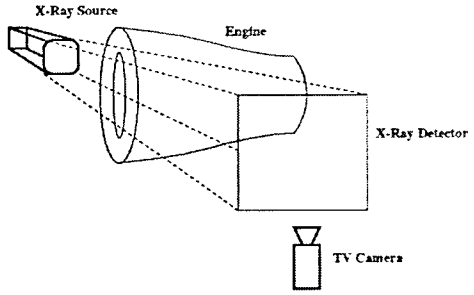


Figure 1: X-Ray Imaging System

2 Background

2.1 Correlation

A template can be located in an image by the use of correlation [4]. This compares pixels in the template with pixels in the image and returns a result which indicates how closely matched the template and the image are at this point. The correlation measure used here is the Euclidean distance measure. This is computationally inexpensive, but very robust. If the ith template point is defined by a grey-level and an x and y offset $\{\nu_i, X_i, Y_i\}$ then the correlation measure for a P point template:

$$C(x,y) = \frac{1}{P}\sum_{i=1}^{P}(\nu_i - I(x+X_i, y+Y_i))^2 \qquad (1)$$

where $I(x,y)$ is the grey-level at the image point (x,y). The time taken to compute this value is proportional to P.

A typical engine image and a rectangular template are shown in Figure 2.

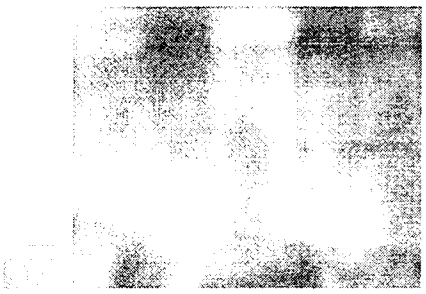


Figure 2: A Template and Typical Engine Image

Although the template shown here is rectangular, user-defined ones in the system can be of any shape. Figure 3 shows the *correlation surface* produced by applying the correlation function with our template at every position in the image. Close matches are shown by minimum values, but as is typical for any correlation surface, there are many local minima. To find the global minimum we must search the whole image. If, however, we know the approximate position of the template (perhaps from a previous frame) then we can initiate a search beginning from this point and gradient descending towards the minimum. While this technique ensures that we will quite quickly find a minimum, it does not guarantee that the global minimum is located. Therefore, the better our initial estimation of the template position, the faster we will locate it and also the more confidence we will have that it is the global minimum.

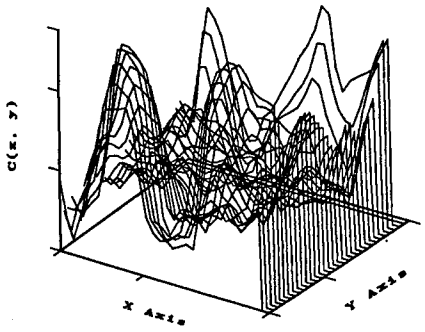


Figure 3: Typical Correlation Surface

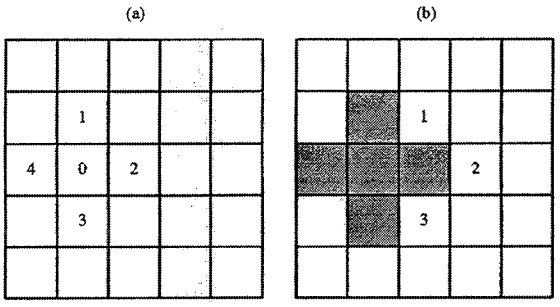


Figure 4: Gradient Descent Algorithm

The gradient descent technique is shown in Figure 4. The central pixel and its nearest four neighbours are searched in (a) with tasks 0...4. The minimum value of these five locations is returned for task 2. Therefore in (b) the search is extended and 3 new correlations are performed. This process is repeated until the minimum is located at the central pixel. If

this minimum is greater than some specified tolerance and therefore, a local rather than a global minimum, an extended search is commenced. This extended search is carried out by correlating the template at many points in a pre-determined spiral pattern centered around the last known position of the template. This is computationally expensive, but fortunately is rarley required.

2.2 The System

The image processing hardware used for this project to track multiple templates at video rates is shown in Figure 5. All of these boards, designed in-house, are based around the Inmos transputer series. The main processing function of the system is carried out by eight worker processors connected using the transputer's links in a tree topology to a system controller. Image data is passed around the network via a high speed digital video bus.

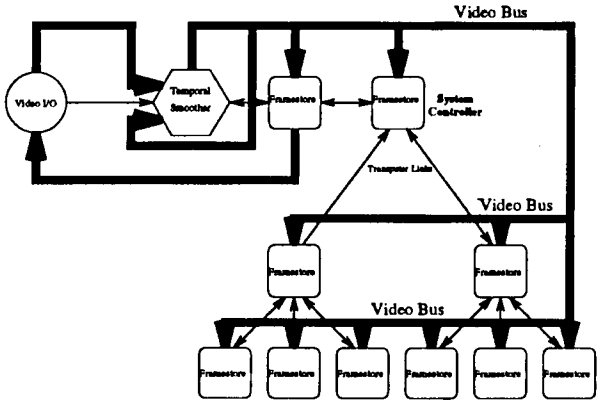


Figure 5: Image Processing Hardware

The main processing element is the framestore board. This has a T805 transputer with access to both program and video memory. Images are can transferred on the digital video bus using direct memory access, controlled by Xilinx type logical cell arrays. This ensures that none of the transputer link bandwidth is used transferring image data. Other boards in the system include a digitiser and display board which can broadcast images to framestores and then display results via a colour monitor. The temporal smoother allows the current image to be blended with previous images and thus reduce noise. The relative weightings of this blending are software defined, and depend upon the amount of movement in the scene. High amounts of smoothing are used for slowly moving scenes for maximum noise supression, but less smoothing is used for quickly changing scenes.

The tracking results obtained by the system controller will be returned to the display board at video rates. However, when the templates being tracked are too large or are difficult to track, a slower rate will result. The system controller is responsible for setting this 'time-slice' and ensuring that results are ready in time. The controller must also determine a template's

position to sub-pixel accuracy by interpolating the correlation surface, and prediction of the template's position in the next frame is achieved using a Wiener predictor [6].

3 The Parallel Implementation

While the system controller provides the input and output interface, the other processors, known as the processing elements (PEs) perform the actual processing of the problem. Since templates can vary considerably in size it is not desirable to assign one processing element the task of tracking a particular template since poor load balancing will result. The amount of work necessary per frame is not constant since features may be moving at different rates. When a feature moves quickly, load balancing is especially important since many processing elements need to be available to assist in its tracking.

Domain decomposition of a problem exploits the parallelism that exists in applying the same algorithm to different data items at the same time. Each of the processing elements in a system using domain decomposition, therefore, typically contain a copy of the entire algorithm. On multiprocessor systems which communicate via message passing, domain decomposition of the problem may be accomplished by either a *data driven* or a *demand driven* model of computation [3, 1]. Data driven models are more appropriate when the computational variation associated with each task is known, while demand driven models, of the which May and Shepherd's processor farm [5] is a simple example, are able to cope more efficiently with unbalanced work loads.

The size and position of the templates are determined before the tracking commences. With this em a priori knowledge it is possible to determine an approximately balanced work load for the initial tasks. However, if any features moves, it will only be possible to calculate a balanced work load for the further correlations required once all the initial results have been assimilated.

A demand driven implementation allocates tasks from a central "pool", where at the start of a frame boundary the pool contains the initial tasks. Once all the initial tasks of one feature have been completed, the system controller can determine if any further work is required, and if so these additional tasks can simply be added to the "task pool" for future allocation to processing elements. Similarly, once these additional tasks have been completed, the system controller can again determine if still more work is required and add yet more tasks.

A hybrid of the data driven and demand driven model is able to use the lower communication overheads of the data driven model for the initial tasks and then switch to the more flexible demand driven model should any of the features move. However, to avoid buffer overflow, such a hybrid system is unable to switch from the data driven to the demand driven model until *all* the results from the initial tasks have been received.

Previous work has shown that the demand driven model is significantly more efficient when tracking moving features than the data driven model and so this model has been adopted [2].

3.1 Structure of a Processing Element

Figure 6 shows the structure of a processing element. The application process and correlation function perform the desired computation, while the communication within the system is dealt with by two router processes, the task router (tR) and the result router (rR). As their names suggest, the task router is responsible for distributing the tasks to the application process, while the result router returns the results from the completed tasks back to the system controller. The frame grabber process sets up hardware registers to allow the next image to be loaded into the board's video memory. Occam is used for the parallel harness and C for the application specific correlation function and to drive the frame-grabber.

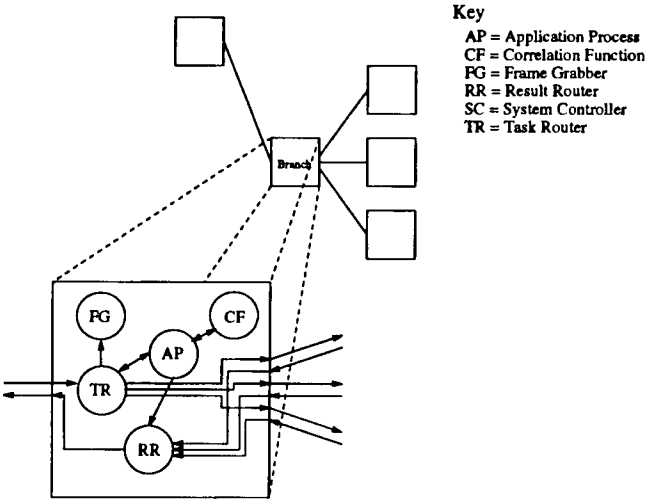


Figure 6: The Structure of a Processing Element

To reduce possible processing element idle time, each task router process contains a single buffer in which to store a task so that a new task can be passed to the application process as soon as it becomes idle. When a task has been completed the results are sent to the system controller. On receipt of a result, the system controller releases a new task into the system. This synchronised releasing of tasks ensures that there are never more tasks in the system than there is space available.

On receipt of a new task, the task router process checks the destination address of the task. If this address is not the address of the processing element, the task router routes the task onwards, otherwise the task router either:

- passes the task straight to the application process if it is waiting for a task; or
- places the task into its buffer if the buffer is empty otherwise

3.2 The Structure of the System Controller

The system controller is responsible for distributing work to the network of processing elements, collating partial results and making intelligent decisions about when to stop sending out task packets in readiness for the next time-slice. The main algorithm is a sequential piece of code which reads in a result and then sends out another task, if one is available from a pool. If no task is available then this processor is queued waiting for more work to arrive.

There are in fact three pools used by the system controller. The first pool is a list of idle processing elements. Since each processing elements can be processing a task while buffering another, the size of this pool is twice that of the number of processors available. The second pool contains the initial tasks which must be completed for each template, that is the initial search size plus any gradient descent tasks which result if the feature moves. The third pool is the list of tasks, the extended jobs, which need to be completed when a template is 'lost', that is, the minimum found by the gradient descend algorithm is judged to be a local and not a global minimum. The sequence of events is represented by Figure 7.

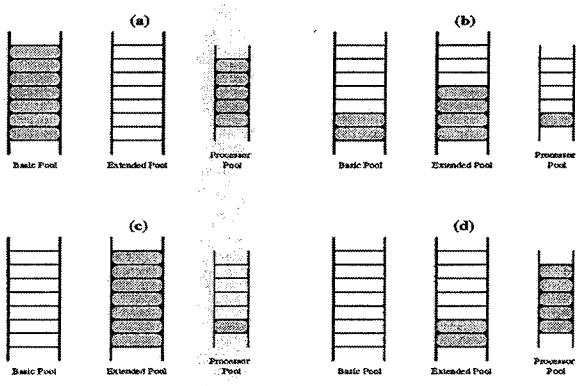


Figure 7: System Controller Pools

At the beginning of processing (a) the basic pool is full of initial jobs and no processor has work to do. These jobs are then distributed to the waiting processing elements until either there are no more tasks in the pool or no more processing elements available to execute them. At some time later (b) most initial jobs have been completed and from the results of these, extra gradient descent jobs have been added to the pool. No extended jobs are executed while any tasks remain in the basic pool. This is the case in (c) where only extended jobs are left to be executed. Once all the jobs are completed, all processing elements will be idle (d). The system controller can then instruct all framestores in the system to update their video memory with the next image.

If, however, too many templates mis-track then the extended pool may still have jobs awaiting execution at a time-slice boundary. These tasks will be stored and not used again until the initial tasks for the rest of the templates in the next frame have been completed. In this way, once a template is 'lost' then it is only re-found as a background computation.

The system controller ensures that all framestores are idle and capable of grabbing the next frame when a time-slice is reached, by sending out no further tasks after an appropriate deadline has been reached. This deadline is easily calculated since the time taken for processing is dependent only upon the number of pixels in each template.

4 Discussion

For small numbers of processing elements it has been found that communication overheads do not play a significant role. Therefore, time taken to locate N templates in a frame using P processing elements, assuming no extended search is required, is related to the size of each template, S_i and the distance in pixels moved by each template, d_i:

$$T = \frac{1}{P}\sum_{i=1}^{N} kS_i(5 + 3d_i) \tag{2}$$

The constant k is the time taken to correlate a one pixel template at one position. The $(5 + 3d_i)$ term comes from the gradient descent algorithm where 5 correlations are required initially, followed by 3 for every pixel moved.

The specification for the VIMS2 project requires that eight templates of 200 pixels each, moving at a rate of one pixel per frame, should be tracked at real-time rates using eight processing elements . In Equation (2), k has been found by experiment to be $8\mu s$ and so the total time required is 12.8ms, very much better than the frame rate of 40ms. Thus, the features can still be tracked even if moving at 5 pixels per frame when using 200 pixel templates, or, as is much more probable, at 1 pixel per frame using 625 pixel templates.

The parallel implementation described here, has allowed features of interest in X-ray images of aircraft engines to be tracked in real-time. The production system will shortly be delivered to Rolls Royce.

Acknowledgements

We would like to thank Dr Roger Miles and the Bristol Transputer Centre. The VIMS2 project is funded by Rolls Royce Plc. Neill Campbell would like to thank SERC and Rolls Royce for their financial support.

References

[1] A. G. Chalmers. *A Minimum Path system for parallel processing.* PhD thesis, University of Bristol, Department of Computer Science, Aug. 1991.

[2] A. G. Chalmers, N. W. Campbell, and B. T. Thomas. Computational models for real-time tracking of aircraft engine components. In A. Wagner, editor, *13th North American Transputer Users Group conference*, IOS Press, Vancouver, May 1993.

[3] A. G. Chalmers and D. J. Paddon. Communication efficient MIMD configurations. In *4th SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, 1989.

[4] R. Gonzalez and P. Wintz. *Digital Image Processing*, chapter 3. Addison Wesley, 1987.

[5] D. May and R. Shepherd. *Communicating Process Computers.* Inmos Technical Note 22, Inmos Ltd., Bristol, 1987.

[6] N. Wiener. *Extrapolation, interpolation, and smoothing of stationary time series with engineering applications.* MIT press, 1949.

A Parallel Processing Environment for Solving Problems with Global Communication Requirements

Alan G. Chalmers Derek J. Paddon

Department of Computer Science,
University of Bristol,
Bristol BS8 1TR,
United Kingdom.
alan@uk.ac.bristol.compsci

Abstract

Large multiprocessor systems may be necessary if complex science and engineering problems are to be solved in acceptable times. The parallel implementation of many of these problems will require global communication of data and tasks within the system. The overheads that result from this communication pattern will increase as the number of processors grows. Thus, if large scale parallelism is to be possible these overheads must be effectively tackled. This paper describes a parallel environment which combines minimum path configurations with efficient system software to minimise these overheads.

1 Introduction

Distributed memory multiprocessors offer one solution to an ever increasing demand for more performance from computer systems. Large numbers of these distributed memory processors may be necessary if complex science and engineering applications are to be solved in acceptable times.

In order to co-operate in the parallel solution of a problem, the individual processors need to communicate data and tasks. The nature of the problem determines the pattern and frequency of communication required. When a problem with a global communication pattern is implemented on a distributed memory multiprocessor system, every processor will need to communicate with all other processors. The overheads that result from this global communication increase as more processors are added. There are a large number of problems, from a wide variety of disciplines, which have these global communication requirements. Some of these problems include: radiosity methods; free Lagrangian codes; boundary element methods; spectral methods; and the modelling of molecular interactions.

For a fixed size problem, there exists an optimum number of processors on which a particular class of problem should be run in order to obtain the best possible performance. This optimum number of processors is a function of both the computation to communication ratio of the problem, and the configuration on which the problem is implemented [1]. For problems which exhibit a high degree of global communication, it is known that the use of traditional

parallel processing approaches results in quite a low value for this optimum number of processors [2]. Therefore, if complex engineering problems requiring global communication are to be solved in acceptable times, effective ways have to be found which will raise this optimum number of processors substantially.

2 The Parallel Processing Environment

The environment we use to implement problems with global communication requirements, consists of two parts:

1. a number of processors arranged in a minimum path (AMP) configuration; and,
2. system software for controlling Input/Output, communication, task allocation and data management.

2.1 System Architecture

To provide a useful parallel processing platform, a multiprocessor system must have access to Input/Output facilities. Most systems achieve this by designating one processing element as the *system controller* (SC) with the responsibilities of providing this Input/Output interface. The other processors perform the actual processing of the problem and may be termed the processing elements (PEs).

In addition to providing the Input/Output facilities, the system controller may also be used to collect and collate results computed by the processing elements. In this case the system controller is in the useful position of being able to determine when the computation is complete and thus "close down" the processing elements. If we separate the activities of the system controller according to areas of responsibility then we can construct a system controller as a number of processes as shown in figure 1.

The User Manager (UM) is responsible for input from the user via the keyboard and for displaying on the screen for the user such useful information as: how the work is progressing; results obtained so far; an analysis of the results etc. The Graphics Manager (GM) is responsible for overseeing the correct display of any graphical results on a specialised graphics device (if appropriate). This will typically entail such actions as colour table lookup, necessary transformations for rendering, etc. The File Manager (FM) controls the reading of data from, and writing of data to, the secondary storage devices. The activities of these processes are co-ordinated by the Application Controller (AC) process which also provides a routing facility between the processing elements and the system controller processes, and ensures the processing elements are initially provided with the correct routing information and application parameters. The application controller process may also be responsible for terminating the processing elements either when the computation is complete, or should the user wish to abort the proceedings.

The solution of a complex problem will require each processing element to undertake a variety of activities. The structure of a processing element which we will need to carry out these areas of responsibility is shown in figure 2.

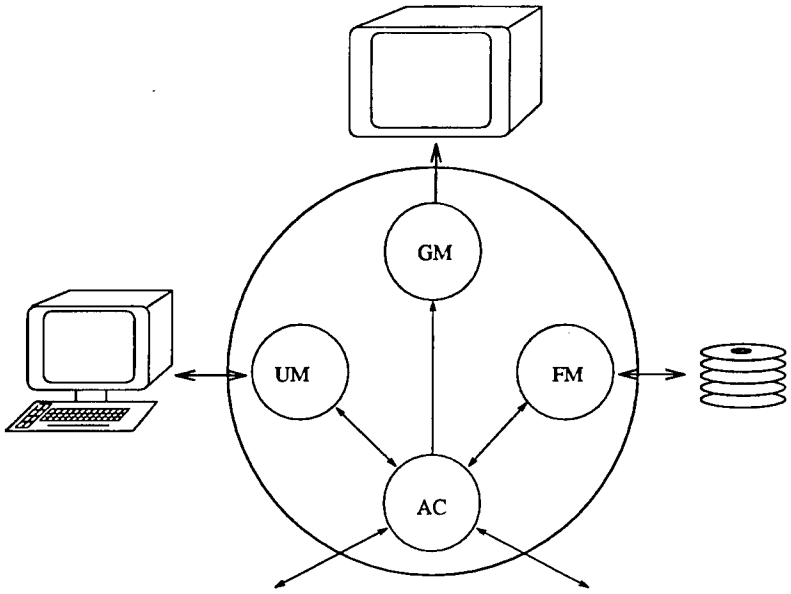


Figure 1: A system controller

In general, a processing element will consist of at least one Application Process (AP) which carries out the computation associated with the application, a Data Manager (DM) to manage the data requests, while demands for tasks and load balancing functions are handled by a Task Manager (TM). The Router process (R) is responsible for the communication of messages to and from the local processes to the system controller and/or other processing elements, as well as routing other messages on towards distant destinations. All node activities are controlled by the Local Controller (LC).

More than one Application Process may be used on each processing element to reduce processing element idle time when data items have to be fetched from remote locations. An Application Process Controller (APC) co-ordinates the activities of these multiple APs. The introduction of a local controller process allows the activities of all the processes on the processing element to be co-ordinated. Requests for data items and tasks pass through the local controller *en route* to the data manager or task manager. This allows any addresses to be converted from a local to a system wide format. The local controller may also be in possession of application specific details which will determine which computational model should be selected.

2.2 System Configurations

The philosophy underlying the construction of a minimum path configuration is to minimise the *diameter*, of the interconnection network; that is, to minimise the number of links a message has to travel between any source processor and any other destination processor within

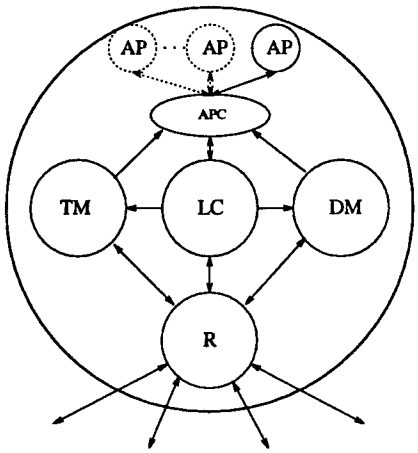


Figure 2: A processing element for solving complex problems

the configuration. This principle is maintained even at the expense of the loss of regularity in a system [3]. Current technology, such as the Inmos T800 transputer [9], is limited to four "on chip" links and so the AMP configurations discussed here are limited to four links per processor. Their methods of construction are equally applicable to systems of processors with a higher degree of connectivity [3].

Figure 3 shows the 63-processor AMP. Each processor has four links and the diameter of the configuration is 3. Table 1 gives the diameters of a number of processors arranged in AMP configurations and six of the configurations frequently used in multiprocessor systems: chains, rings, hypercubes, meshes, tori, and ternary trees. As can be seen from the table, for a given number of processors, the diameter of AMP configurations are less than any of the other configurations, and indeed the 64-processor AMP configuration has the same diameter as a 8-processor ring.

	Processors								
	8	13	16	23	32	40	53	64	128
AMP	2	2	3	3	3	4	4	4	5
Hypercube	3	-	4	-	5	-	-	6	7
Torus	3	-	4	-	6	7	-	7	12
Ternary Tree	4	4	5	6	6	6	8	8	10
Mesh	4	-	6	-	10	11	-	14	22
Ring	4	6	8	11	16	20	26	32	64
Chain	7	12	15	22	31	39	52	63	127

Table 1: Comparison of configuration diameters

Table 2 gives the average interprocessor distance values for the different configurations. This average interprocessor distance is equivalent to the average number of links a message has to cross from any source processor to its desired destination processor. So, for example, a message from a source processor in a 64-processor AMP would have to cross 2.92 links on

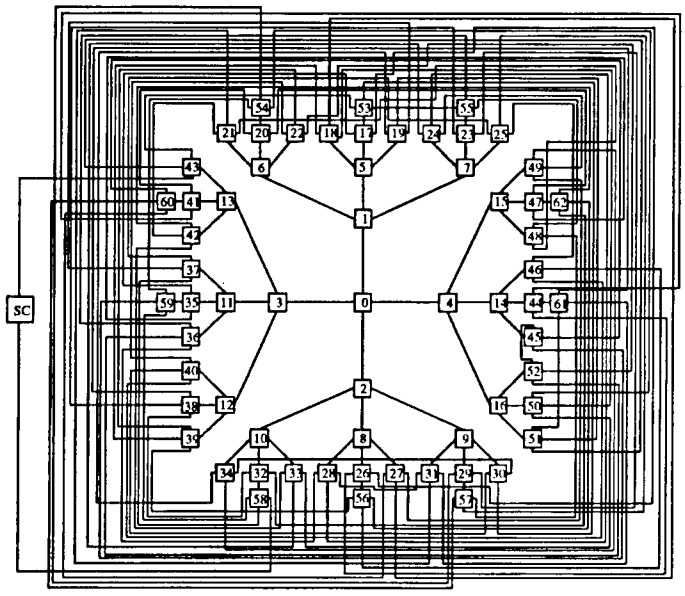


Figure 3: The 63-processor AMP configuration

average while, for a processor on a 64-processor ring, the same message would have to traverse on average 16 links.

	Processors								
	8	13	16	23	32	40	53	64	128
AMP	1.28	1.55	1.73	2.05	2.31	2.53	2.76	2.92	3.58
Hypercube	1.50	-	2.00	-	2.50	-	-	3.00	3.50
Torus	1.50	-	2.00	-	3.00	3.50	-	4.00	5.64
Ternary Tree	1.97	2.56	2.91	3.39	3.93	4.25	4.77	5.01	6.25
Mesh	1.75	-	2.5	-	3.88	4.23	-	5.26	7.94
Ring	2.00	3.23	4.00	5.74	8.00	10.00	13.25	16.00	32.00
Chain	2.63	4.31	5.31	7.65	10.67	13.99	17.66	21.33	42.66

Table 2: Comparison of average interprocessor distances

The number of intermediate processors that the message has to pass through *en route* to its destination is one less than the average distance. As the computation of these intermediate processors may be interrupted during the course of the message transfer, it is desirable that this number of processors should be as low as possible. So, from Table 2, it can be seen, for example, that on average 5 more processors are disturbed for a message on a 32-processor ring than for a message on a 32-processor torus. The average interprocessor distances for the AMP configurations are lower than all the other configurations shown except for the 128-processor (7-dimensional) hypercube where it is marginally higher. However, it must be remembered that the 128-processor hypercube requires *seven* links per processor as opposed to the four links that are used in the AMP configurations, and despite this, the diameter of the 128-processor AMP is less than that of the hypercube with the same number of processors.

2.3 System Software

As well as providing configurations with a minimised diameter, the environment also consists of system software designed to help reduce the impact of global communication on overall system performance. This system software is responsible for supporting the desired computational model at each processing element and for providing the communication harness between these processing elements.

2.3.1 Computational Models

Domain decomposition of a problem exploits the parallelism that exists in applying the same algorithm to different data items at the same time. Each of the processing elements in a system using domain decomposition, therefore, typically contain a copy of the entire algorithm. On multiprocessor systems which communicate via message passing, domain decomposition of the problem may be accomplished by either a *data driven* or a *demand driven* model of computation [4].

In static data driven systems, a fixed portion of the data is allocated to each processing element. The processing elements now apply the required algorithm to their allotted data items. The disadvantage of such a computational model is the serious load balancing difficulties that can occur if the computation associated with different sections of the data differs markedly in

complexity. This imbalance can result in the majority of the processing elements standing idle while a few struggle to complete their computationally complex portions. A preprocessing stage may help to reduce these load balancing difficulties, but this requires some form of *a priori* knowledge of the computational requirements of all sections of the data, which may not be known. The availability of *a priori* knowledge enables an asymmetric allocation of the data items amongst the processing elements, such that the computational effort required to process each of these is approximately equal, thereby minimising processing element idle time.

A demand driven computational model is able to cope with unbalanced computational complexity. In this model, work is allocated to processing elements as they become idle, with processing elements no longer bound to any particular portion of the data. On completion of a task, the processing elements demand the next one from some supplier process. The *processor farm* as suggested by May and Shepherd [10] is an example of a simple demand driven system.

2.3.2 Data Management

Applications with data requirements which are such that the total number of data items is small enough to be accommodated at each processing element, may be solved without recourse to additional fetching of data items. Real applications, however, typically require far more data items than can be held at each processing element. For these applications, and for those problems with data dependencies, we find that efficient data management techniques are necessary to cope with the communication of data messages that arise.

The concept of *data sharing* may be used to cope with very large data requirements [4, 8]. Data sharing allocates every data item an unique identifier. This allows a required item to be "tracked down" from somewhere within the system, or from secondary storage if necessary. The size of problem that can now be tackled is, therefore, no longer dictated by the size of the local memory at each processing element, but rather only by the limitations of the secondary storage.

To manage all the data requirements at each processing element, the data manager process maintains a list of the data items that are currently available at the processing element. If a data item requested by the application process is available, the data manager may immediately transfer it to the application process. If not, the data manager issues a request to the router to acquire the item. If the data item's location within the system is known, the router is able to issue a direct request for the item from the appropriate processing element. On arrival at the correct destination, that router process forwards the request to its local data manager which subsequently replies with the required item. This item is then returned to the requesting router, from there to the data manager and onto the application process. If, however, the location of the data item is unknown, the source router must issue a global broadcast for the item. Once a copy of the data item is found, it is once again returned to the requesting application process via the source router and data manager.

The time delay in acquiring a data item which is not available locally can be significant. A number of techniques may be used to reduce this delay time. If it is known at the start of the computation which data items will be required by each processing element then these data items can be *prefetched* by the data manager so that they are available locally when required. This knowledge about the data items may be known *a priori* by the nature of problem, for

example, it is known beforehand that the parallel solution of radiosity methods requires all the patch data at each stage of the computation, and the order of this data is unimportant. If the order in which the data is required is unknown then ways must be found to keep the processing element busy while the data is fetched from a remote location.

One possibility is for the application process to save the current state of a task and commence a new task whenever a requested data item is not available locally. When the requested data item is finally forthcoming either this new task could be suspended and the original task resumed, or processing of the new task could be continued until it is in turn suspended awaiting a data fetch, and only then may the original task be continued. Saving the state of a task may require a large amount of memory and indeed, several states may need to be saved before one requested data item finally arrives. Should the nature of the problem allow these stored tasks to in turn be considered as task packets, then this method has the additional advantage that these task packets could potentially be completed by another processing element in the course of load balancing.

Another possible option is to have not one, but several application processes on each processing element, as shown in figure 2. Now, although one application process may be suspended awaiting a remote data item, the other application processes may still be able to continue. Again, it may not be feasible to determine just how many of these application processes will be necessary to avoid the case where all of them are suspended awaiting data. A further disadvantage of this method is the overhead incurred by the additional context switching between all the application processes, as well as the router process and the data manager, that are all resident on the same processor.

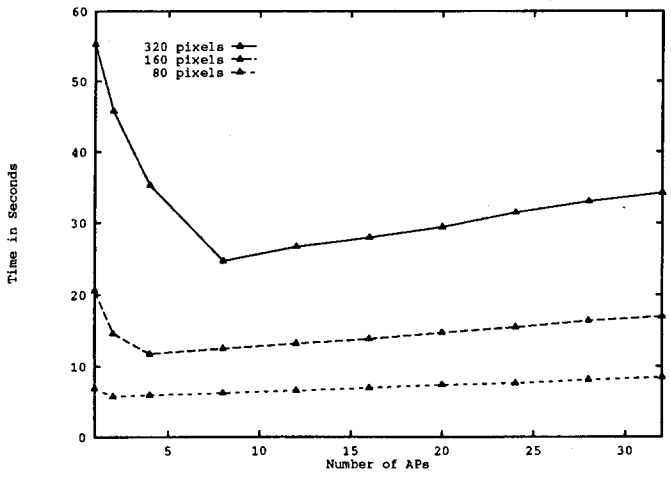


Figure 4: Time in seconds on 63 PEs for different numbers of pixels with 100 data fetches

Figure 4 shows the time in seconds to solve a complex parallel raytracing problem with large data requirements in which multiple APs are used at each processing element to minimise the processing element idle time [6]. As can be seen, increasing the number of APs per processing element produces a performance improvement until a certain number of APs are

added. Beyond this point, the overheads of having the additional APs are greater than the benefit gained, and thus the times to solve the problems once more start to increase. This becomes an increasing problem as more processing elements are added to the system because the more APs there are per processing element, the larger the message output from each processing element and as the average distances the messages have to travel in larger systems is greater, the impact of the increasing numbers of messages on message latencies is more significant. Adding more APs now no longer helps overcome communication delays, but in fact, the increasing number of messages actually exacerbates the communication difficulties. Future work will examine ways of dynamically scheduling the optimum number of APs at each processing element depending on the current system message densities.

2.3.3 Task Management

A demand driven system requires every processing element to continue performing tasks until the desired problem has been solved. Tasks for the initial stages of a problem are issued by the system controller. The application process requests the next task to be performed from the task manager. The task manager maintains a buffer of available tasks to ensure that a task is present when one is requested so as to minimise application process idle time. The size of this buffer is obviously dictated by the computation time of each task. The lower the computation time the more tasks must be buffered locally to avoid the application process being delayed while a task is fetched.

When the system begins processing the first tasks are allocated from the system controller. Thereafter, when the number of tasks in the buffer falls below a certain level, the task manager must issue a request via the router process to fetch another task. This task may be fetched directly from the system controller. However, if the processing element is "far" from the system controller then a significant period of time may pass before the task request is satisfied.

An alternative strategy is for the task request to proceed to the task manager of a neighbouring processing element. This neighbouring task manager now satisfies the request (if possible). If, in satisfying the request, this neighbouring task manager's buffer of tasks falls below the critical level, it must in turn issue a request to another task manager, and so on. The direction of this "chain reaction" is towards the application controller process of the system controller which then supplies the adjacent requesting task manager with a new task. This course of action is similar to a *producer-consumer* model where the application process is the initial consumer and its local task manager the producer. If more tasks have to be fetched then this task manager becomes the consumer and the next task manager the producer, and so on until the ultimate producer, the application controller process of the system controller.

If no further tasks exist at the system controller then the last requesting task manager may change the direction of the search. This situation may occur towards the end of a stage of processing and facilitates load balancing of any tasks remaining in task manager buffers. An additional advantage of this "chain reaction" strategy is that the number of request messages in the system is reduced, thereby lowering the overall message density.

2.3.4 Balancing of Partial Result Storage

The solution of many problems simply require a single computation on the data items to produce the desired results. However, other problems exist in which it may be necessary

to apply further computations to the *partial results* already produced. The gather radiosity method is an example of such a *multi-stage* problem [5]. In this method, the first stage entails the setting up of a matrix of form factors and the second stage involves the solution of this matrix. So, while task management provides implicit load balancing of tasks, it may also be necessary to ensure that an equal number of partial results from one stage of the computation are stored at each processing element in anticipation of the following stage.

This balancing of partial result storage could be achieved statically by all the results of a stage being returned to the system controller. At the end of that current stage the system controller is now in a position to evenly distribute these results. The communication of these potentially large data packets twice, once during the previous stage to the system controller and again from the system controller to specific processing elements, may obviously impose an enormous communication overhead. A better static distribution strategy might be to leave the results in place at the processing elements for the duration of the stage and then have them distributed from the processing elements in a manner prescribed by the system controller rather than actually communicate any of the partial results back to the system controller. In a demand driven model of computation the uneven computational complexity may result in a few processing elements completing many more tasks than others, and so a flaw in this second static storage strategy is that the individual processing elements may simply not have sufficient local memory to store more than their fair share of the partial results until the end of the stage.

Two dynamic methods of balancing this partial result data have been included in the parallel processing environment. A *conceptual* allocation of partial results to processing elements will allow the distribution of any results produced by one processing element from another's conceptual portion, due to the load balancing of the tasks, explicitly to its designated processor. Note, unlike the data driven model of computation, the portions of data items allocated to each processing element are purely conceptual in nature. A demand driven model of computation is still used, but the tasks are not now allocated in an arbitrary fashion to the processing elements. Rather, a task is dispatched to a processing element from its conceptual portion. Once all tasks from a processing element's conceptual portion have been completed, only then will that processing element be allocated its next task from the portion of another processing element which has yet to complete its conceptual portion of tasks. Generally this task should be allocated from the portion of the processing element that has completed the least number of tasks. So, for example, from figure 5, on completion of the tasks in its own conceptual region, PE_1 may get allocated task number 53 from PE_5's conceptual region. On completion of this task, the partial result produced by PE_1 will be stored at PE_5.

If this conceptual allocation is not possible, or not desirable, then balancing the partial results dynamically requires each processing element to be kept informed of the progress of all other processing elements. This may be achieved by each processing element broadcasting a short message on completion of every task to all other processing elements. To ensure that information is as up to date as possible, it is advisable that these messages have a special high priority so that they may be handled immediately by the router processes, by-passing the normal queue of messages. Once a processing element's local storage reaches its capacity the results from the next task is sent in the direction of the processing element that is known to have completed the least number of tasks and, therefore, the one which will have the most available space. To further reduce the possible time that this data packet may exist in the system, any processing element on its path which has storage capacity available may absorb the packet and thus not route it further.

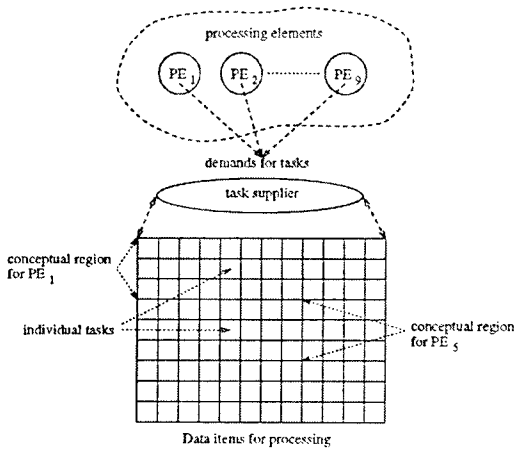


Figure 5: Partial result storage balancing by means of conceptual regions

3 Applications

The sytem architecture discussed in this paper has been used to implement a number of problems which exhibit global communication requirements, for example [2, 5]. The radiosity method from computer graphics is used to here to illustrate the effectiveness of such an environmnet for solving a problem with global communication requirements.

3.1 Radiosity

Realistic image rendering is important in many fields of application such as: science, engineering, architecture, animation, the media, etc. To achieve realism it is necessary to simulate the interactions of light between diffuse and non-diffuse surfaces that exist within a complex environment, thus giving a more natural representation of light, shade and texture. One of the approaches that closely models this physical propagation of light through the environment is the radiosity method. Radiosity methods, first introduced to computer graphics by Goral *et al* in 1984 [7], are based on the fundamental law of the conservation of energy within closed systems. They model the interaction of light between diffusely reflecting surfaces and accurately predicts the global illumination effects. The computation needed by the radiosity method is very expensive and can require very large resources in power and memory and thus is a good candidate for parallel processing.

Tables 3 and 4 give the time in seconds for an increasing problem size for AMP, torus and ring configurations of 32 and 63 processors respectively. The 32-processor AMP configuration is 7.1% faster than the torus and 34.1% faster than the ring configuration for the same number of processors. When dealing with a problem size of 2268 patches, for 32 processors the AMP configuration is now only 2.6% faster than the torus and 10.8% faster than the ring. The corresponding difference in time between the solution of 448 patches and 2268 patches on

the AMP configuration, however, has risen from 6.6 seconds to 26.1 seconds when compared to the torus configuration and 31.7 seconds to 107.3 seconds when compared to the ring configuration.

Problem size	AMP	Torus	Ring
448	92.863	99.491	124.548
700	159.794	168.202	203.336
1008	268.480	280.929	323.865
1372	437.157	446.410	498.552
1792	676.630	697.977	766.065
2268	997.340	1023.468	1104.618

Table 3: Times in seconds for increasing problem size on 32-processor configurations

Now examining the percentage differences between the time required to solve the 448 patch and 2268 problem on the 63-processor configurations, we see that the AMP is 11.2% faster than the torus and 122.7% faster than the ring for the 448 patch problem and 5.0% quicker than the torus and 37.7% quicker than the ring for the 2268 patch problem. The difference in the times taken between the AMP and the ring and torus configurations are 9.7 seconds and 107.0 seconds for the 448 patch problem and 29.5 seconds and 220.7 seconds for the 2268 patch problem.

Problem size	AMP	Torus	Ring
448	87.199	96.947	194.176
700	128.066	140.220	259.962
1008	192.156	209.165	357.064
1372	283.004	302.443	478.847
1792	411.057	434.849	643.296
2268	585.474	614.951	806.150

Table 4: Time in seconds for increasing problem size on 63-processor configurations

4 Conclusions

Parallel processing is seen as an answer to the physical limitations on the achievable performance of sequential processing. Systems of large numbers of distributed memory MIMD processors may provide the performance necessary to solve complex problems, such as radiosity methods, in acceptable times.

The AMP configurations perform consistently better than those configurations previously described in the literature with the same number of links per processor. The AMP configurations described were confined to four links per processor. This choice of four links was a compromise between the desirability of a fully connected network and the reality of currently available technology. Even so, the AMP configurations still perform better than hypercube configurations with more than four links per processor. Of course, the principles applied to minimum path configurations of processors with four links may be naturally expanded to include processors with any number of links.

The solution of complex science and engineering applications in acceptable times may require very large numbers of processors, however, we believe that it will simply not be possible to solve complex problems with global communication patterns on these large numbers of processors unless the crucial issue of communication overheads is *effectively* addressed. Parallel processing is not just solving a problem on 2 or 3 processors and then extrapolating the performance improvements that may be achieved by these low numbers of processors as an indication of the performance that may be possible on large numbers of processors. If we are to achieve anything approaching "massively parallel processing", the system architecture on which we solve these problems must combine minimum path configurations with system software which includes effective data and task management strategies and the ability to cope with partial results. Only such an environment will unify the needs of a complex problem and the sometimes conflicting features of the multiprocessor system on which the problem is to be solved.

5 Acknowledgements

We would like to thank Dr Roger Miles and the Bristol Transputer Centre for the use of their equipment.

References

[1] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS*, AFIPS Press, Reston, Va, Atlantic City, Apr. 1967.

[2] A. G. Chalmers, S. Fiddes, and D. J. Paddon. Parallel panel methods. In H. S. M. Zedan, editor, *13th Occam Users Group conference*, pages 313–321, IOS Press, York, 1990.

[3] A. G. Chalmers and S. Gregory. Constructing minimum path configurations for multiprocessor systems. *Parallel Computing*, 1993. To appear.

[4] A. G. Chalmers and D. J. Paddon. Communication efficient MIMD configurations. In *4th SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, 1989.

[5] A. G. Chalmers and D. J. Paddon. Parallel radiosity methods. In D. L. Fielding, editor, *4th North American Transputer Users Group*, pages 183–193, IOS Press, Ithaca, NY, Oct. 1990.

[6] A. G. Chalmers, D. Stuttard, and D. J. Paddon. Data management for parallel raytracing of complex images. In S. P. Mudur, editor, *International Conference on Computer Graphics*, Bombay, Feb. 1993.

[7] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modelling the interaction of light between diffuse surfaces. *ACM Computer Graphics*, 18(3):213–222, July 1984.

[8] S. A. Green and D. J. Paddon. A non-shared memory multiprocessor architecture for large database problems. In M. Cosnard, M. H. Barton, and M. Vanneschi, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, Pisa, 1988.

[9] Inmos. *IMS T800 Architecture*. Inmos Technical Note 6, Inmos Ltd., Bristol, 1988.

[10] D. May and R. Shepherd. *Communicating Process Computers*. Inmos Technical Note 22, Inmos Ltd., Bristol, 1987.

The Evolution of the Meiko Computing Surface

Dave Wilson
Consultant Technologist
Meiko Scientific
650 Aztec West
Bristol BS12 4SD.
Email: davol@uk.co.meiko
Tel: 0454 616171

1 Introduction

At the time of writing, Meiko is setting its sights on the goal of producing a viable teraflop computer. While it is premature to reveal details of the development work currently underway at Meiko's Bristol Research headquarters, the architectural outlines of the machine have been established and are best understood in the light of the technological developments which have brought us to this point.

With the benefit of hindsight it is tempting to describe all earlier work as steps on the path to the realisation of an original grand design. Nevertheless, I believe the process can be more accurately described as one of evolution. The analogues of environmental pressures and threats to survival originate in the market, where engineers, scientists and corporations - devoid of any sentimental fascination with computer wizardry, require cost effective dependable platforms for highly demanding applications. As each new concept is put to the test, insight into the essential nature of the technology is enhanced, problems are redefined and unyielding lines mercilessly abandoned.

In the first part of this paper, I will describe how the Meiko Computing Surface has attained its present shape. The second section explains how Computing Surface technology is currently deployed in a wide range of application contexts, while the third and final section outlines some of the design decisions which are shaping Meiko's future products.

2 First generation - occam

The case for parallel, scalable computing platforms is no longer as controversial as it was in 1985 when Meiko first launched the Computing Surface. The design of the machine centred on the use of the Inmos T800 transputer as an integrated compute and communication engine. All processor boards incorporated elctronically configurable switch chips which permitted transputer links to be wired together in a wide variety of ways.

The programming model available to application programmers was occam and there was no operating system as such. Parallel programs were loaded into the machine from a host computer which supported a file system at run-time.

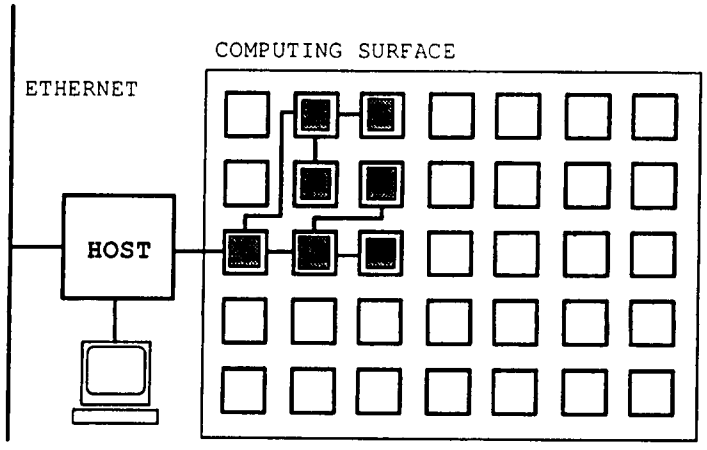


Fig. 1 Computing Surface - initial manifestation

Since the target machine for Occam programs was devoid of any sort of system service layer, it was common for much of the application programmers effort to be diverted into the provision of through routing and file serving

utilities. Worse still, these system components tended to get mixed up with the application program.

On the whole, occam proved unpopular with the majority of potential users of the Computing surface. This prompted Meiko to propose a scheme in which application programs developed in C or Fortran could be linked with a standard occam *harness* designed to implement one of a number of popular interconnection topologies. This method still suffered from the limitation that the program could only be run on a machine of a set size, wired in a specific way. Running the same program on a larger or smaller Computing Surface meant having to relink it with a modified harness. The problem was the lack of a suitable abstract machine.

While it is true that the first Computing Surfaces were efficient and direct in the provision of basic processing and communication resources, the relationship between program and machine was too close. As the promise of higher performance compute and communication technology began to emerge, the occam machine appeared awkward and restrictive.

M2VCS

In parallel with these concerns, Meiko introduced a set of system software components which permitted a single large Computing Surface to be viewed as a number of independent machines. This was in response to the fact that the Computing Surface was initially a single user machine. The system was known as M2VCS (Meiko Multiple Virtual Computing Surface) and introduced the concept of a *system spine* - a set of processors and links reserved for system use. In particular this consisted of support for the through-routing of file system requests and terminal traffic. Attached to the system spine were a number of *user domains*. For each such domain a mapping was defined between the unique numeric identifier of each processor and a local identifier accessible to the user. This enabled domains in the M2VCS machine to be configured and used just as if they each represented a single dedicated Computing Surface.

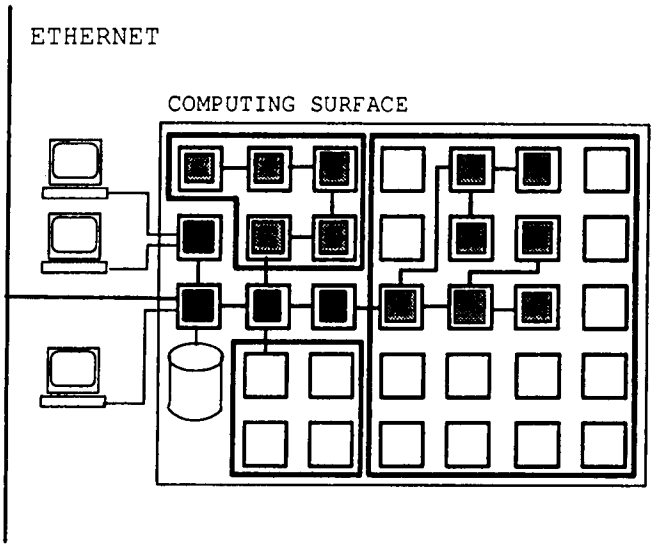


Fig. 2 Computing Surface - M2VCS

3 Second Generation - CSTools

Despite enabling the Computing Surface to be used as a shared machine, M2VCS did not address the problem of defining an abstract target machine for applications. This task was undertaken by a small team of programmers in 1988/89. The brief centred around three requirements:

- Programming must be straightforward and direct
- Programs must be immune to changes in the underlying hardware technology.
- Programs must be capable of exploiting the hardware efficiently.

The second and third of these requirements appeared at first sight to be mutually exclusive. In particular, the second requirement seemed to call for an abstract machine definition which by its very nature implied the introduction of an layer of system software capable of realising the abstraction on real hardware.

On further consideration however, we came to realise that the problem could be circumvented if the requirements of the parallel application program were assessed prior to loading the machine and the appropriate hardware and a minimal set of software modules assembled to provide the necessary support.

CSTools is a software system designed to achieve this objective - which we term Adaptive Abstraction. In fact, though CSTools can be used to realise a number of alternative abstract machines, the term Adaptive Abstraction describes the ability to realise a specific abstract machine in a manner which is optimised for a specific piece of software.

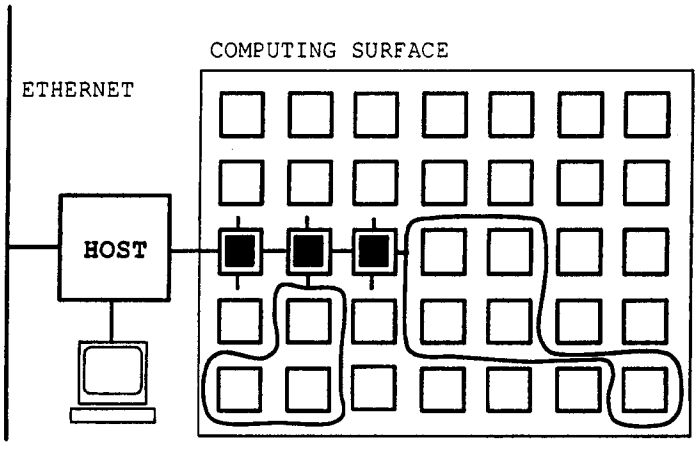


Fig. 3 Computing Surface - CSTools

At its simplest, the specification of a parallel program is made using a *parfile*. In the example below, `fred`, `jane` and `bill` are the names of transputer-executable files. Each is essentially a straightforward, single-threaded sequential program.

```
par
   processor 0 fred
   processor 1 jane
   processor 2 bill
endpar
```

CSTools interprets this as meaning that each of the three programs is to be run on a separate transputer and that any demand for file I/O is to satisfied by invoking requests on the host file system. *Parfiles* are read by an interpreter which is a particular example of a *CSBuild* program. Meiko supplies the CSBuild library as part of its CSTools product. Direct use of CSBuild is necessary in certain specialist applications where the capabilities of the *parfile* syntax is found to be inadequate.

The component processes which make up the parallel program are separately compiled and linked within a standard UNIX environment, using a transputer cross-compiler. The linker is designed to treat certain unresolved global variables as values to be fixed at a later phase of evaluation and rather than reporting them as undefined references they are appended to the executable file as a table of *imports*. The CSTools configuration utility (CSBuild) reads the table of imports attached to each executable file and interprets each import name as signifying a requirement for a specific service. Rather than simply appending additional library modules onto the executable image however, each service is implemented in accordance with a specific *resolution strategy* loosely incorporated into the configuration program. Thus the existence of an import called `_library_meikosRte` results from the fact that the process has linked a library module which makes a Remote Procedure Call to the fileserver: this in turn means that it must require access to the file system. The strategy for resolving this reference not only involves creating a local file system client but extends to the location of the fileserver itself and the the establishment of an efficient network path between them.

The CSTools configuration utility is developed and run in a familiar operating system environment on a sequential machine such as a Sun workstation. As it executes it builds up a detailed model of the parallel program AND the machine on which it will execute. It draws upon three sets of inputs:

- Recognisable application components.
- Knowledge of the machine on which the application is to run.
- A set of construction rules and the program components with which to implement them.

The second input is provided by the *Machine Manager* which runs on the host computer and discovers the characteristics and location of every hardware element in the machine. The Machine Manager is resposible later for establishing connections between transputer links and for the allocation and release of individual processors.

The most important system component included under the third heading is the Computing Surface Network (CSN). This supports a communication scheme based on addressed messages of arbitrary length between arbitrary pairs of processes.

The introduction of CSTools persuaded us that the management and exploitation of parallel computers made up of a heterogenous mix of processors need not be difficult with appropriate software tools. This intuition was put to the test with the introduction of the first i860 vector processor board - the MK086 - in 1989. Extending the capabilities of the CSTools model to incorporate a component comprising a vector processor, two T800 transputers and 8 hardware links involved 2-3 weeks work for one programmer.

On completion of the development work, parallel applications consisting of a mix of transputer and i860-based components could be easily run. In the example *parfile* above simply recompiling one or more of the sequential components for the i860 processor is sufficient to effect the transition to the new hardware.

The introduction of the Sparc processing element in 1990 and a second i860 board in 1991 have prompted similar revisions to the CSTools programming toolset. Paradoxically these developments have tended to reveal

a regularity in the underlying model rather than the opposite. As occam channel communications have been displaced by the CSN message passing network, differences in the run-time environments on similar processors have disappeared and the case for a uniform application interface common to all processors has to be reconsidered.

4 Third Generation - Concerto

The evolution of Computing Surface technology has led to the recognition of the generic processing element as the fundamental building block of the machine.

A generic processing element has four fundamental components - a compute processor for doing real work, a memory system, a communications engine and a control interface. Optionally, elements may have an interface to one or more I/O devices.

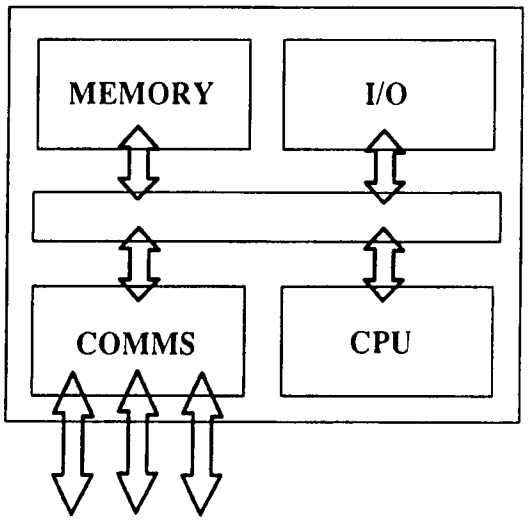


Fig. 5 Generic Processing Element

Compute processors are Sparcs, i860s or T800 transputers; each provides a different type of functionality. Sparc processors provide general purpose computational capabilities with memory protection and virtual memory. i860s deliver very high levels of performance for vectorisable applications. Transputers provide excellent support for rapid context switching and are significantly cheaper, but have lower levels of computational power and no memory protection.

In all current Computing Surfaces, the communications engine comprises one or more transputers. Where there is a dedicated compute processor, communications between the various processors comprising the element are closely coupled via access to a shared local memory.

Communications engines are currently made up of one or more transputers and enable elements to be connected in a wide range of topologies. General purpose arrays such as rings, trees, grids, torii and hypercubes are possible, as are application-specific topologies.

In order for processing element to work effectively the compute and communications functions must operate concurrently, otherwise the computer is said to be communication bound. In the first generation Computing Surface a single transputer fulfilled both compute and communication functions for each element. While communications were based on Occam channels this was not a serious deficiency as the transputer incorporates hardware link engines which do not severely impact CPU performance. With the introduction of the CSN message-passing network it was necessary to separate the two functions since network through-routing and address decoding draws upon the transputer CPU. Seen in this light the original transputer-based processing element is seen as the exception to the natural pattern since it loads two naturally separate functions onto a single processor.

The combination of SPARC processors providing a POSIX-compliant UNIX environment and i860 vector compute modules is referred to as the *Concerto* machine.

Standards

Computing Surfaces comprising advanced, multi-processor elements continue to represent cost effective computing platforms in a variety of specialised application areas. Nevertheless we recognise that the variety of sys-

tems, non-standard libraries and programming models offered to users has a uniformly discouraging effect on the market. Some of these differences represent significant intellectual divergence between their respective proponents, whilst others are simply the inevitable result of development in isolation. Where we believe the latter to be the case we have been eager to implement commonly used standards on top of the Computing Surface Network message passing layer. Thus, as a collaborator in the European Genesis project, Meiko has implemented the PARMACS communication macros on the Computing Surface.

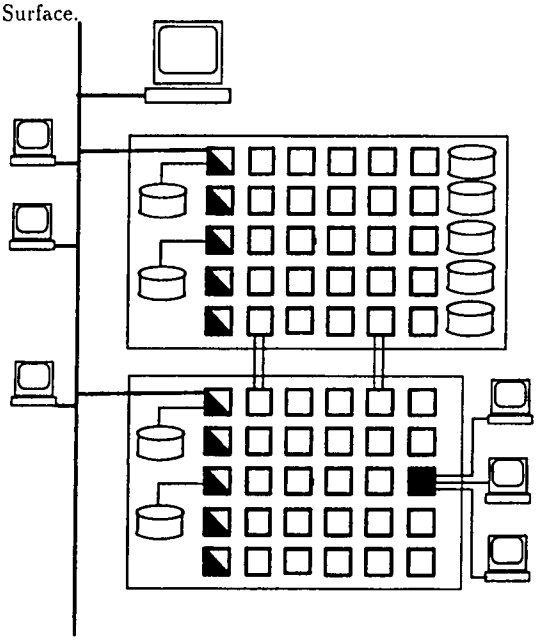


Fig. 6 The Concerto machine

In contrast to the work done as part of the Genesis project, an Intel hypercube compatibility library was implemented on the Computing Surface in the context of a highly competitive (and successful) bid. It is worth

decision support activities can be supported concurrently without the effects of one affecting the performance of the other. This feature is referred to as *firewalling*

Certain processors within the Computing Surface run instances of ORACLE. Each of the processing elements in the system uses the Computing Surface Network (CSN) interface to provide transparent access to remote resources. The remote resources include a Parallel Lock Manager and a parallel disc subsystem. The number of instances of ORACLE is increased in proportion to the CPU requirements of parsing and executing SQL statements.

In addition to running multi-instance ORACLE on the Computing Surface, it is possible to provide additional processing resource to support ORACLE applications. These applications benefit from high performance connections to the ORACLE instances over the CSN. Since SPARC modules can be used for these applications, many hundreds of ORACLE-based applications are immediately available to run on the Computing Surface.

The Relational Datacache is available as a database server for use in a networked environment, as an ORACLE accelerator for Digital VAX machines, or as an embedded component within a larger Meiko parallel computer system.

5 The next generation

In all the developments outlined above, the baisc machine architecture is still present at a low level. A machine composed entirely of i860 processors, each with 32 Mbytes of memory will run an occam application on its principal communications transputer provided the 2 Mbyte RAM available to the processor is sufficient for the program's needs.

Our strategy has been one of augmenting an initial set of simple and versatile components with increased computational capability in response to the needs of users, while the introduction of appropriate higher level services like message routing has greatly enhanced the ease with which the machine can be targetted by application programs.

Nevertheless, in order to deliver the sort of computational power required of a teraflop supercomputer it is necessary to return to a consideration of the underlying hardware architecture. At Meiko this exercise has been underway for a number of years.

In conclusion

The characteristics of the next generation Computing Surface can be summarised as follows:

- Greatly improved communications. This will be realised by direct hardware support for message routing. Furthermore, the network will be truly scalable. As the machine increases in size the number of components dedicated to the network must increase at a faster rate than the rate of increase in the number of processors.
- Use of commodity processors. As compiler technology matures instruction sets become less relevant. The economic argument for using standard commodity VLSI processor components is irresistable.
- Fault tolerance through redundancy. For machines of the scale envisaged (1000+ processing elements) fault tolerance becomes necessary for processors, I/O devices and the communication network.
- Continued support for the CSTools programming model.
- Support for virtual shared memory. Software and hardware environments are evolving along convergent paths.
- Support for standards - GPMIMD, UBIK/ASI.

NONPROCEDURAL LANGUAGE NORMA AND PROBLEMS OF ITS IMPLEMENTATION

A. N. Andrianov, K. N. Efimkin, I. B. Zadykhailo

Keldysh Institute of Applied Mathematics, Moscow
4, Miusskaya Sq.,Moscow, 125047, RUSSIA
Fax: (095) 972-07-37
E-mail: zadykhailo@applmat.msk.su

1. The specific features of the Norma language

The Norma language [1,2,3] is a tool aimed at automated solution of the mathematical physics problems on parallel computer systems.

The purpose of the Norma language is to eliminate the programming phase which is necessary to pass from computation formulas, derived by an application specialist to a computer program. There is no essential difference between computation formulas and the Norma program structures – these formulas, in fact, are an input for the Norma translation system.

The translator for the Norma language solves the problem of output program synthesis: the order of computations and its mode (parallel, vector or serial) are defined automatically. Sentences may be written in an arbitrary order – information relationships will be revealed and taken into account in the process of translation. The programming notions such as memory, loops or other control operators, are not used. The output program generation is done according to the target computer architecture.

In fact, the Norma program is a nonprocedural specification of problems to be solved. The mathematical problems connected with the synthesis of output program are solvable in case of the Norma language.

Norma has some specific features which largely determine both the process of programming and class of the problems to be solved. Among these features the following are most important.

1. The domains where values may be calculated are, in fact, regular orthogonal grids in the n-dimensional space (coordinates of the grid points are integers). In this manner, we impose a constraint on the space of values indices – each index may take values in a continuous range of integers.

2. The domains are static objects – their dimensions are determined by constant expressions whose values are calculated at the stage of translation.

3. The index expressions of the calculated variables have the form $i + k$, where i is the index name and k is an integer constant.

4. Norma is a language with a single assignment: any variable may take a value only once (a variable is defined on the grid only once at each point of the definition domain).

The first and the second constraints are directly connected with a class of problems which can be solved by using Norma. The problems requiring irregular, dynamic, etc. grids for their solution cannot be described in Norma.

The third constraint determines a class of calculational formulas used for solving the problem. From the practical point of view, this constraint does not seem to be strong because the index expressions of other forms are seldom encountered.

The fourth constraint is connected, in fact, with issues of memory allocation and saving, which are automatically settled at the translation stage.

1.1 The program structure

The program in Norma consist of one or several parts. For each part the localization rule is valid: all the identificators defined in the part are localized in this part; the concept of global variables is not used in the language. The only data exchange mechanism defined in the language implies interaction between formal and actual parameters (the parts may also execute data transfer through external files by using input and output). The part may be of three types – the main part, the simple part and the function. The side effect is impossible in the Norma language.

In the part body the declarations and the operators may be given in an arbitrary order. The former declare entities used in the program. The latter define the rules which determine the computations.

The following entities may be declared in Norma: the domains, the scalars, the variables defined on the grid, the grid parameters, the input and output, the names of external functions and parts.

Four kinds of operators are defined in Norma: the scalar operator, the operator **assume**, the part call and the iteration.

A domain, a variable and operator **assume** are key notions in Norma. Other notions are connected only with conveniences of programming.

1.2 Declaration of domains

The domain concept in Norma is equivalent to the space-time concept. Informally, the domain is a collection of integer sets $(i_1, i_2, \ldots, i_n)$, each of them corresponds to coordinates of a separate grid point. Declarative means enable a programmer to give the grids of relatively complicated configurations provided the point coordinates are integers.

Using the domains in Norma allows defining the entire grid of the problem or its segments, giving names of domains, executing simplest operations over them.

An example of declaring the two-dimensional domain obtained by the Cartesian product of the one-dimensional domains $c1$ and $c2$:

$$square : (c1 : (k = 1 \,..\, 5)\,;\, c2 : (l = 1 \,..\, 5)).$$

It is possible to specify a domain through modification of earlier declared domains. The subdomain modification may consist in either adding some points, or their removal or explicit redeclaration.

An idea of the domain modification by means of the logical conditions is as follows. The entire domain S under modification is divided into two non-intersecting subdomains. One of them contains points of the domain under modification, at which a value of the logical expression is true, while the other contains points at which this value is false. For example, let us give the domains:

$$s : ((i = 2 \,..\, n)\,;\, (j = 1 \,..\, m)).$$
$$sa, sb : s/x + y[i-1, j] - z[j+1] > 0.$$

The domain sa consists of the points of domain s, at which the condition is true, while sb consists of the points where the condition is false.

1.3 Declaration of variables

The scalar variables and the variables definied on the grid refer to arithmetic (calculations). Each arithmetic variable has its type: **integer**, **real** or **double precision**.

Each variable, defined on the grid, is connected with a certain domain. This domain defines the indices names which may be used in the index expressions when this variable is called. Calling the variable specified on grid, for example, in the arithmetic terms is the record in the form:

$$\langle variable\ name\rangle[\langle index\ expression\rangle, \ldots, \langle index\ expression\rangle].$$

The index expression has the form

$$\langle index\ name\rangle + \langle integer\ constant\rangle.$$

When an index is given by the constant we should specify what index direction the given constant belongs to, for example, $v[i = 1\,,\, j-1]$. If one index direction must be connected with the other by a relationship it is done explicitly. For example, diagonal elements of a matrix may be defined in the following way: $v[i\,,\, j = i]$.

In the Norma language it is not required to indicate all index expressions of the variable – the index expressions coinciding with the index direction name may be omitted.

1.4 Operator assume

The operator **assume** is used for assigning arithmetic values to the variables defined on the grid.

⟨*operator* **assume**⟩ ::= **for** ⟨*domain*⟩ **assume** ⟨*relation*⟩; . . . ; ⟨*relation*⟩.

⟨*relation*⟩ ::= ⟨*name of variable on the grid*⟩ = ⟨*arithmetic expression*⟩

Each relation specifies a rule for obtaining values of the variable in the left side by using values of the variables in the right side. In this case it is required to calculate values of the variable in the left side at all points of ⟨*domain*⟩ from operator title. Let all the domain points be known: by using their values we calculate the values of index expressions for the variables in the right side of the relation and obtain a set of desired arguments values for the right side. If at a certain time all the desired arguments are calculated for a certain point of ⟨*domain*⟩, then we may calculate values of the variable in the left part for this point. If not all arguments are determined then calculations at this point are currently impossible (it does not mean that the calculations are impossible in general).

For example, if the variable x is declared as

variable x **defined on** *matrix* : $((i = 1 \,..\, 10); (j = 1 \,..\, 20))$.

then the operator

for *matrix* / $i = 1 \,..\, 5,\ j = 1 \,..\, 5$ **assume** $x = 0$.

will lead to filling 25 components of variable x with zeros. The method of executing this operator is not fixed in the Norma language (it may be serial, parallel, vector, etc.).

Index form constraint (indices without bias) in the left side of the relation is not important because ⟨*domain*⟩ from the operator title allows defining rather complicated relations.

1.5 Other Norma program structures

The scalar operator in Norma is an analog of the assignment statement in the conventional programming languages, whose left-hand side contains a scalar variable while the right-hand side – a scalar arithmetic expression.

The Norma language provides definitions for standard arithmetic functions, mathematical functions and external functions of the user. The standard arithmetic functions are: *sqrt*(x), *abs*(x), *exp*(x), *sin*(x), etc. The mathematical functions are: *sum* (sum), *mult* (multiplication), *max* (maximum), *min* (minimum).

It is required to calculate

$$V_i = W_i + \sum_{j=1}^{m}(A_{ij} * X_j), \quad i = 1, \ldots, n$$

A fragment of this calculation in Norma is:

variable a **defined on** $oij : (oi : (i = 1 \,..\, n)\,; oj : (j = 1 \,..\, m))$.
variable v, w **defined on** oi.
variable x **defined on** oj.
  for oi **assume** $v = w + sum((oj)a * x)$.

The part call is, in fact, an extension of the relation concept from the operator **assume** since it allows simultaneously several results (values of different variables). The part call outside the operator **assume** is an extension of the scalar operator concept. The external function is the case of Norma structure part defined by the user.

A special language structure **iteration** makes it possible to define the iterational procedure of calculations.

2. The methods for Norma implementation on parallel computers

The problem of output program synthesis consists of two subproblems: discovery of natural parallelism in Norma program and mapping of this parallelism onto a target computer (class of computers).

The solution of the first one is founded on an analysis of the information dependencies graph. Information dependencies graph $G(V, E, P)$ is an oriented marked graph. The V is a set of graf nodes; each node correspond to a variable. If a certain variable is calculated in a few operators, then a few nodes correspond to it. The E is a set of edges reflecting the variables dependences: if the variable with indices $i_1, \ldots, i_n$ depend on the variable Y with indices $i_1 + a_1, \ldots, i_n + a_n$, then there exist the edge from X to Y with mark $a_1, \ldots, a_n$. The P is a set of all marks.

After the graph G has been constructed the search for maximum strongly connected subgraphs (MSCS) is performed. MSCS is strongly connected subgraph of graph G, which is not contained in any other strongly connected subgraph of graph G. When all MSCS of G have been picked out the reduction of G is done: all MSCS for special nodes are substituted.

In the reduced graph a partial ordering is performed: if an edge from V_i to V_j exists, then $V_i > V_j$.

By basing on the partial ordering results the parallel layer scheme [4] is constructed.

The computations corresponding to nodes of the same layer may be parallel and independent. The passage from a current layer to the next one is performed after all computations of the current layer have been performed.

Besides, a possibility of parallel computations in the separate nodes is discovered (for example, parallel computations at different points of the indices space). To do this, in some cases the problem of integer linear programming must be solved.

The methods for solving the mathematical problems arising at this stage are described in [5], along which the mapping of natural parallelism onto computers with shared memory architecture. These results are formulated in the form of constructive theorems about correctness of the Norma program (or solvability of the synthesis problem).

Our approach to the mapping of the natural parallelism onto a distributed computer system is based on the next computational model.

Distributed computer system is a matrix of processor elements (PE). Each PE has the coordinates denoted by (i, j). The coordinates (i, j) of PE can be recieved by the subroutine *getxy* (i, j).

Each PE (with the exception of boundary PEs) is connected with four neighbouring PEs.

Let the task, which is executed on one PE, be called a subtask. Subtask can exchange information with other subtasks with the help of two subroutines: *inmessage* and *outmessage*. The first one is intended to recieve data, the second one – to send data.

The parameters of these subroutines are:

$$outmessage\,(r, n, pri, prj, tag)\,;\ inmessage\,(r, n, pri, prj, tag)\,;$$

r is the first element of data array, which is sent (is recieved) ;

n is the size of data (in bytes), which is transferred;

pri and prj are relative coordinates of PE with which the data exchange is done,

tag is the name of message; this parameter establish an accordance between *inmessage* and *outmessage* : if a message has the tag T, then this message can be recieved by the subroutine *outmessage* with the same tag T.

The subroutine *inmessage* must wait for the data to be transferred from corresponding (in relative coordinates and tag) subroutine *outmessage* .

The result of translation from the Norma language onto a distributed computer system is a Fortran program for each PE. The case is possible when all PEs execute the same program.

In the process of translation two main problems must be solved: the data distribution and the control distribution.

Let the variables X^k, $k = 1, \ldots, p$ (p is the number of variables in the Norma programs) be defined on the domain:

$$(i = 1\ ..\ u_1^k;\ j = 1\ ..\ u_2^k;\ i_1 = 1\ ..\ v_1^k; \ldots;\ i_n = 1\ ..\ v_n^k),$$

Let the indices i and j of domains correspond to the indices i and j of the PE matrix, and the data distribution is made along these directions, the quantity

of PEs along the direction i is m, along j is n, $u^1 = \max_k u_1^k$, $u^2 = \max_k u_2^k$. Then in each PE the part of array X^k is placed. This part is defined on the domain

$$(i = 1 \,..\, \frac{u^1}{m}; j = 1 \,..\, \frac{u^2}{n}; i_1 = 1 \,..\, v_1^k; \ldots; i_n = 1 \,..\, v_n^k).$$

Such a data distribution can lead to inefficient execution when the variables in Norma program have quite different boundaries u_1^k (or u_2^k), $k = 1, \ldots, p$. If there are variables whith the boundary $u_1^k < \frac{u^1}{m}$ (or $u_2^k < \frac{u^2}{m}$), then most of PEs do not have these variables (the data distribution is "unbalanced"). If these variables are needed in a computations, then the mechanism *inmessage - outmessage* must be applied to recieve these variables.

It is possible also to distribute data in the "balanced" manner, then in each PE near equal quantity of array's elements are placed.

The control distribution is analogous to data distribution in a sense.

The range of operator's computation along some direction $i, i = l_r \,..\, u_r$, is divided into the subranges

$$l_1 = l_r \,..\, u_1, l_2 = u_1 + 1 \,..\, u_2, \ldots, l_k = u_{k-1} + 1 \,..\, u_k = u_r$$

by definite rules.

Therefore, PE compute the operator in the subrange $l_t \,..\, u_t$. It is possible that some PEs will not participate in computing this operator.

The variables defined on the grid with indices $i + k$, where k is integer, or $i = c$, c is constant, usually lead to a necessity of using in PEs the data exchange subroutines. To solve the problem of automatic generation of data exchange subroutines two tasks must be solved: finding the parameters for subroutines *inmessage* and *outmessage* ; finding the place in the output program where these subroutines must be called.

If the variables have the indices $i + k, k$ is integer, then the parameters *ipr* and *jpr* (relative coordinates of PEs) can be found in the process of index expressions analysis. In this case the parameters *ipr* and *jpr* can be found at the translation stage.

If the variables have the indices-constants, then the parameters *ipr* and *jpr* are the expressions depending on the coordinates of a current PE.

The task of placing the data exchange subroutines in the output programs is solved by using graph G. Let the values of variable X be calculated in subtask A and be needed in subtask B. With the help of graph G the last operator to calculate X in subtask A can be determined. The call of subroutine *outmessage* isplaced in subtask A directly after this operator. The call of corresponding subroutine *inmessage* is placed in subtask B directly before the operator, in which X is needed.

This method of placing subroutines enable us to execute the data exchange on the backgroud of computations.

The algorithms for solving the problems entitled above have been worked out by now.

3. State of the art and perspectives

Currently there are versions of the Norma compiler for which the target language is *FORTRAN-77* and vector-parallel version for which the target language is an extension of *FORTRAN-8X*, called *FORTRAN-VP*. The last version is oriented to a specific multiprocessor shared memory computer with a vector-pipeline architecture.

The version of compiler for distributed systems is under implementation now.

References

1. A. N. Andrianov, K. N. Efimkin, I. B. Zadykhailo. The Norma language. Prepr. Keldysh Inst. of Appl. Math., 165, 1985 (In Russian).
2. A. N. Andrianov, K. N. Efimkin, I. B. Zadykhailo. Nonprocedural Norma language and methods of its implementation. Algorithms and algorithmic languages. Moscow, Nauka, 1990 (In Russian).
3. A. N. Andrianov, K. N. Efimkin, I. B. Zadykhailo. Nonprocedural language for solving the problems of mathematical physics. Programmirovanie, N2, 1991 (In Russian).
4. V. V. Voevodin. Mathematical models and methods in parallel processes. Moscow, Nauka, 1986 (In Russian).
5. A. N. Andrianov. The synthesis of parallel and vector programs by the nonprocedural Norma specification. Ph. D. Thes., Moscow, 1990 (In Russian).

Author Index

Alexakhin, V., 309
Andrianov, A.N., 523
Arapov, D., 309
Argile, A., 246
Aristov, V., 417
Arrowsmith, E., 260
Bargiela, A., 21, 242
Barsky, A.B., 235
Baykal, N., 371
Beaumont, C., 217
Belov, C.G., 47
Benaini, A., 426
Bezdushny, A.N., 47
Biriukov, A., 328
Boronat, P., 217
Bourget, D, 38.
Cai, F.-F., 207
Campbell, N.W., 489
Cant, R., 177
Carcagno, L., 167
Chalmers,A.,280,489,498
Champeau, J., 217
Chepaite,Y., 187
De Michiel, M., 167
Dhaussy, P., 446
Dours, D., 167
Duval, D., 160
Efimkin, K.N, 523
Eglin-Leclerc, M.-C., 111
Erciyes, K., 271
Evstigneev. V., 31
Facca, R., 167
Filippi, V., 338
Filloque, J.M., 217
Gaissaryan, S., 76
Galligani, E., 406
Galligani, I., 435
Giavitto, J.-L., 102, 381
Gil ,M., 391
Gorodnichy, D.O., 134
Grossetie, J.C., 150
Hagmann, S., 143
Hartley, J., 21
Herrmann, B., 299
Huang, K., 197
Ilyin ,V.D., 357
Ivanov, V., 309
Julliand, J., 66, 111
Jun, Y.-K., 317
Kasyanov, V., 31
Katserov, S., 76
Khaletsky, D., 76
Kihl, H., 143
Koh, K., 317
Konovalov, N.A., 123
Krukov, V.A., 123, 309
Kuhn,D., 143
Laiymani, D., 426
Lastovetsky, A., 76
Ledovskih, I., 76
Lee, F.H., 10
Loli Piccolomini ,E., 435
Mahiout, A., 381
Maia, D.M., 270
Maier, J., 475
Mamedova, I., 417
Markhoff, B., 66
Martorell, X., 391
McMillin, B., 260
Meliokhin, V., 328
Menasche, M., 270
Michel,O., 102
Mihailov, S.N., 123
Navarro, N., 391
Ortega, R., 38
Paddon, D.J., 498
Perrin, G.-R., 299
Petrenko, A., 309
Pinti, A., 150
Pogrebtsov, A.A., 123
Pottier, B., 217
Proenca, A., 280
Redaelli, G.L., 338
Reznik, A.M., 134
Ribeiro Justo, G.R., 289
Rubini, S., 446
Ruggiero, V., 406, 435
Sakho, I., 227
Sansonnet, J.-P., 102, 381
Santos, L.P.,280
Sautet, B., 167
Serebriakov, V.A., 47
Shilov, V.V., 235
Silva, J.G., 347
Silva ,L.M., 347
Spletukhov, J., 366
Stiles, G.S., 10
Swaminathan, V., 10
Tarasenko, L.G., 84
Thomas, B.T., 489
Torgashev, V.A., 93
Tsaryov, I.V., 93
Urban, J.P., 143
Vashakidze, A., 309
Voevodin, V.V., 400
Vollmer, J., 463
Wilson, D., 511
Wirtz, G., 56
Wu, J., 197
Zadykhaylo, I.B, 523
Zama, F., 435