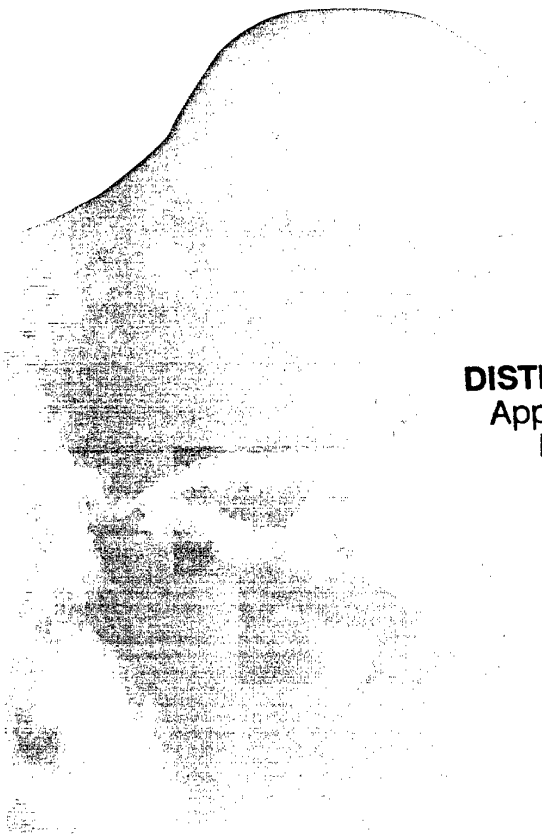
VECPAR

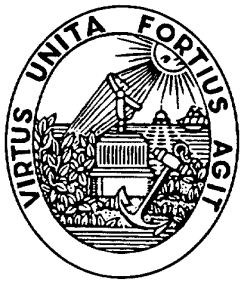
and

parallel processing



Proceedings

Part III (June 23)



Faculdade de Engenharia
da Universidade do Porto

2000 June, 21 22 23

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	12 January 2001	Conference Proceedings

4. TITLE AND SUBTITLE

VECPAR - 4th International Meeting on Vector and Parallel Processing

Part III

5. FUNDING NUMBERS

F61775-00-WF071

6. AUTHOR(S)

Conference Committee

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

FEUP-FACULDADE DE ENGENHARIA DA Universidade do Porto
RUA DOS BRAGAS
Porto 4050-123
Portugal

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0200

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CSP 00-5071

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

A

13. ABSTRACT (Maximum 200 words)

The Final Proceedings for VECPAR - 4th International Meeting on Vector and Parallel Processing, 21 June 2000 - 23 June 2000, an interdisciplinary conference covering topics in all areas of vector, parallel and distributed computing applied to a broad range of research disciplines with a focus on engineering. The principal topics include: Cellular Automata, Computational Fluid Dynamics, Crash and Structural Analysis, Data Warehousing and Data Mining, Distributed Computing and Operating Systems, Fault Tolerant Systems, Imaging and Graphics, Interconnection Networks, Languages and Tools, Numerical Methods, Parallel and Distributed Algorithms, Real-time and Embedded Systems, Reconfigurable Systems, Linear Algebra Algorithms and Software for Large Scientific Problems, Computer Organization, Image Analysis and Synthesis, and Nonlinear Problems.

14. SUBJECT TERMS

EOARD, Modelling & Simulation, Parallel Computing, Distributed Computing

15. NUMBER OF PAGES
Three volumes:
1016 pages total
(plus TOC, and front matter)

16. PRICE CODE

N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

VECPAR'2000

4rd International Meeting on
Vector and Parallel Processing

2000, June 21-23

Conference Proceedings
Part III
(Friday, June 23)

FEUP
Faculdade de Engenharia
da Universidade do Porto

Preface

This book is part of a 3-volume set with the written versions of all invited talks, papers and posters presented at *VECPAR'2000 - 4th International Meeting on Vector and Parallel Processing.*

The Preface and the Table of Contents are identical in all 3 volumes (one for each day of the conference), numbered in sequence. Papers are grouped according to the session where they were presented.

The conference programme added up to a total of 6 plenary and 20 parallel sessions, comprising 6 invited talks, 66 papers and 11 posters.

It is our great pleasure to express our gratitude to all people that helped us during the preparation of this event. The expertise provided by the Scientific Committee was crucial in the selection of more than 100 abstracts submitted for possible presentation.

Even at the risk of forgetting some people, we would like to express our gratitude to the following people, whose collaboration went well beyond the call of duty. Fernando Jorge and Vítor Carvalho, for creation and maintenance of the conference web page; Alice Silva for the secretarial work; Dr. Jaime Villate for his assistance in organisational matters; and Nuno Sousa and Alberto Mota, for authoring the procedure for abstract submission via web.

Porto, June 2000 *The Organising and Scientific Committee Chairs*

VECPAR'2000 was held at Fundação Dr. António Cupertino de Miranda, in Porto (Portugal), from 21 to 23 June, 2000.

VECPAR is a series of conferences, on vector and parallel computing organised by the Faculty of Engineering of the University of Porto (FEUP) since 1993.

Committees

Organising Committee

A. Augusto de Sousa (Chair)
José Couto Marques (Co-chair)
José Magalhães Cruz

Local Advisory Committee

Carlos Costa
Raimundo Delgado
José Marques dos Santos
Fernando Nunes Ferreira
Lígia Ribeiro
José Silva Matos
Paulo Tavares de Castro
Raul Vidal

Scientific Committee

J. Palma (Chair)	Univ. do Porto, Portugal
J. Dongarra (Co-chair)	Univ. of Tennessee and Oak Ridge National Lab., USA
V. Hernandez (Co-chair)	Univ. Politécnica de Valencia, Spain
P. Amestoy	ENSEEIHT-IRIT, Toulouse, France
T. Barth	NASA Ames Research Center, USA
A. Campilho	Univ. do Porto, Portugal
G. Candler	Univ. of Minnesota, USA
A. Chalmers	Univ. of Bristol, England
B. Chapman	Univ. of Southampton, England
A. Coutinho	Univ. Federal do Rio de Janeiro, Brazil
J. C. Cunha	Univ. Nova de Lisboa, Portugal
F. d'Almeida	Univ. do Porto, Portugal
M. Daydé	ENSEEIHT-IRIT, Toulouse, France
J. Dekeyser	Univ. des Sciences et Technologies, Lille, France
P. Devloo	Univ. Estadual de Campinas (UNICAMP), Brazil
J. Duarte	Univ. do Porto, Portugal
I. Duff	Rutherford Appleton Lab., England, and CERFACS, France
D. Falcão	Univ. Federal do Rio de Janeiro, Brazil
J. Fortes	Purdue Univ., USA
S. Gama	Univ. do Porto, Portugal
M. Giles	Univ. of Oxford, England
L. Giraud	CERFACS, France
G. Golub	Stanford Univ., USA
D. Heermann	Univ. Heidelberg,Germany
W. Janke	Univ. of Leipzig, Germany
M. Kamel	Univ. of Waterloo, Canada
M.-T. Kechadi	Univ. College Dublin, Ireland
D. Knight	Rutgers-State Univ. of New Jersey, USA
V. Kumar	Univ. of Minnesota, USA
R. Lohner	George Mason Univ., USA
E. Luque	Univ. Autónoma de Barcelona, Spain
J. Macedo	Univ. do Porto, Portugal
P. Marquet	Univ. des Sciences et Technologies, Lille, France
P. de Miguel	Univ. Politécnica de Madrid, Spain
F. Moura	Univ. do Minho, Portugal
E. Oñate	Univ. Politécnica de Catalunya, Spain
A. Padilha	Univ. do Porto, Portugal
R. Pandey	Univ. of Southern Mississipi, USA
M. Peric	Technische Univ. Hamburg-Harrurg, Germany
T. Priol	IRISA/INRIA, France
R. Ralha	Univ. do Minho, Portugal
M. Ruano	Univ. do Algarve, Portugal
D. Ruiz	ENSEEIHT-IRIT, Toulouse, France
H. Ruskin	Dublin City Univ., Ireland
J. G. Silva	Univ. de Coimbra, Portugal
F. Tirado	Univ. Complutense, Spain
B. Tourancheau	Univ. Claude Bernard de Lyon, France
M. Valero	Univ. Politécnica de Catalunya, Spain
A. van der Steen	Utrecht Univ., The Netherlands
J. Vuillemin	École Normale Superieure, Paris, France
J.-S. Wang	National Univ. of Singapore, Singapore
P. Watson	Univ. of Newscastle, England
P. Welch	Univ. of Kent at Canterbury, England
E. Zapata	Univ. de Malaga, Spain

Sponsoring Organisations

The Organising Committee is very grateful to all sponsoring organisations for their support:

FEUP - Faculdade de Engenharia da Universidade do Porto
UP - Universidade do Porto

CMP - Câmara Municipal do Porto
EOARD - European Office of Aerospace Research and Development
FACM - Fundação Dr. António Cupertino de Miranda
FCCN - Fundação para a Computação Científica Nacional
FCG - Fundação Calouste Gulbenkian
FCT - Fundação para a Ciência e a Tecnologia
FLAD - Fundação Luso-Americana para o Desenvolvimento
ICCTI/BC - Inst. de Cooperação Científica e Tecnológica Internacional/British Council
INESC Porto - Instituto de Engenharia de Sistemas e de Computadores do Porto
OE - Ordem dos Engenheiros
Porto Convention Bureau

ALCATEL
CISCO Systems
COMPAQ
MICROSOFT
NEC European Supercomputer Systems
NORTEL Networks
SIEMENS

PART I (June 21, Wednesday)

Session 8: Linear Algebra

June 22, Thursday (Room A, 9:00-10:20)

Invited Talk

(June 22, Thursday, Auditorium, 10:50-11:50)

Posters

The poster session will be held simultaneously with the Industrial session.
(June 22, Thursday, Entrance Hall, 11:50-12:50)

A Component-Based Stiff ODE Solver on a Cluster of Computers

J.M. Mantas Ruiz[1] and J. Ortega Lopera[2]

[1] Dpto. Lenguajes y Sistemas Informáticos. E.T.S. Ingeniería Informática.
Univ. Granada. Avda. Andalucía 38, 18071 - Granada, Spain.
jmmantas@ugr.es
[2] Dpto. Arquitectura y Tecnología de Computadores. E.T.S. Ingeniería Informática.
Univ. Granada. Avda. Andalucía 38, 18071 - Granada, Spain.
jortega@atc.ugr.es

Abstract. The most outstanding aspects of a general-purpose solver of stiff Ordinary Differential Equations (ODEs) for a cluster of computers are described. The numerical method used consists of adjusting an inner iteration process to solve the linear systems which arise when an Implicit Runge-Kutta (IRK) method is applied. The numerical scheme attains a high degree of task parallelism by decoupling several calculations. In order to exploit the data parallelism arising from linear algebra, software components of parallel linear algebra libraries have been used and a component-based methodological approach to derive parallel programs is followed. This approach is especially suitable to exploit the multilevel parallelism presented by this type of method and extends the TwoL proposal, made by Rauber and Rünger, by incorporating performance polymorphism. Several numerical experiments performed with different test problems reveal satisfactory runtime results with respect to one of the most advanced sequential stiff ODE solvers.

1 Introduction

The Ordinary Differential Equations (ODEs) arising from the modelling process may take different forms. One important formulation is that of the Initial Value Problem (IVP) for ODE, which can be formulated as:

Given a function $f : \Re \times \Re^d \rightarrow \Re^d$, find the function $y : \Re \rightarrow \Re^d$ that fulfils:

$$y'(t) = f(t, y), \qquad y(t_0) = y_0 \in \Re^d, \qquad t \in [t_0, t_f] \ . \tag{1}$$

Numerical methods for integrating IVPs generally work in a *step-by-step* manner: the interval $[t_0, t_f]$ is divided into subintervals $[t_0, t_1], [t_1, t_2], ..., [t_{N-1}, t_N]$ where $t_N = t_f$, and approximations $y_1, y_2, \ldots, y_N$ for the solution at the end of each interval are computed in a so-called *integration step*.

The *stiff* IVPs are an important class of IVPs that impose severe stability demands on the numerical methods of solution [9]. The solution of large stiff ODE systems is indispensable for modelling a wide variety of time-dependent processes in science and engineering [4, 9] (pollution models, chemical kinetics, biological

modelling, control systems, etc.). In order to solve stiff systems efficiently, it is necessary to use stable implicit methods [4, 9]. These methods demand a great deal of computing power, which can be achieved by using efficient parallel algorithms on Distributed-Memory Parallel Machines (DMPMs).

Because of the nature of the implicit methods, there is a strong connection between algorithms to solve stiff IVPs and linear algebra techniques. This fact reveals that the methods to solve stiff ODEs exhibit two levels of potential parallelism: *task parallelism*, owing to the fact that these methods can be decomposed into submethods which can be executed independently by disjointed groups of processors, and *data parallelism*, because the most basic submethods of the decomposition are typically linear algebra computations which are susceptible to parallelization following a SPMD style.

There exist stiff ODE solvers which run on multicomputers. One of the most outstanding solvers is the *ParSODES* package [2], which implements IRK methods using MPI [8] and gains a typical speedup 3 to 5 over state-of-the-art sequential stiff solvers on an IBM SP2. However, this solver does not suitably exploit the potential data parallelism due to the linear algebra operations.

The GSPMD [1] (*Group Single Program Multiple Data*) [11] programming model is specially suitable to exploit the multilevel parallelism of these applications on DMPMs. In [11], a methodological approach for the derivation of GSPMD programs from existing SPMD modules is presented. This approach, termed *TwoL*, has also been applied to the implementation of ODE integrators on multicomputers. However, this approach does not explicitly support maintenance and selection among multiple implementations of a submethod. This characteristic, the so-called *performance polymorphism*, is very desirable to improve flexibility in performance tuning during the design of a GSPMD program.

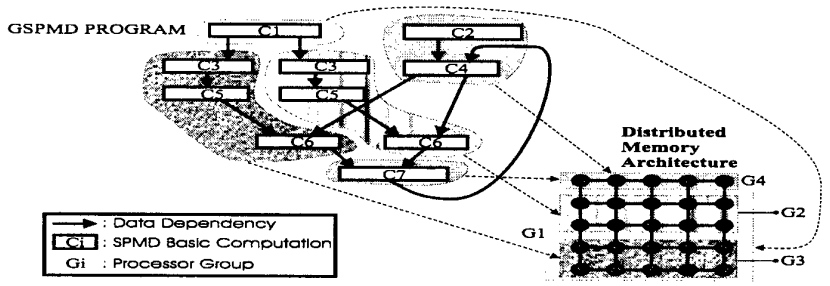


Fig. 1. Parallel program implemented in a GSPMD style

The interest of this work focuses on the component-based development of GSPMD software for the numerical solution of large stiff ODE systems on clus-

[1] In a GSPMD computation, several independent subprograms are executed by independent groups of processors in parallel.

ters of computers. In particular, we examine the parallelization of the class of one-step Radau IIA IRK methods [9]. These methods combine high accuracy with excellent stability properties and it is possible to derive parallel numerical schemes of this class with a lot of potential parallelism in the abovementioned two levels. In order to maintain flexibility when deriving parallel implementations of ODE solution methods, we propose an extension of the TwoL approach which has explicit constructs for performance polymorphism and facilitates the appropiate reuse of modules of parallel linear algebra libraries [7, 5, 13]. On the basis of our methodological proposal, we derive a distributed implementation of an advanced numerical method to integrate stiff ODEs for a PC cluster. This stiff ODE solver exploits both levels of potential parallelism of the numerical method, and makes effective use of existing parallel linear algebra software components. The speedup results obtained show that it is possible to derive efficient cluster implementations of an advanced stiff ODE solver by combining previously existing SPMD parallel linear algebra components.

The numerical algorithm to be implemented is obtained as a result of several decisions described in section 2. Section 3 introduces the TwoL approach [11]. An extension of TwoL that supports performance polymorphism is followed to implement the algorithm on a PC cluster in section 4. Section 5 presents the experimental results obtained by executing the solver for several test problems with different dimensions. Conclusions are drawn in section 6.

2 The Newton-PILSRK Method

One of the most interesting approaches to the parallel solution of ODEs consists of modifying sequential algorithms in order to exploit the parallelism within an integration step (*parallelism across the method*) [4, 1]. An advanced numerical method to achieve parallelism across an IRK method is described in [14]. This method, the so-called *Newton-PILSRK* method, is based on the inclusion of a Parallel Iterative Linear Solver for the Newton systems that arise when an IRK method is implemented. An implementation of this numerical scheme on a four-processor shared-memory CRAY C98/4256 [6] obtains a speedup of 2.4 to 3.1 with respect to one of the most advanced sequential solvers, RADAU5 [9]. We will derive an efficient distributed implementation for a PC cluster.

An s-stage IRK method can be conveniently represented by two vectors $c, b \in \Re^s$ and a full matrix $A \in \Re^{s \times s}$. When this is applied to an IVP-ODE given by (1), the method takes the form:

$$y_n = y_{n-1} + h_n (b^T \otimes I_d) F(Y_n), \quad Y_n = (\mathbb{1} \otimes I_d) y_{n-1} + h_n (A \otimes I_d) F(Y_n), \; n = 1, .., N \quad (2)$$

Here h_n is the *stepsize*, i.e., the length of the n-th integration subinterval $[t_{n-1}, t_n]$. $Y_n \in \Re^{sd}$ is the so-called *stage vector* defined as:

$$Y_n = (Y_{n,1}^T, Y_{n,2}^T, ..., Y_{n,s}^T)^T \text{ where } Y_{n,i} \approx y(t_{n-1} + c_i h_n) \in \Re^d, \quad i = 1, ...s \; .$$

The symbol $\otimes$ denotes the direct product of matrices. $\mathbb{1}$ stands for the s-dimensional unit vector $(1, ..., 1)^T$, I_d denotes the identity matrix of dimension $d \times d$ and $F(Y_n)$ means the componentwise f-evaluation of Y_n, i.e.

$$F(Y_n) = (f(Y_{n,1})^T, \ldots, f(Y_{n,s})^T)^T \in \Re^{sd} \ .$$

Therefore, we have to solve, in every integration step, an sd-dimensional system of nonlinear equations. This system is usually solved by the modified Newton iteration that yields a sequence $Y_n^{(0)}$, $Y_n^{(1)}$, $Y_n^{(2)}$, ... defined by

$$(I_{sd} - A \otimes h_n J_n)(Y_n^{(j)} - Y_n^{(j-1)}) = -R(Y_n^{(j-1)}), \quad j = 1, 2, ..., m \quad (3)$$
$$R(X) = X - (E \otimes I_d)Y_{n-1} - h_n(A \otimes I_d)F(X), \ \forall X \in \Re^{sd}, \ E = \left[0 \cdots 0 \ \mathbb{1}\right]$$

where $Y_n^{(0)} = P(Y_{n-1})$ ($P(\cdot)$ denotes a predictor operator), and the matrix J_n is an approximation to the Jacobian of f in (y_{n-1}, t_{n-1}). Process (3) must be applied as many times as needed to make $Y_n^{(m)}$ sufficiently close to the true solution of (2). The definitive approximation of $y(t_n)$ wold be $y_n = Y_{n,s}^{(m)}$.

A linear system of dimension sd arises in (3). In order to tackle this computational demand using parallel processing, a relevant numerical scheme is proposed in [14]. This approach is based on applying an iterative solver to the Newton systems in (3). This Parallel Iterative Linear System Solver for IRK methods (PILSRK method) is defined by: $v = 1, .., r$

$$(I_{sd} - T \otimes h_n J_n)\, \Delta X^{(j,v)} = (I_{sd} - L \otimes h_n J_n)\left(X^{(j-1)} - X^{(j,v-1)}\right) + R_n, \quad (4)$$

where:

- $R_n = h_n(S^{-1}A \otimes I_d)F(Y^{(j-1)}) + (S^{-1}E \otimes I_d)Y_{n-1} - X^{(j-1)}$,
- $X^{(0)} = (S^{-1} \otimes I_d)P(Y_{n-1})$,
- $\Delta X^{(j,v)} = X^{(j,v)} - X^{(j,v-1)}$, $X^{(j,0)} = X^{(j-1)}$, $X^{(j)} = X^{(j,r)}$, $Y_n = (S \otimes I_d)X^{(m)}$.

The matrix $L = S^{-1}AS \in \Re^{s \times s}$ is a block diagonal matrix with two blocks, $T \in \Re^{s \times s}$ is a diagonal matrix with positive entries and $S \in \Re^{s \times s}$ is a real, nonsingular matrix. Therefore, we can now identify two main sources of task parallelism in the method:

a) The solution of a linear system of dimension sd with coefficient matrix $I_{sd} - A \otimes h_n J_n$ can be replaced with the solution of s independent systems of dimension d, with matrix $I_d - diag_i(T)h_n J_n$.
b) Several computations, where the matrix L appears, can be decoupled into two independent computations on different blocks of L.

Several decisions have been taken to obtain a particular numerical algorithm. We have concentrated on the superconvergent Radau IIA IRK method with $s = 4$ stages [9] because it is very suitable for most real applications. We consider $r = 2$ in (4) because it is sufficient for convergence [14] and the value of m is determined dynamically. To obtain an initial approximation of the stage vector in every step, we use the extrapolation predictor defined by $Y_n^{(0)} = (P \otimes I_d)Y_{n-1}$, $P \in \Re^{s \times s}$ [6]. The resulting algorithm is presented below.

Algorithm

$Y_0 = (\mathbb{1} \otimes I_d) y_0$
for $n = 1, ..., N$ { $J_n \simeq \frac{\partial f}{\partial y}(y_{n-1}, t_{n-1})$; (N is the number of steps)
parfor $i = 1, 2$ { $M_i = I_{2d} - Block_i(L) \otimes h_n J_n$ }
parfor $i = 1, ..., 4$ { $LU_i = I_d - diag_i(T) h_n J_n$ }
$W = (S^{-1}E \otimes I_d) Y_{n-1}$; ${Y_n}^{(0)} = (P \otimes I_d) Y_{n-1}$; $X^{(0)} = (S^{-1}P \otimes I_d) Y_{n-1}$;
for $j = 1, 2, ..., m$ { $R = h_n (S^{-1}A \otimes I_d) F({Y_n}^{(j-1)}) + W - X^{(j-1)}$;
parfor $i = 1, ..., 4$ { $\Delta X_{i,d}^{(j)} = LU_i^{-1} R_{i,d}$ }
parfor $i = 1, 2$ { $R_{i,2d} = R_{i,2d} - M_i \Delta X_{i,2d}^{(j)}$ }
parfor $i = 1, ..., 4$ { $\Delta X_{i,d}^{(j)} = \Delta X_{i,d}^{(j)} + LU_i^{-1} R_{i,d}$ }
$X^{(j)} = X^{(j)} + \Delta X^{(j)}$; ${Y_n}^{(j)} = {Y_n}^{(j-1)} + (S \otimes I_d) \Delta X^{(j)}$ }
$Y_n = {Y_n}^{(m)}$; y_n = Last d-block in Y_n } $y_f = y_N$;

Notation

$Block_i(L)$= i-th 2×2-block of L
$R_{i,d}$= i-th d-block of $R = (R_{1,d}^T, ..., R_{4,d}^T)^T \in \Re^{sd}$
$R_{i,2d}$= i-th $2d$-block of $R = (R_{1,2d}^T, R_{2,2d}^T)^T \in \Re^{sd}$

3 Component-Based Derivation of GSPMD Programs: the TwoL Approach

In [11], a parallel programming methodology to derive structured parallel implementations of numerical methods by using previously existing SPMD modules was presented. This approach, termed *TwoL*, is based on the GSPMD model and makes it possible to exploit the multilevel parallelism exhibited by numerous numerical methods. In TwoL, the derivation of a parallel program for a particular DMPM is carried out starting from a mathematical description of the numerical method to be implemented, followed by three stages:

1. **Definition of the Module Specification for the numerical method.** In this stage, the programmer must specify the hierarchical structure of the numerical method by composing methods which are effected by modules. The modules which appear in the decomposition can be *basic modules* or *composed modules*. The **basic modules** represent regular algebraic computations on homogeneous data structures that work internally following a SPMD style according to a certain data distribution. These modules hide the potential data parallelism of the submethods. The **composed modules** are structured combinations of basic modules. The data dependencies between related modules are expressed by constructors of sequential composition and constructors of concurrent composition are used to express the possibility of parallel execution. As a result, we obtain a module specification which expresses the maximum degree of task parallelism of the method.
2. **Deriving the Parallel Frame Program.** A parallel frame program is a module specification augmented with several parallel design decisions which attempt to adapt the module specification to a particular architecture and problem in order to achieve good performance results. These decisions are briefly described below:

- *Scheduling*: The execution order of modules without data dependencies (concurrent composition) is determined. These modules can be executed in parallel on different groups of processors or consecutively on the same group.
- *Load Balancing*: The number of processors used to execute each basic module is determined.
- *Selection of the data distributions for the basic modules:* If the basic modules follow parameterized data distributions, then the parameters for these modules that result in an optimal runtime solution must be selected taking into account the cost associated with data redistributions.

3. **Obtaining the parallel implementation**. These decisions must be translated into a message-passing program for the target parallel machine.

4 Incorporating Performance Polymorphism into TwoL: Application to the Derivation of a Stiff ODE Solver

When deriving GSPMD implementations for complex numerical applications like stiff ODE integration, the functionality of a submethod can be implemented by using different parallel numerical algorithms which differ only in their performance characteristics. Ideally, several implementations for each submethod are maintained and it would be possible to select the best implementation for each context without having to deal with the details of the respective implementation. This desirable characteristic is called *performance polymorphism.* The selection of the best implementation depends fundamentally on the target machine, the problem to be solved, and the data distribution scheme of the application.

One approach to achieve performance polymorphism is based on the use of *poly-algorithms* [13]. However, this approach involves the design of efficient heuristic decision routines to select the best algorithm at runtime and the incorporation of new algorithms requires the modification of these routines.

The TwoL methodological approach does not have explicit constructs for performance polymorphism. On the basis of a component-based development approach [12], a proposal to integrate performance polymorphism within the TwoL framework will be presented. The central idea of our methodological proposal is the construction of a basic module by linking two different components: *concepts* and *realizations.*

A **concept** formally defines the functional properties of a basic module, including all functional aspects of the module interface. The operation signature must be presented together with a description of module functionality. A concept that denotes the computation of an approximation to the Jacobian of a function f in a point $(t, y) \in \Re^{d+1}$ is presented in figure 2.

A **realization** encapsulates the information related to an implementation of a basic module and has two parts: the *header* and the *body.* The *header* is a client-visible part which describes all performance aspects that the customer programmer needs to know in order to use the component, such as the data distributions for the matrices (parameterized distributions are generally used), and

runtime formulas, obtained from a simplified DMPM model. The runtime formulas allow the user to select the best realization to implement the functionality of a given concept according to several parameters; these describe the data distributions, the target machine (number of processors N, per word communication cost t_w, message startup time t_s, etc.), and the problem size. The *body* is the implementation code of the module which can be obtained from linear algebra parallel libraries [5, 3] or can be defined by the user.

Each concept can have several associated realizations. The explicit distinction between concept and realization as separate components allows us to select the implementation that offers the best performance, for a given functionality specification. Logically, this selection must include the establishment of the data distribution parameters which appear in the realization header.

Two different realizations for the concept $MJacobian$ are shown in figure 2. The realization $Block\_MJacobian$ works internally computing one block of the Jacobian matrix in each processor and the realization $BCyclic\_MJacobian$ works according to a more general block-cyclic distribution of the matrix J_n.

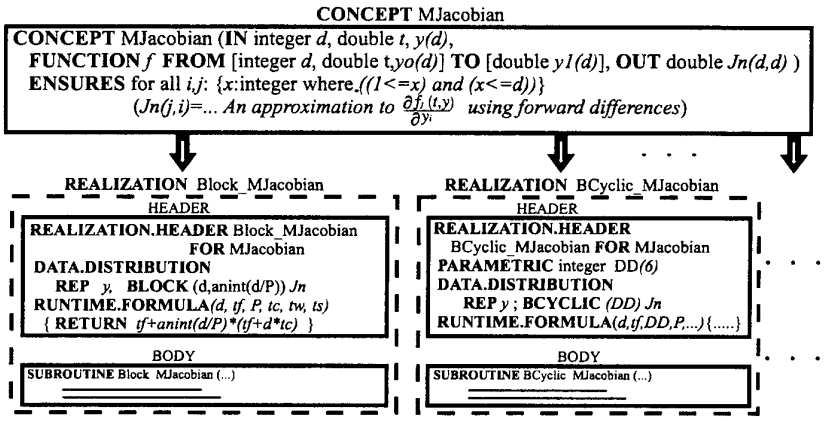


Fig. 2. Several realizations for the concept $MJacobian$

Taking into account the two types of components required to build a basic module, an extension of the TwoL approach will be proposed. In our proposal, three stages are also used, as shown in figure 3. This proposal will be used to derive an implementation of the Newton-PILSRK method for a PC cluster.

4.1 Concept Composition

When the mathematical description of the numerical method is available, several concepts must be selected in order to achieve the functionality of the submethods.

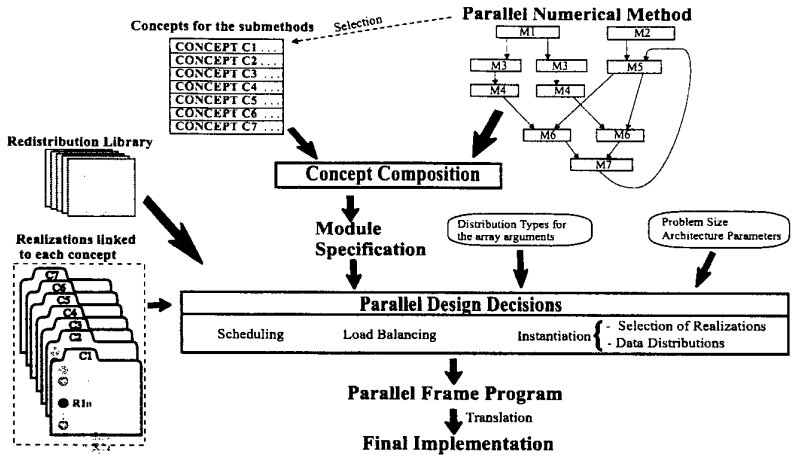


Fig. 3. Derivation of Parallel Programs in the proposed extension of TwoL

In order to describe the functionality of the parallel numerical algorithm presented in section 2, we have selected the following concepts:

Basic Concept	Brief functionality description	
E1dpIdV(s,d,y,Y)	$Y \longleftarrow (\mathbb{1} \otimes I_d)y$	$y \in \Re^d$, $Y \in \Re^{sd}$
Vcopy(n,X,Y)	$Y \longleftarrow X$	$X, Y \in \Re^n$
Mcopy(m,n,A,B)	$B \longleftarrow A$	$A, B \in \Re^{m \times n}$
Vsum(n,a,X,Y)	$Y \longleftarrow aX + Y$	$X, Y \in \Re^n$, $a \in \Re$
MVproduct(m,n,a,A,X,b,Y)	$Y \longleftarrow aAX + bY$	$X, Y \in \Re^n$, $a, b \in \Re$, $A, B \in \Re^{m \times n}$
LUdecomp(m,n,A,Ip)	LU Factorization of $A \in \Re^{m \times n}$	
SolveSystem(n,A,Ip,X)	$X \longleftarrow A^{-1}X$, assumes LUdecomp(n,n,A,Ip)	
MJacobian(d,t,y,f,Jn)	$Jn \approx \frac{\partial f}{\partial y}(y,t)$	$y \in \Re^d$, $t \in \Re$, $J_n \in \Re^{d \times d}$
MsumId(n,a,A)	$A \longleftarrow I_n + aA$	$A \in \Re^{n \times n}$, $a \in \Re$
Mdirectproduct(m,n,r,s,A,B,C)	$C \longleftarrow A \otimes B$	$A \in \Re^{m \times n}$, $B \in \Re^{r \times s}$, $C \in \Re^{mr \times ns}$
MdirectpIdV(s,d,A,Y)	$Y \longleftarrow (A \otimes I_d)Y$	$A \in \Re^{s \times s}$, $Y \in \Re^{sd}$
VFeval(s,d,c,t,h,f,Y)	$Y \longleftarrow F_f(Y)$,	$c \in \Re^s$, $Y \in \Re^{sd}$, $t, h \in \Re$
InitVectors (s,d,A,P,Y_0,W,Y,X)	$[W,Y,X] \longleftarrow [(A \otimes I_d)Y_0, \quad (P \otimes I_d)Y_0, \quad (AP \otimes I_d)Y_0]$	$A, P \in \Re^{s \times s}, Y_0, W, Y, X \in \Re^{sd}$
ConvergenceCtrl (...)	Control the convergence of the Newton Process	
ErrorCtrl (...)	Control the local error and adjust the stepsize	

These concepts must be combined by using constructors of sequential and concurrent composition to obtain a module specification. In contrast to TwoL module specification, in this case the components which are combined are not basic modules, that denote particular implementations of submethods, but are fundamentally concepts that denote only the functionality of the submethods. These concepts are substituted by specific implementations in the next stage, when all the design decisions that influence performance are taken.

The module specification can also contain references to composed modules. These modules must be previously defined by combining concepts. Two composed modules have been defined to implement the Newton-PILSRK method and their functionality is briefly described below.

Composed Module	Brief functionality description
ComputeR $(s, d, A, .., f, Y, W, X, R)$	$R = h(A \otimes I_d)F_f(Y) + W - X,\ A \in \Re^{s \times s},\ Y, W, X, R \in \Re^{sd}$
UpdateVectors (s, d, A, DX, X, Y)	$[X, Y, DX] \longleftarrow [X + DX, Y + (A \otimes I_d)DX, (A \otimes I_d)DX]$ $A \in \Re^{s \times s},\ X, Y, DX \in \Re^{sd}$

A graphical description of the module specification for the Newton-PILSRK method is presented in figure 4. The arrows denote data dependencies between components and the rectangles represent references to components.

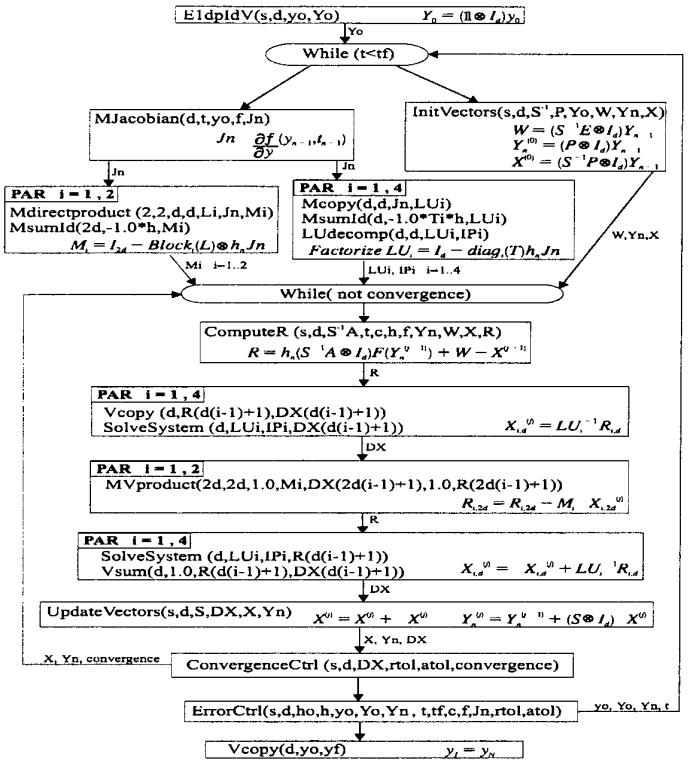


Fig. 4. Graphical representation of the module specification for the method

4.2 Taking Parallel Design Decisions

All the design decisions that affect the performance of the final implementation must be taken at this stage. To carry out this stage, the following information, besides the previously obtained module specification, has to be available:

- The existing realizations for each concept of the module specification.

- The data distribution types for every array argument of the program to be implemented. In our implementation of the Newton-PILSRK method, all the vector arguments of the solver (the initial vector y_0 and the solution vector y_f) are assumed to be replicated among the processors.
- Parameters defining the target machine and the problem size. In our implementation, the target machine is a cluster of 8 PCs based on Pentium II (333 MHz with 128 MB SDRAM) with a switch fast Ethernet interconnection network ($t_s \approx 320\mu s$ and $t_w \approx 0.39\mu s$). We will consider IVPs with dimension lower than 1700.

The decisions to be taken in this stage include, as in TwoL, scheduling and load balancing, but also a task called *instantiation* must be performed. The instantiation involves two main decisions. The first of which concerns the selection of the best realization for each concept that appears in the module specification. In the final structure, only calls to realizations appear. All references to a composed module which appear in the module specification must be recursively replaced with realizations. The second decision implies determining the most suitable data distribution parameters for each realization chosen and the insertion of the required data redistribution routines. The two selection processes are interdependent and must be performed in conjunction with the scheduling and load balancing to obtain a good global runtime solution.

Although, different heuristic techniques are being evaluated to carry out this stage automatically, currently there is no technique available to automate this stage by using the runtime formulas embedded in the realizations. In order to derive an efficient stiff ODE solver, we will follow three guidelines: the even distribution of computational work among the processors, the use of optimal realizations and data distributions for those tasks that impose greatest computational demands and the minimization of redistribution costs. Just by following these guidelines, we have obtained a very satisfactory solution.

Scheduling and Load Balancing We consider a one dimensional 8×1 grid as being the logical topology, and so the SPMD modules run on the groups of processors defined in this structure. These groups are described using a simple notation which must appear in a section of the parallel frame program called GRID.DESCRIPTION (see figure 5(a)).

A graphical description of the parallel frame program can be seen in figure 6. In this figure, the arrows denote the real execution order and each component is labelled with an identifier indicating the processor group where it is executed. As can be seen, the only tasks executed in parallel on disjointed groups are in the concurrent loops which denote most of the task parallelism of the method. These tasks include the solution of the linear systems for each stage as well as every independent computation in which the block-diagonal matrix L takes part. This choice makes it possible to obtain an even distribution of the computational load.

Instantiation Some realizations have been obtained from the libraries PBLAS ($PDCOPY$, $PDAXPY$ and $PDGEMV$) [5] and SCALAPACK ($PDLACPY$, $PDGETRF$ and $PDGETRS$) [3]. These realizations manage arrays according to a parameterized block-cyclic data distribution. The remaining realizations were implemented to perform in parallel several matrix computations which are frequently required when IRK methods are implemented.

The particular data distributions used in the instantiation are described in the DATA.DISTRIBUTION section of the parallel frame program (see figure 5 (b)), where the block-cyclic distribution template is instantiated by specifying a processor group and a block size.

Figure 6 indicates the data distribution which follows each matrix argument in every realization. The same figure has arrows labelled with redistribution operations. Most of redistribution operations have been implemented with the ScaLAPACK redistribution/copy routine PDGEMR2D. However, two user-defined redistribution operations have been used to replicate a block-cyclically distributed matrix ($BCYCLIC(..)\ A \longrightarrow REP\ B$) and to copy a replicated matrix in a block-cyclically distributed matrix ($REP\ A \longrightarrow BCYCLIC(..)\ B$).

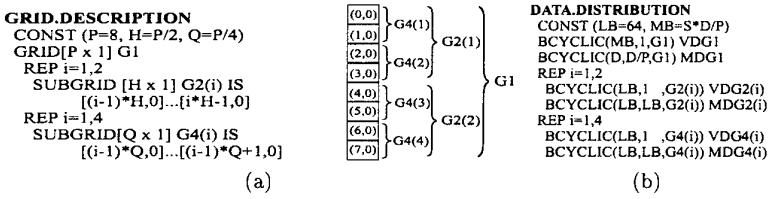


Fig. 5. Description of the processor groups (a) and the data distribution types (b)

We have selected optimal realizations and data distributions for the tasks which make greatest computational demands. These tasks are:

- The dense LU factorizations of the matrices $LU_i = I_d - diag_i(T)h_nJ_n \in \Re^{s\times s}$, $i = 1,\ldots s$, and the solution of the linear systems which use these coefficient matrices. We have used 64×64 blocks in the ScaLAPACK realizations PDGETRF (LU Factorization) and PDGETRS (system solution) which provide a good performance.
- The computation of an approximation to the Jacobian. We have used the $Block\_MJacobian$ realization because this gives the best runtime results.
- The computation of $Block_i(L) \otimes h_nJ_n$, $i = 1,\ldots,2$. To perform this task, we have selected a parallel version of the direct product of matrices in which the input arguments are assumed to be replicated among all the processors of a group and the output matrix is block-cyclically distributed. This realization ($Bcyclic\_Mdirectproduct$) allows us to achieve the best runtime results.

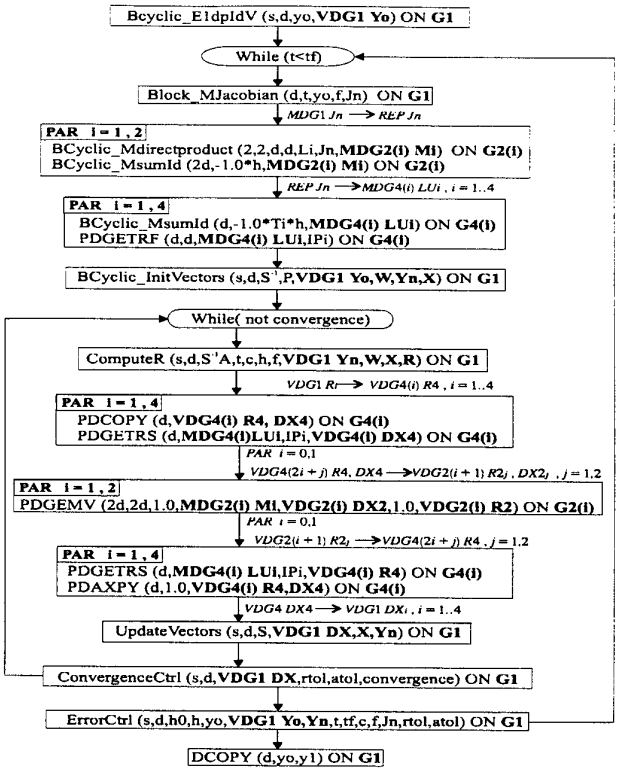


Fig. 6. Graphical description of the parallel frame program

With these decisions, a perfect load balance is achieved because the data for each stage are assigned to a different processor group. On the other hand, redistribution expenses are not high if compared with computation costs.

4.3 The parallel program obtained

The parallel frame program has been translated into a message passing program which is expressed in Fortran augmented with BLAS and BLACS [3]. A version of BLAS [3] optimized for INTEL Pentium II has been used.

5 Numerical Experiments

We compare the runtime performance of our parallel solver with one of the most robust and efficient sequential stiff ODE solvers, the experimental code RADAU5 [9]. We use three test stiff IVPs from [10, 9] which describe several phenomena:

Problem	dimension (d)	Description
Emep	66	The chemistry part of an ozone chemistry model
Cusp	96	A model problem combining three smaller problems
Medical Akzo	400	Injection of a medicine into a tumorous tissue

Since our solver still lacks an error control strategy which would make it possible to select dynamically the most suitable stepsize in every integration step, we perform the tests for one integration step.

It is important to note that in order to get effective performance from a parallel ODE solver, the IVP dimension must be large and the function f should be expensive to evaluate [4]. Otherwise, communication time will dominate computation time. Therefore, to test our solver the IVPs must be scaled. To scale the IVPs, we use a technique described in [1] which basically produces a linear coupling of the original IVP with a function f that is expensive to evaluate.

Some runtime results using different dimensions and the resultant speedup values are presented in figure 7. The results reveal that for systems with more than 1000 equations and a right hand function f relatively expensive to evaluate, a speedup of 4.5 to 5.8 can be achieved on a cluster of 8 PCs.

Approx. *Dim.*	Medakzo		Cusp		Emep	
	T_1 Rad5	T_8	T_1 Rad5	T_8	T_1 Rad5	T_8
100			0.05	0.63	0.15	0.83
200			0.35	0.72	0.52	1.04
300			1.58	1.22	2.64	1.91
400	4.17	1.80	3.68	1.73	4.22	2.18
500			7.3	2.58	9.73	3.53
600			19.6	5.28	18.5	5.47
700			28.0	7.16	25.5	6.87
800	33.2	7.71	41.3	9.67	31.7	8.17
900			56.1	12.4	49.9	11.4
1000			75.5	15.7	73.5	16.1
1100			95.9	19.3	89.9	19.0
1200	115.7	21.6	125	24.7	127	25.4
1300			153	29.3	141	27.5
1400			190	36.1	188	35.6
1500			220	40.3	222	44.8
1600	284	48.8	279	48.4	299	54.1

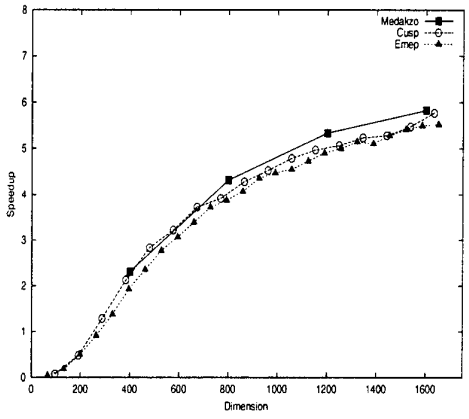


Fig. 7. Some runtime results in seconds and speedup values with the test problems

6 Conclusions

An extension of the TwoL approach which explicitly supports performance polymorphism has been proposed. The proposal allows the user to select the most suitable SPMD implementations for the computation phases of a numerical GSPMD program. This improves flexibility in the performance tuning of the GSPMD software. The proposal has been employed to derive an efficient stiff ODE integrator for a cluster of 8 PCs. The integrator is based on an advanced

numerical method with excellent stability properties and exploits both levels of potential parallelism exhibited by the method and makes effective use of existing parallel linear algebra modules. The implementation achieves a speedup of 4.5 to 5.8 for relatively large dense stiff ODE systems. These results confirm that it is possible to derive efficient parallel implementations of IRK methods on a PC cluster by composing SPMD modules.

Acknowledgements

The experimental results of this work were obtained on a PC cluster supported by the project TIC97-1149 of the CICYT. We would like to thank the contribution of Jose Antonio Carrillo (Granada Univ.) for his assistance in numerical methods.

References

[1] Claus Bendtsen. *Parallel Numerical Algorithms for the Solution of Systems of Ordinary Differential Equations.* PhD thesis, Institute of Mathematical Modelling. Technical University of Denmark, 1996.

[2] Claus Bendtsen. ParSODES - A Parallel Stiff ODE Solver. User's Guide. Technical Report 96-07, UNI-C,DTU, Bldg, 1996.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[4] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations.* Oxford Science Publications, 1995.

[5] J. Choi, J. J. Dongarra, S Ostrouchov, A. Petitet, D. Walker, and R. Clint Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical Report CS-95-292, Computer Science Dept. University of Tennesee, May 1995.

[6] Jacques J. B. de Swart. *Parallel Software for Implicit Differential Equations.* PhD thesis, Amsterdam University, 1997.

[7] J. J. Dongarra and D. W. Walker. Software libraries for linear Algebra Computations on High Performance Computers. *SIAM Review*, 1995.

[8] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard.* Univ. of Tennessee, Knoxville, Tennessee, 1995.

[9] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential Algebraic Problems.* Springer-Verlag, 1996.

[10] W. M. Lioen, J. J. B. De Swart, and W.A. Van Der Veen. Test set for IVP solvers. Technical Report NM-R9615, CWI, 1996.

[11] T. Rauber and G. Rünger. Deriving Structured Parallel Implementations for Numerical Methods. *The Euromicro Journal*, 41:589–608, 1996.

[12] M. Sitaraman and B. Weide. Special Feature: Component-Based Software Using RESOLVE. *ACM SIGSOFT, Software Engineering Notes*, 19, 1994.

[13] A. Skjellum, Alvin P. Leung, S. G. Smith, R.D. Falgout, C.H. Still, and C. H. Baldwin. The Multicomputer Toolbox - First-Generation Scalable Libraries. *HICSS-27*, pages 644–654, 1994.

[14] P. J. Van der Houwen and J. J. B. de Swart. Parallel linear system solvers for Runge-Kutta methods. *Advances in Computational Mathematics*, 7:157–181, March 1997.

Efficient Pipelining of Level 3 BLAS Routines

Frédéric Desprez and Stéphane Domas

LIP, ENS Lyon
46 Allée d'Italie
F-69364 Lyon cedex 07
(desprez,sdomas)@ens-lyon.fr

Abstract. This paper presents a method that handles automatically a pipeline of level 3 BLAS routines, executed on different processors. It is based on linear graph of tasks (each task is a single BLAS 3 call), which describes the dependences between the matrices used for computations. From such a graph and from a theoretical model of level 3 BLAS execution times, we determine the best blocking for each task and the optimal blocking strategy size for the pipeline.

1 Introduction

Optimizing parallel algorithms is a difficult task, especially if we keep in mind the portability issues. However, several techniques exists to speed up parallel applications on distributed memory machines. Asynchronous communications are a way to improve the performances and the scalability of parallel routines. Using such communications, a processor can compute while communicating, reducing the total overhead of communications. Pipelining is an optimization technique that can be used when data dependences produce too much idle time. It consists in splitting a task in a certain number of sub-tasks and to communicate the result of one sub-task as soon as it has been computed, allowing the next processor to start sooner. The way tasks are split is of course dependent of many parameters (algorithm, network bandwidth and latency, computation speed, ...). It can often be computed or approximated to obtain the best performances.

Since many numerical applications are now using the standardized level 3 BLAS calls, our aim is to provide a general mechanism for the use of pipelined overlap when the result of a level 3 BLAS task has to be communicated. Because level 3 BLAS routines are already optimized for a broad range of processors, possible optimizations can be made in a succession of BLAS calls, executed on different processors. At the moment, we target linear tasks graphs where each task is executed on a different processor and where tasks can be split. We show how tasks have to be split and when.

Many numerical applications can benefit of such optimization. In [3], the authors describe mechanisms to parallelize applications with two levels of parallelism; coarse-grain task parallelism at the outer level and data-parallelism at the inner level. Their motivating example is an algorithm for the computation of eigenvalues of a dense non-symmetric matrix [1]. Sparse matrix factorization like

the one described in [5] or [7] can also make use of these mechanisms. Those algorithms compute dense BLAS operations on different blocks with dependencies between the blocks (given by the sparse structure of the matrix). The problem in this last example is that dependencies are only known at run-time. However, the optimization we describe can be used inside the run-time system or after the symbolic factorization.

One important part of the optimization process is the blocking of the computation routine. The goal is to keep the computation speed high and to reduce its grain. Several papers discuss the optimization of level 3 BLAS codes using loop partitioning techniques and efficient use of the memory hierarchy [4, 8]. The most recent works are inside the PHiPAC [2] and ATLAS projects [9].

In this paper, we present an overview of problems and solutions for a motivating example. Then, we describe in detail the two main steps of our methodology. We finish by experimental results and an example of combination of our approach with data-parallelism.

2 Motivating Example

Our motivating example is a linear task graph made of five matrix product tasks (level 3 BLAS `DGEMM` routine) executed on five different processors. Dependencies between the tasks are given in Figure 1. The first arrow means that after its computation, matrix C is sent to the next processor as an entry of the next matrix product. Its a flow dependence (read after write).

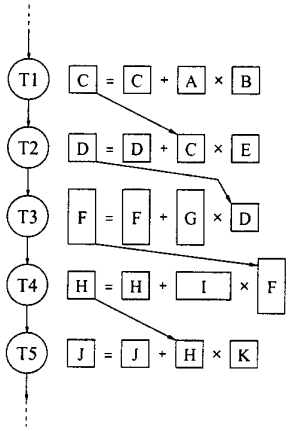


Fig. 1. Motivating Example: pipeline of 5 matrix product tasks.

This example is quite simple but it shows the main issues in the optimization of such a pipeline of level 3 BLAS routines. We have chosen the matrix product because it is used as a kernel operation of many linear algebra algorithms and

there are many simple ways of blocking its loops. Moreover, other level 3 BLAS routines can be efficiently rewritten with DGEMM calls [8].

Because of the dependences between the tasks, the execution of such task graph can not be optimized without a split of the tasks. If different processors are chosen for the execution of each task, the overall parallel execution time is even worse than the sequential one because we add the cost of the communications between different processors.

3 Pipelining the motivating example

There are several ways to block the computations to pipeline the DGEMM tasks. It is equivalent to split the matrices. In Figure 2, we give three different ways to block the matrices used in a multiplication. Every loop can be blocked but it may not lead to an efficient communication pipeline.

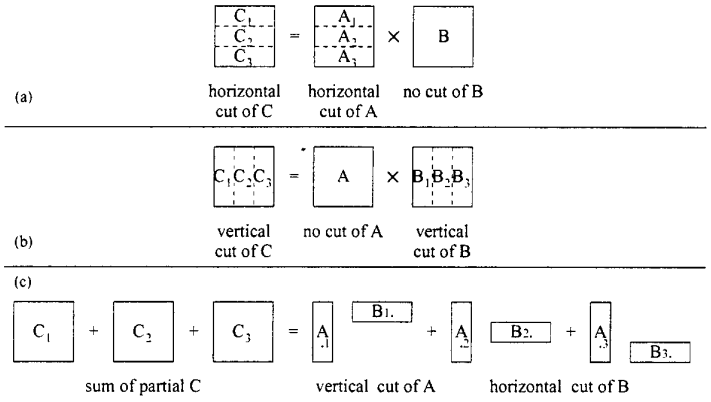


Fig. 2. Motivating Example: three possible blockings for the matrix product algorithm.

In the first two cases, we only block one matrix and compute vertical or horizontal blocks of C. The resulting algorithm is perfect for a pipeline since a task receives, computes and sends rectangular blocks. The third case is problematic since we compute square blocks which are intermediate results. This situation is best avoided but in our example, the dependencies between matrices have been chosen to bring such a case. We will show later how to solve this problem.

3.1 Choosing a Blocking for the Computations

Taking into account the dependencies, we must find an efficient way to block the computations for each task in the graph. Instead of vertical or horizontal blocks, we will use more obvious notations. According to the BLAS reference guide [6], a DGEMM uses three matrices: $A_{(M\times K)}$, $B_{(K\times N)}$ and $C_{(M\times N)}$. In the

the first blocking in Figure 2, A is blocked along its M dimension. In the second solution, B is blocked along its N dimension. In the last solution, A and B are blocked along their K dimension. Thus, we will use the notations M-block, N-block and K-block for the three possibilities to block a `DGEMM` and more generally, each BLAS 3.

We will also use the notation S-block when the matrices are the sum of partial results like C in the third case. Meanwhile, an S-block `DGEMM` requires more than $O(MNK)$ floating operations to compute and thus, must be avoided.

3.2 Choosing a Blocking for the Communications

In our example, we always send a result matrix. Thus, the "orientation" (vertical or horizontal) of the blocks that are sent is determined by the blocking used for the computation. For example, if the first task is M-block, the result matrix C' is blocked horizontally and horizontal blocks must be sent. If it is not a result matrix that is sent, we have the choice for the orientation.

Since there is no preferred dimension for the communications, we use the notation H-block or V-block depending on the "orientation" of the blocks that are sent. We also use the notation S-block when a matrix is sent in partial results and $\oslash$-block when the matrix is sent in a whole.

3.3 Some Solutions to Block the Matrices

In Table 1, we present four ways of blocking computations and communications for the motivating example. For each solution, the left column gives the blocking used for the computations and the right column, the blocking for the communications.

	Task 1	**Task 2**	**Task 3**	**Task 4**	**Task 5**
Solution 1	O/O	O/O	O/O	O/O	O/-
Solution 2	M/H	M/H	K/S	S/S	S/-
Solution 3	M/H	M/H	K/O	M/H	M/-
Solution 4	M/H	M/O	N/V	N/V	K/-

Table 1. Four different blocking for our motivating example.

- **First solution:** no pipeline. Sequential execution.
- **Second solution:** Task 1 and 2 use M-block for the computations. Task 3 receives horizontal blocks in D. It implies that Task 3 is K-block and the result matrix F is S-block. Further tasks in the pipeline are also S-block.
- **Third solution:** it begins like the second solution but Task 3 sends the whole F matrix ($\oslash$-block communication) instead of an S-block communication. Then, Task 4 begins a "new" pipeline with M-block computations.
- **Fourth solution:** Task 1 and 2 are M-block for computations but Task 2 sends the whole D matrix. Then, Task 3 begins a "new" pipeline with N-block computations.

Assuming that each task needs 3 seconds to complete and we divide them in 3 (i.e. blocking size = 3). We also assume that each sub-task takes 1 second to complete (which is not experimentally true). The Figure 3 shows a Gantt-like diagram for each solution. Rectangles are the tasks or the sub-tasks and the lines, the communications.

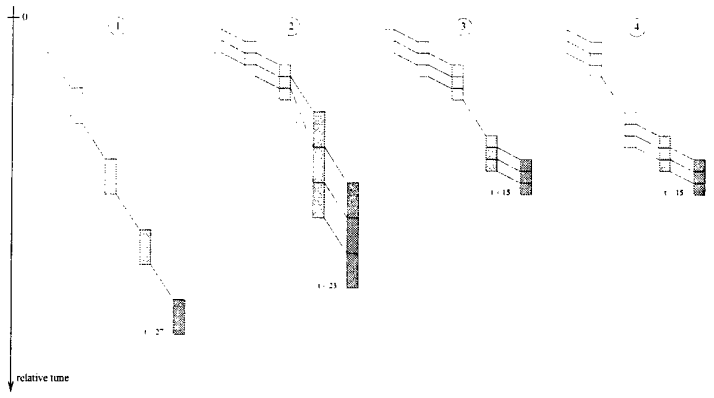


Fig. 3. Motivating example: Gantt chart for the four solutions.

- In the first column, all tasks are completed before sending the result to the next task in the pipeline. It is the no-pipeline, no-overlap solution and it takes 27 seconds to complete.
- The second solution takes a little less time than the no-pipeline solution since there are some S-block computations. Task 4 computes $F.I + H \rightarrow H$. Blocking F in 3, Task 4 mathematically computes $(F_1 + F_2 + F_3).I + H \rightarrow H$ which requires $3n^2 + n^3$ operations. But the F_i matrices are received sequentially. Thus, Task 4 computes $F_1.I + F_2.I + F_3.I + H \rightarrow H$ which requires $3n^2 + 3n^3$ operations. The same occurs for the Task 5 and it leads to an execution time in 23 seconds. It is better than the first solution but largely more than a sequential execution on a single processor (15 seconds).
- For the third and fourth solutions, a gain close to 2 can be reached despite of the chain of BLAS is divided in two pipelines.

4 Computing the optimal parameters for the pipeline

The motivating example points out that the pipeline efficiency greatly depends on the blocking for each matrix. Solutions 3 and 4 prove that there is a set of best blockings. Another parameter is the blocking size. A blocking size of 3 is surely not the optimal value for experimental tests, especially for solutions 3 and 4 where the two pipelines may have different blocking sizes. Choosing a blocking

size which is too small leads to an inefficient pipeline. Choosing one size too big multiplies the communication latencies and reduces the BLAS 3 efficiency since the blocks used in computations are smaller. In order to fix these two parameters, our method consists in two steps: finding the set of best blocking solutions and then finding the optimal blocking size(s) for each solution.

With the optimal blocking size(s), we can compute the total execution time of each pipeline and determine the best solution.

4.1 Finding the Set of Best Blocking Solutions

The blocking of each task is very often determined by a previous one in the pipeline but sometimes, it can be chosen. In this case, we obtain several possible chains of blocking. A tree with all possibilities can be constructed but with an assumption and a condition, only the "best" solutions in the tree can be picked. Best means that these solutions lead to an efficient pipeline.

In the second solution of the Figure 3, we see that S-block computations are propagated and reduce the efficiency of the pipeline. In the solution 3, we exchange the first S-block communication by an $\oslash$-block communication. Then, the pipeline is more efficient.

In the fourth solution, there are no K-block computations (except the last but it has no influence). Mathematically, a K-block computation always implies a S-block communication, which must be avoided. In this solution, each task sends a block that does not produce a K-block computation in the next task. If it is not possible, the task sends a $\oslash$-block matrix. To summarize, we have:

- **NSP-assumption** ("No S-block Please !"): each time an S-block communication occurs, replace it by the communication of the whole matrix, i.e. delay the communication outside the loops.
- **NCB-condition** ("Next Computation Blocking"): sending horizontal or vertical blocks produces a K-block computation in the next task.

According to Table 1 and Figure 3, when the NCB-condition occurs, we have the choice to send horizontal or vertical blocks as in solution 3, or to send the whole matrix as in solution 4. Indeed, the third and the fourth solution are equally efficient thus we cannot decide when it is interesting to send the whole matrix and when it is not. Consequently, two solutions must be generated.

The algorithm that constructs the set is given in Figure 4. The computation and communication blocking of the i^{th} task are noted $BL_i = [comp_i; comm_i]$ (for example, $[M; H]$). Each solution is a set of BL_i, constructed by iterating on the tasks, taking account of the dependencies, the type of BLAS 3 and testing if NSP-assumption or NCB-condition occurs.

4.2 Computation of the Optimal Blocking Size

We have seen in the motivating example that it is mandatory to know the execution time of each task to determine the best blocking solution. We have also supposed that the tasks were split in 3 sub-tasks. In fact, the blocking size very

```
nb_sol = 1    /* number of solutions */
Sol = ∅    /* set of solution is empty */
For i = 1 to nb_task − 1 do
  For j = 1 to nb_sol do
    Find BL_i from description of task_i and BL_{i−1}.
    If (comm'_i = S) || (comm'_i = ⊘) then
      Use NSP-assumption→ comm'_i = ⊘.
      Concat BL_i with Sol_j.
    Else
      Concat BL_i with Sol_j.
      If (NCB-condition applies) then
        /* create a new solution */
        cut'_i = BL_i
        comm'_i = ⊘
        Sol_{nb_sol+1} = Sol_j.
        Concat cut'_i with Sol_{nb_sol+1}.
        nb_sol = nb_sol + 1.
      Endif
    Endif
  Endfor
Endfor
```

Fig. 4. Algorithm to construct the set of the best solutions.

influences the efficiency of the pipeline. A matrix multiplication split in L partial computations takes more time than the the computation performed in one time. But, the execution time of the pipeline can only be determined if the blocking size is fixed. Our goal is to compute automatically the optimal blocking size L for each solution.

Theoretical Level 3 BLAS Execution Time All level 3 BLAS subroutines are matrix-matrix operations and thus compute $O(n^3)$ floating point operations for $O(n^2)$ memory accesses. More generally, the execution time of a level 3 BLAS depends on the size of the matrices which are involved in the computation. Let $A_{(M\times K)}$, $B_{(K\times N)}$, and $C_{(M\times N)}$ be the three matrices used by a level 3 BLAS. We assume that the execution time of a level 3 BLAS is given by the expression:

$$T_{BLAS3}(M,N,K) = aMNK + bMN + cMK + dNK + eM + fN + gK + h \quad (1)$$

where $a \ldots h$, parameters that depend on the algorithm used in the routine and on the target machine used. These parameters can be found by an interpolation on experimental tests. Unfortunately, we have noticed that they vary as the protocol of the tests vary (increasing step for the matrix size, which dimension increases ...). In order to have consistent parameters, it is necessary to have specific expressions of level 3 BLAS execution time for our problem.

According to the BLAS syntax, a DGEMM ($\alpha A.B+\beta C \rightarrow C$) can be M-block, N-block or K-block. The expression 1 does not take care of the blocking dimension. Consequently, we take a different expression for each case. For example, the expression for an M-block computation of DGEMM is:

$$T^M_{GEMM}(M,N,K) = (a_M M + b_M)NK + (c_M M + d_M)\sqrt{NK} + (e_M M + f_M)$$

Furthermore, the matrices can be transposed. It implies that the parameters $a_M, b_M, \ldots$ are also different for each possibility of transposition. For DGEMM, there are four possibilities thus, a total of 12 set of parameters.

Execution Time of a Level 3 BLAS Cut in L Blocks In the pipeline, a BLAS 3 is computed in several times. What is its execution time when it is cut in L sub-computations along the D dimension ? We assume that the blocks must almost have the same width in order to have a regular pipeline. Choosing a single size may penalize the pipeline efficiency because the last block may be to large or to small. We prefer to choose two sizes that differ only by one.

Let $r = D \bmod L$. There will be $L - r$ blocks of width $\lfloor \frac{D}{L} \rfloor$ and r blocks of width $\lceil \frac{D}{L} \rceil$. Assuming that $D = M$, the execution time of a D-block level 3 BLAS cut in L blocks becomes:

$$\begin{aligned} T^M_{BLAS3}(M,N,K,L) = \\ (L-r)\left[(a_M \lfloor \tfrac{M}{L} \rfloor + b_M)NK + (c_M \lfloor \tfrac{M}{L} \rfloor + d_M)\sqrt{NK} + (e_M \lfloor \tfrac{M}{L} \rfloor + f_M)\right] + \\ r\left[(a_M \lceil \tfrac{M}{L} \rceil + b_M)NK + (c_M \lceil \tfrac{M}{L} \rceil + d_M)\sqrt{NK} + (e_M \lceil \tfrac{M}{L} \rceil + f_M)\right] \end{aligned}$$

It gives after simplification:

$$T^M_{BLAS3}(M,N,K,L) = (a_M M + b_M L)NK + (c_M M + d_M L)\sqrt{NK} + (e_M M + f_M L)$$

$$T^M_{BLAS3}(M,N,K,L) = L \times T^M_{BLAS3}(\tfrac{M}{L}, N, K)$$

Thus, the theoretical execution time of a level 3 BLAS split in L blocks is a linear function of L. Meanwhile, experimental results show that it is partially true. In fact, the $a_{M_{nn}}$, $b_{M_{nn}}$,... parameters are defined by intervals. Figure 5 gives the experimental execution time of a M_{nn} DGEMM, as a function of M. We can see that there are at least two intervals ($[2,11],[12,128]$) in which the parameters are different. For an accurate simulation, we define a lot of intervals for small matrix sizes because a small number of floating operations does not allow a good utilization of the arithmetic unit pipeline. But it very depends on how the level 3 BLAS is implemented and which processor is used.

In order to compute parameters on a large number of intervals, we have implemented a routine that produces sets of experimental curves for each BLAS 3, computes automatically the intervals and the parameters defined on, and finally, outputs a C code containing array initializations.

Execution Time of a Pipeline The next step is to fix the optimal blocking size L to determine the total execution time of each solution. In fact, there may

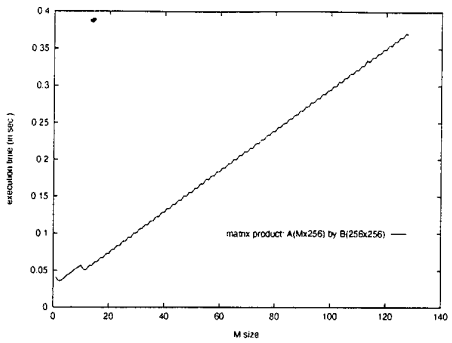


Fig. 5. Execution time of a `DGEMM` as a function of dimension M.

be several pipelines for a single graph, like solutions 3 and 4 of the motivating example. Thus, several optimal blocking sizes must be computed to obtain the total execution time of the graph.

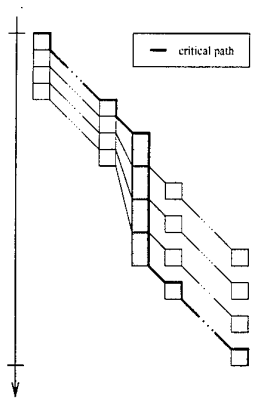


Fig. 6. Critical path.

For each pipeline, we define a critical task and a critical path. If there are N tasks in the graph, the critical path is an execution path in the Gantt diagram which crosses $N-1$ sub-tasks, the critical task and $N-1$ communications. A general example is given in Figure 6. The bold line is the critical path. All the dark gray blocks form the critical task which is the longest task in the pipeline and is totally crossed by the critical path. We also assume that this task has enough space for communication buffers, to avoid waiting to send a message between its sub-task. For other tasks, only one of their sub-tasks is crossed by the critical path. The length of the critical path gives the total execution time

of the pipeline.

From equations given in Section 4.2, we obtain the execution time of the critical task and each sub-task crossed by the critical path. The sum of all gives the execution time of the pipeline as a polynomial function of the blocking size L. With a derivative, we find the optimal blocking size. Another way is to make an exhaustive research of the optimal blocking size, testing all possible values of L. The second method is longer but leads to better results because all parameters are taken at the execution time. With a derivative, some parameters are removed.

5 Library Interface

With the optimal blocking size of each pipeline of each solution, we determine the best solution and then, have all keys to implement an efficient pipeline. Meanwhile, this implementation may be hard for someone who is not familiar with the BLAS 3 syntax and communications libraries such as PVM, MPI or BLACS. For a convenient use of our method, we have provided an interface which handles at runtime and transparently the pipeline(s) of any linear task graph of level 3 BLAS. It consists in 5 main routines that a user has to call. The Table 2 gives the C code that implements a pipeline for motivating example.

```
NewPipe(5);
InitTask(0,DGEMM,P_nn,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,NO,MCC,NO,NO,0,1);
InitTask(1,DGEMM,P_nn,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,MA,MCC,0,0,0,2);
InitTask(2,DGEMM,P_nn,2*i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,MB,MCC,0,1,0,3);
InitTask(3,DGEMM,P_nn,i,i,2*i,1.2,A,LDA,B,LDB,1.3,C,LDC,MB,MCC,0,2,0,4);
InitTask(4,DGEMM,P_nn,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,MA,NO,0,3,NO,NO);
InitPipe();
RunPipe();
FreePipe();
```

Table 2. Motivating example: C code to handle the pipeline at run-time.

The description of each task is done by `InitTask(...)` calls. Parameters are the name of the BLAS 3, the matrices and their sizes, which matrices are sent, ... `InitPipe()` computes the best solution and the optimal blocking size(s). `RunPipe()` execute the pipeline.

This interface has three advantages. First, it is very simple. The user just have to describe the task graph. He does not need special knowledge on what is a pipeline and how to implement it. The execution is also transparent and no calls to communication routines is necessary. Secondly, it is portable. The communications are based on the BLACS and the computations on the BLAS. Finally, everything is handled at runtime. Thus, the initialization of the task graph can dynamically parameterized.

6 Experimental results

Our library gives solution 4 as the best solution for the motivating example. This result is logical since tasks 3 and 4 work on larger matrices. It is obvious that pipelining the biggest tasks together (solution 4) is more accurate than pipelining them separately (solution 3).

In order to validate our method, we have also tested solution 3. Figure 7 presents the execution time of solutions 1 (no pipeline), 3 and 4, on our cluster of Pentium, using the BLACS over MPI-LAM for communications. The time is given for the optimal blocking size. Figure 8 shows the gain for solutions 3 and 4. The gain is the execution time of these solutions divided by the execution time of solution 1. It clearly shows that solution 4 is better than 3. Furthermore, a gain greater than 2 can be reached despite the graph is broken into two pipelines.

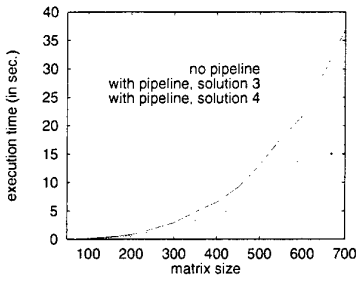


Fig. 7. Motivating example: execution time on a cluster of Pentium.

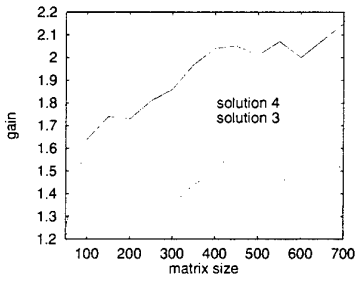


Fig. 8. Motivating example: gain on a cluster of Pentium.

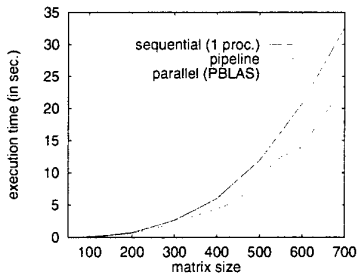


Fig. 9. Motivating example: comparison between sequential, parallel and pipeline execution time on a cluster of Pentium (POPC).

In Figure 9, we give a comparison between three different execution times on the cluster of Pentium. The first one is the purely sequential time: only one processor computes the five `DGEMM`. The second one is the pipelined version time. The last one is the purely parallel time: no pipeline but all processors computes each BLAS 3 with a PBLAS routines.

We notice that the pipelined version is less efficient than the PBLAS version. It clearly shows that breaking the task graph into several pipelines decreases performances. Meanwhile, the PBLAS approach has two drawbacks. First, it is not so easy to use and requires some advanced knowledge in message passing implementation style. Secondly, it is more complicated to distribute the matrices. Also, our method is a good compromise between efficiency and ease of use. Moreover, our pipeline strategy is more efficient when the chain of task is already mapped on the processors, due to previous computations. This is the case for example when pipelining the kernel of a sparse matrix factorisation [5, 7]. Dense blocks are already mapped on the processors and we would like to pipeline the updates. Our method can easily be used in this case.

We have also tested our method on other examples and machines (Paragon, cluster of PowerPC,...). For example, with 5 identical `DGEMM`, we reach a gain of 3.85, on the cluster of Pentium. More generally, we have noticed that such a graph (identical BLAS with a single pipeline) with n tasks always leads to a gain near form $n-1$ (which is indeed optimal).

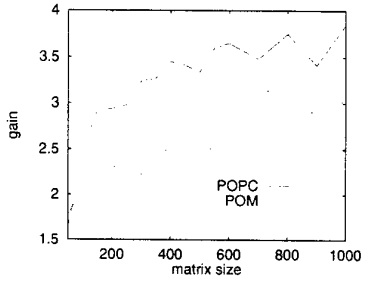


Fig. 10. 5 identical `DGEMM`: gain on clusters of Pentium (POPC) and PowerPC (POM).

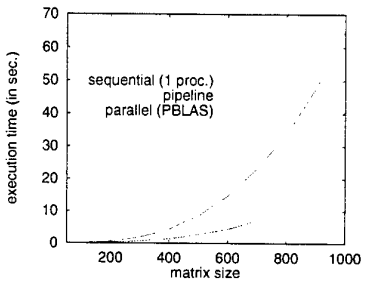


Fig. 11. 5 identical `DGEMM`: comparison between sequential, parallel and pipeline execution time on a cluster of Pentium (POPC).

In Figure 11, we give a comparison between sequential, pipeline and parallel execution times (as in Figure 9). This time, the pipeline version is as efficient as the parallel version and much more than a sequential execution. This result is identical for other tests with a single pipeline even if there are different BLAS 3 in the graph. It proves that our method is as efficient as PBLAS/ScaLAPACK in such cases.

7 Using our Approach in a Data-Parallel Programs

We have seen in Figures 9 and 11 that pipelining is always better than doing computations sequentially on a single processor. Meanwhile, the parallelism (implicitly, the number of processors) strictly depends on the number of tasks in the graph. It seems to be a limitation if we consider that the problem size may increase beyond the resource of the processors. This is not a problem for a PBLAS

computation since we can use a variable number of processors. In fact, the dynamical behavior of our method allows to mix it with data-parallelism. Especially in block algorithms, there are dependencies between sequential computations on different processors. In some case, these dependencies can be expressed as a set of linear graph of task and each graph uses a separate set of processor. For example, we can construct a block algorithm with such dependencies for the multiplication of two matrices, distributed by block (cyclic or not) on a grid of processor. In all these cases, our method can be used whatever the number of processors is since our interface allows to define the graph dynamically.

We have implemented the matrix multiplication algorithm and compared it to that proposed in the PBLAS: `PDGEMM`. It uses a simple full-block distribution vs. block scattered for `PDGEMM`). The algorithm is nearly similar to the `PDGEMM` routine except that we replace the broadcast at the beginning of each step by a pipeline for each processor on the same row. Thus, all processors are working on the same amount of data and it leads to a perfect load balance. If the blocking size is well chosen, the idle time due to contentions is minimal.

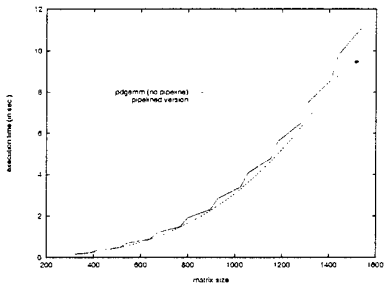


Fig. 12. Execution time of `PDGEMM` and the pipelined matrix multiplication on 16 processors of the Paragon.

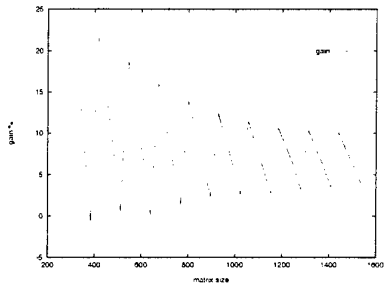


Fig. 13. Gain of the pipelined matrix multiplication over `PDGEMM`.

First, we notice in Figure 12 that the execution time of `PDGEMM` is not perfectly cubic. This comes from a slight load imbalance for some matrix sizes. Our version does not have this problem and thus, gives largely better execution time in these cases.

Secondly, our routine has always better results for large matrices. For such matrices, the average gain of our routine, given in Figure 13, is 7%. In fact, there are a lot of broadcast and global sum communications in `PDGEMM` and the processors are often idle. In our routine, there are only four steps of computations (4×4 processors), thus only four global sum and three pipeline phases. During these phases, the processors are communicating while they are computing: there are less idle than in `PDGEMM`.

8 Conclusion

In all cases, the experimental results have confirmed the usefulness and efficiency of the pipeline in tasks graph over a sequential execution. With asynchronous communications, the total execution time of the graph can be reduced by two or even more, depending on the size of the graph and the dependencies between the tasks. Moreover, experimental results have validated our methods to compute the set of best blockings and the optimal blocking size. When the dependencies leads to a single pipeline, our method gives a code as efficient as a message passing approach but much easier to implement. Ten lines of code are sufficient, against hundred for a PBLAS/ScaLAPACK code.

Finally, some real applications can be optimized by our method. We have already give some of them in the introduction but it also works each time the dependencies between sequential computations produce a set of linear graph of tasks, with each graph working on different processors.

Our future work is the validation of our method in a sparse matrix factorisation by pipelining the block updates using level 3 BLAS routines and to extend the algorithm to non-linear task graphs.

References

1. Z. Bai and J. Demmel. Design of a Parallel Nonsymmetric Eigenroutine Toolbox - Part I. In SIAM, editor, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 1993.
2. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC.
3. S. Chakrabarti, J. Demmel, and K. Yelick. Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. *Journal of Parallel and Distributed Computing*, 47:168–184, 1997.
4. M.J. Dayde, I.S. Duff, and A. Petitet. A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors. Technical Report RT/APO/93/1, ENSEEIHT - Département Informatique N7 -I.R.I.T. Gpe Algorithmes Parallèles, 1992.
5. J.W. Demmel, J.R. Gilbert, and X.S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination, February 1997. [Available via ftp in `ftp://parcftp.xerox.com/pub/gilbert/parlu.ps.Z`].
6. J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. on Mathematical Soft.*, 16(1):1–17, 1990.
7. A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. Design and Implementation of a Scalable Parallel Direct Solver for Sparse Symmetric Positive Definite Systems. In *Proc. of the 8th SIAM Conf. on Parallel Processing*, March 1997.
8. B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High Performance Model Implementations and Performance Evaluation Benchmark. Technical Report UMINF-95.18, Umeå University, October 1995.
9. R. Whaley and J. Dongarra. LAPACK Working Note 131: Automatically Tuned Linear Algebra Software. Technical Report CS-97-366, The Universtity of Tennessee - Knoxville, 1997. `http://www.netlib.org/atlas/`.

A Parallel Algorithm for Solving the Toeplitz Least Squares Problem*

Pedro Alonso[1], José M. Badía[2], and Antonio M. Vidal[1]

[1] Dpto. de Sistemas Informáticos y Computación, Univ. Politécnica de Valencia, Cno. Vera s/n, 46022 Valencia, Spain
{palonso, avidal}@dsic.upv.es
[2] Dpto. de Informática, Univ. Jaume I, Castellón, Spain
badia@inf.uji.es

Abstract. In this paper we present a parallel algorithm that solves the Toeplitz Least Squares Problem. We exploit the displacement structure of Toeplitz matrices and parallelize the Generalized Schur method. The stability problems of the method are solved by using a correction process based on the Corrected Semi-Normal Equations [8]. Other problems arising from parallelizing the method such as the data dependencies and its high communication cost have been solved by using an optimized distribution of the data, rearranging the computations and designing new basic parallel subroutines. We have used standard tools like the ScaLAPACK library based on the MPI environment. Experimental results have been obtained in a cluster of personal computers with a high perfomance interconnexion network.

1 Introduction

Our goal is to solve the Least Squares (LS) problem

$$\min_x \|Tx - b\|_2 \,, \tag{1}$$

where the matrix $T \in \mathrm{R}^{m \times n}$ is Toeplitz, $T_{ij} = t_{i-j} \in \mathrm{R}$ for $i = 0, \ldots, m-1$ and $j = 0, \ldots, n-1$, and for an arbitrary vector $b \in \mathrm{R}^m$, in a parallel computer.

This problem arises in many applications such as time series analysis, image processing, control theory, statistics, in some cases with real-time constraints (e.g. in radar and sonar applications).

It is well known that the LS Problem can be computed in $O(mn^2)$ flops in general, using e.g. Householder transformations [10]. But, for a Toeplitz matrix T, several fast algorithms of $O(mn)$ flops exist [13, 6]. All these fast algorithms are generalizations of a classical algorithm by Schur [12]. However, the stability properties of the fast algorithms are inferior to those of the standard LAPACK [3] algorithm based on Householder transformations. A good overview of the stability and accuracy of fast algorithms for structured matrices can be found in [4].

* Partially funded by the Spanish Government through the project CICYT TIC96-1062-C03.

In this paper, we will use several enhancements that make the fast algorithm based on the Generalized Schur Algorithm more accurate and reliable. The parallel algorithm that we have implemented is based on a sequential one proposed by H. Park and L. Eldén [8]. In that paper the accuracy of the R factor computed by the Generalized Schur Algorithm is improved by post-processing the R factor using Corrected Semi-Normal Equations (CSNE). Our parallel algorithm inherits the accuracy properties of that method.

Assume that the matrix $R_0 \in \mathrm{R}^{(n+1)\times(n+1)}$ is the upper triangular submatrix in the R-factor of the QR decomposition for the matrix $[b\ T]$,

$$R_0 = \mathrm{qr}\left([b\ T]\right) = \begin{pmatrix} \kappa & \omega^T \\ 0 & G \end{pmatrix}, \quad \kappa \in \mathrm{R}, \quad \omega \in \mathrm{R}^{n\times 1}, G \in \mathrm{R}^{n\times n}, \tag{2}$$

where G is upper triangular and with the operator qr we denote the upper square submatrix of the R factor of the QR decomposition of a given matrix. Then, the LS problem (1) can be solved via a product of Givens rotations J that reduces a Hessenberg matrix to de upper triangular form,

$$J\begin{pmatrix} \omega^T & \kappa \\ G & 0 \end{pmatrix} = \begin{pmatrix} R & r_1 \\ 0 & r_{nn} \end{pmatrix}, \tag{3}$$

where $R \in \mathrm{R}^{n\times n}$ is the upper triangular factor of the QR decomposition of T. The vector solution x (1) is obtained by solving the triangular linear system $Rx = r_1$ [10].

Solving the LS problem (1) involves four main steps that we summarize in Algorithm 3 in section 5. The first one is to form a *generator pair* (which we will explain further on). Starting from the *generator pair*, we obtain the triangular factor G which appears in (2) by means of the Generalized Schur Algorithm. The third step consists of *refining* that factor G in order to acquire a more accurate factor G, and the last step is an ordinary solution of a triangular system that solves the LS problem (1) as made in the standard method for solving a general LS problem via a QR decomposition of the augmented matrix $[T\ b]$.

We have parallelized the four steps. The second step is described in section 2 while the third one is described in sections 3 and 4. With the parallel version of the Generalized Schur Algorithm proposed here, we can reduce the time needed to obtain the triangular factor G and we obtain a parallel kernel available to other problems based on this method (e.g. linear structured systems see [1, 2]). With the parallel version of the third step we can aliviate the overcost introduced by the correction step needed to obtain a more accurate solution in the case of non strongly regular matrices.

Several problems arise in the development of the parallel version of the method. First, the difficulty in obtaining high parallel performance from a low cost algorithm. Second, the sequential algorithm has a lot of data dependencies as in many other fast algorithms that work on structured matrices. This fact reduces drastically the granularity of the basic computational steps, so increases the effect of the communications. However, we have reduced the effect of these problems by using an appropriate data distribution, rearranging the operations

carried out in order to reduce the number of messages, and implementing new efficient basic parallel subroutines adequate to the chosen data distribution.

We have implemented all algorithms in FORTRAN, using BLAS and LAPACK libraries for the sequential subroutines, the PBLAS and ScaLAPACK [9] libraries for solving basic linear algebra problems in parallel and for data distribution. The utilization of standard libraries assures portability and produces a parallel program based on well known and efficient tested public code. We have used subroutines of the BLACS package over MPI [7] to perform the communications. As it is shown bellow, the best topology to run the parallel algorithm is a logical grid of $p \times 1$ processors.

2 Exploiting the Displacement Structure

The displacement of the matrix $[b\ T]^T[b\ T]$ with respect to the shift matrix $Z = [z]_{i,j=0}^{n}$, where $z = 1$ if $i = j + 1$ and $z = 0$ otherwise, is denoted by ∇_Z and defined as

$$\nabla_Z = [b\ T]^T[b\ T] - Z[b\ T]^T[b\ T]Z^T = \mathcal{G}\mathcal{J}\mathcal{G}^T\ . \tag{4}$$

The matrix $[b\ T]^T[b\ T]$ has low displacement with respect to Z if the rank of ∇_Z is considerably lower than n [11]. The factor $\mathcal{G}$ is an $n \times 6$ matrix called *generator*, and $\mathcal{J}$ is the *signature* matrix $(I_3 \oplus -I_3)$. The pair $(\mathcal{G}, \mathcal{J})$ is called a *generator pair*. Given a general Toeplitz matrix T and an arbitrary independent vector b, the generator pair for equation (4) is known [8], and is computed at the first step of the parallel Algorithm 3 to solve the least squares problem (1).

The Generalized Schur Algorithm computes the factor R_0 (2) of the following Cholesky decomposition,

$$[b\ T]^T[b\ T] = R_0^T R_0\ ,$$

in $O(mn)$ flops instead of the $O(mn^2)$ flops required by the standard LAPACK algorithm. The Generalized Schur Algorithm is a recursive process of n steps. In step i $(i = 1, \ldots, n)$, a $\mathcal{J}$-unitary transformation Θ_i $(\Theta_i \mathcal{J} \Theta_i^T = \mathcal{J})$ is computed in order to transform the first nonzero row of $\mathcal{G}$ to a vector of the form $\begin{pmatrix} x & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$. The first column of generator $\mathcal{G}$ is the i row of the Cholesky triangular factor of the matrix $[b\ T]^T[b\ T]$. Each $\mathcal{J}$-unitary transformation is performed by a composition of two Householder transformations and a hyperbolic rotation [5].

In the parallel version of the Generalized Schur Algorithm that we present, the generator $\mathcal{G}$ is divided in blocks of v rows and cyclically distributed over a $p \times 1$ processors grid as it is shown in Fig. 1. The processor having the i row of $\mathcal{G}$ (first nonzero row of the generator denoted by the x entries) computes the $\mathcal{J}$-unitary transformation and broadcasts it to the rest. The rows from i to n of the generator are updated in parallel. Afterwards, the nonzero entries of the distributed first column of $\mathcal{G}$ are copied on to the i-th column (entries 1_i) without communication cost. When the n steps have been executed, we will have

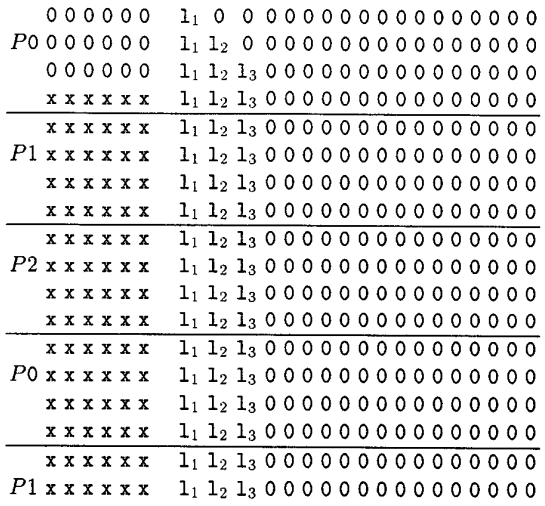


Fig. 1. *Block row cyclic distribution of an example generator G of size 18×6 (entries* x*) and a 18×18 lower triangular factor L (entries* l_i*) obtained by the Generalized Schur Algorithm, with a block size of $v=4$ rows over a 3×1 processor grid. The figure shows the distributed workspaces after the step 3.*

the upper triangular R factor transposed of the QR decomposition of the matrix $[b\ T]$ distributed over all processors.

Finally, in each step, the first column of generator $\mathcal{G}$ has to be shifted one position down. This last operation involves a critical communication cost with respect to the small computational cost of each iteration. Each processor has to send one element per block to the next, and it has to receive a number of messages equal to the number of blocks of the previous one. This implies a point to point communication of several messages of one element. In our parallel algorithm, each processor packs all the elements in one message. The destination processor receives it, unpacks the scalars and puts them into the destination blocks. In Fig. 2 we show the parallel shift process. The total number of messages is reduced from $O(\frac{n^2}{v})$ to $O(pn)$ and, therefore, the global latency time is also reduced.

3 Improving the Accuracy of the Method

When the R factor in (3) is ill-conditioned, we can expect a large error in the matrix G computed by Generalized Schur Algorithm. Therefore, we need to apply the correction step proposed in [8] in order to refine the R factor obtained by the Generalized Schur Algorithm described above.

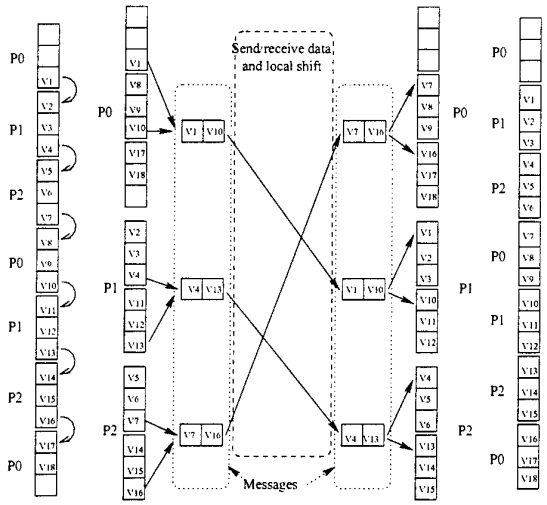


Fig. 2. *Shift of elements 3 to 20 of a vector of 24 elements distributed on a 3×1 processors grid. The size of block is v=3.*

We assume the following partition of the Toeplitz matrix T

$$T = \begin{pmatrix} t_0 & f_r^T \\ f_c & T_0 \end{pmatrix} = \begin{pmatrix} T_0 & l_c \\ l_r^T & t_{m-n} \end{pmatrix} ,$$

where $T_0 \in \mathrm{R}^{(m-1)\times(n-1)}$ is a Toeplitz submatrix of T, f_r, $l_r \in \mathrm{R}^{(n-1)\times 1}$, and f_c, $l_c \in \mathrm{R}^{(m-1)\times 1}$.

Indeed, we assume a partition of the matrix composed of the factor G and the vector ω in (2),

$$\begin{pmatrix} \omega^T \\ G \end{pmatrix} = \begin{pmatrix} \omega_1 & \omega_b^T \\ g_{11} & g_r^T \\ 0 & G_b \end{pmatrix} = \begin{pmatrix} \omega_t^T & \omega_n \\ G_t & g_c \\ 0 & g_{nn} \end{pmatrix} , \tag{5}$$

where $G_b, G_t \in \mathrm{R}^{(n-1)\times(n-1)}$, $g_r, g_c, \omega_b, \omega_t \in \mathrm{R}^{(n-1)\times 1}$, $g_{11}, g_{nn}, \omega_1, \omega_n \in \mathrm{R}$, and define matrix $\hat{G} \in \mathrm{R}^{(n-1)\times(n-1)}$ such as

$$\hat{G}^T\hat{G} = G_t^T G_t + \omega_t \omega_t^T + f_r f_r^T . \tag{6}$$

It can be shown that the matrix $\hat{G}$ is the upper triangular R factor of the QR decomposition of matrix X,

$$\hat{G} = \mathrm{qr}(X) , \quad X = \begin{pmatrix} f_r^T \\ T_0 \\ l_r^T \end{pmatrix} \in \mathrm{R}^{(m+1)\times(n-1)} . \tag{7}$$

Basically, the refinement step proceeds as follows. First of all, we have to find an orthogonal matrix $\hat{W}^T$ which transforms the first two columns of $[b\ T]$ to upper triangular form and, accordingly, the first two rows into

$$\begin{pmatrix} \kappa & \omega_1 & \omega_b^T \\ 0 & g_{11} & g_r^T \end{pmatrix} . \tag{8}$$

We define a matrix W_1 as

$$W_1 = \begin{pmatrix} 0 & w_1 & w_2 \\ 1 & 0 & 0 \end{pmatrix} , \tag{9}$$

where w_1 and w_2 denote the first and second columns of matrix $\hat{W}$.

Once the factor G_t (5) has been obtained from the Generalized Schur Algorithm, the matrix $\hat{G}$ is computed by updating G_t via Givens rotations. Then, a *refined* matrix G_b (5) is obtained by downdating the block H from $\hat{G}$,

$$G_b^T G_b = \hat{G}^T \hat{G} - H^T H ,$$

where $H^T = \begin{pmatrix} l_r & g_r & \omega_b \end{pmatrix} \in \mathrm{R}^{(n-1)\times 3}$.

The downdating process is performed by solving the LS problem

$$\min_V \|W_1 - XV\|_F , \tag{10}$$

with the Corrected Semi-Normal Equations method (CSNE).

Once the LS problem (10) is solved, we obtain an orthogonal matrix $Q_1 \in \mathrm{R}^{(n-1)\times 3}$ and an upper triangular matrix $\Gamma \in \mathrm{R}^{3\times 3}$, and we construct the following matrix

$$\begin{pmatrix} Q_1 & \hat{G} \\ \Gamma & 0 \end{pmatrix} , \tag{11}$$

that we have to triangularize by a product of Givens rotations M,

$$M \begin{pmatrix} Q_1 & \hat{G} \\ \Gamma & 0 \end{pmatrix} = \begin{pmatrix} I_3 & \hat{H} \\ 0 & G_b \end{pmatrix} , \tag{12}$$

in order to obtain the *refined* factor G_b and, therefore, a more accurate R factor of the QR decomposition of the matrix T.

All steps described above can be summarized in the following algorithm.

Algorithm 1 (CSNE Refinement Step).
Let G_t denote the triangular factor (5) computed from the Generalized Schur Algorithm, and W_1 the matrix defined in (9) and let X from (7), the refinement step proceeds as follows:

1. *Compute $\hat{G}$ triangularizating $(G_t^T\ \omega_t\ f_r)$.*
2. *Compute Q_1, V and F from*

$$\hat{G}^T Q_1 = X^T W_1 , \quad \hat{G} V = Q_1 , \quad F := W_1 - XV .$$

3. (a) Update Q_1, V, and F:

$$\hat{G}^T Q_1' = X^T F\ , \quad Q_1 := Q_1 + Q_1'\ ,$$
$$\hat{G} V' = Q_1'\ , \qquad F := F - XV'\ .$$

(b) Compute the upper triangular factor Γ of the QR decomposition of F.

4. Triangularize $\begin{pmatrix} Q_1 & \hat{G} \\ \Gamma & 0 \end{pmatrix}$ *by a product of Givens rotations M:*

$$\begin{pmatrix} I_3 & \hat{H} \\ 0 & R \end{pmatrix} := M \begin{pmatrix} Q_1 & \hat{G} \\ \Gamma & 0 \end{pmatrix}\ .$$

The computational complexity of the Generalized Schur Algorithm and the *Refinement Step* of Algorithm 1 is $13mn + 24.5n^2$ flops [8].

4 The Parallel Refinement Step

With the correction step proposed in [8] we have an overcost in the global algorithm to solve de LS problem that we can aliviate using parallelism. The keys to obtain a good parallel version of the correction process are, first, an adequate distribution of the involved data and, second, the construction of new appropriate parallel computational kernels to solve each step of the algorithm.

A cyclic row block of size v (v must be greater of equal to 3 ever) distribution have been used (Fig 3). The workspace proposed is divided into two distributed sub-workspaces: the *generator*, the entries of which are denoted by G and the *rest*, which entries q and Γ denote the entries of the matrices Q_1 and Γ (11) once they have been calculated and before applying transformation M (12). The entries $\hat{\mathtt{G}}$ denote the triangular factor obtained by the Parallel Generalized Schur Algorithm that will be updated later in order to obtain the factor $\hat{G}$ defined in (6). The rest of the entries are distributed matrices for auxiliar purposes as it is described in Algorithm 2. Indeed, a local workspace called F of size $(m+1) \times 3$ is used by each processor.

Algorithm 2 (Parallel Refinement Step).
Given the triangular factor G_t^T computed on workspace $\hat{\mathtt{G}}$ (Fig. 3) by the Parallel Generalized Schur Algorithm and a matrix W_1 (9) replicated on local workspace F *in all processors, this algorithm obtains the refined triangular factor G_b^T (5) arising in entries* $\mathtt{G}_b$ *of workspace shown in Fig. 4.*

1. *Update the distributed factor* $\hat{\mathtt{G}}$ *by triangularizating the distributed matrix* $(\hat{\mathtt{G}}\ \mathtt{A}_1\ \mathtt{A}_2)$, *where columns* $\mathtt{A}_1$ *and* $\mathtt{A}_2$ *(first two columns of workspace* A*) have ω_t and f_r respectively, by parallel Givens rotations in order to form the triangular factor $\hat{G}^T$ (6).*
2. *(a) Compute* $\mathtt{A} := X^T\mathtt{F}$. *All processors know matrix X because it is not explicitly formed, and have matrix W_1 replicated on the local workspace* F. *Each processor can calculate its local blocks of the distributed matrix* A *without communications.*

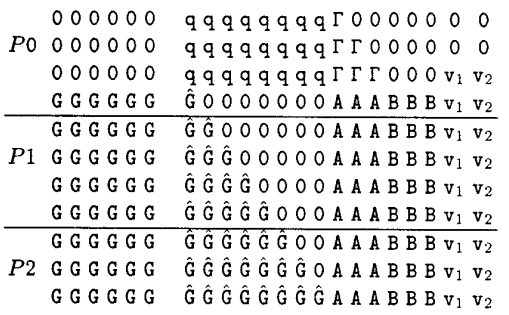


Fig. 3. *Parallel workspace for refinement step over a 3×1 processors grid. The size of block is 4. Matrices are stored in memory in transposed form.*

(b) Solve the triangular linear system $\mathtt{A} := \hat{\mathtt{G}}^{-1}\mathtt{A}$ *in parallel.*

(c) Solve the triangular linear system $\mathtt{A} := \hat{\mathtt{G}}^{-T}\mathtt{A}$ *in parallel.*

(d) Perform the operation $\mathtt{F} := \mathtt{F} - X\mathtt{A}$. *Each processor* P_k *calculates locally a factor* F_k, $k = 0, \ldots, p-1$ *and, by a global sum* $\mathtt{F} := \sum_{k=0}^{p-1} F_k$, *all processors will have the factor* F *replicated after this step.*

3. *Save factor* A *into* B.

4. *(a) i. Compute* $\mathtt{A} := X^T\mathtt{F}$. *Each processor can calculate its local blocks of the distributed matrix* A *without communications.*

ii. Solve the triangular linear system $\mathtt{A} := \hat{\mathtt{G}}^{-1}\mathtt{A}$ *in parallel.*

iii. Update factor B, $\mathtt{B} := \mathtt{B} + \mathtt{A}$.

iv. Solve the triangular linear system $\mathtt{A} := \hat{\mathtt{G}}^{-T}\mathtt{A}$ *in parallel.*

v. Perform the operation $\mathtt{F} := \mathtt{F} - X\mathtt{A}$. *Each processor* P_k *calculates locally a factor* F_k, $k = 0, \ldots, p-1$ *and, by a global sum* $\mathtt{F} := \sum_{k=0}^{p-1} F_k$, *the factor* F. *In this case, only the processor* P_0 *will have the resulting factor* F *of the global sum.*

(b) If $P_k = P_0$, *calculate triangular factor* Γ *of the QR decomposition of* F *and copy it transposed on entries denoted by* Γ *in Fig. 3.*

5. *(a) Redistribute factor* B *to entries denoted by* q *in Fig. 3. These entries are owned by processor* P_0 *always because in the distribution chosen* $v \geq 3$.

(b) Set the entries A *to zero and triangularize the workspace formed by entries* q, Γ, $\hat{\mathtt{G}}$ *and* A, *the matrix defined in (12) in transposed form, by a product of parallel Givens rotations* M *in order to obtain the triangular factor* $\mathtt{G}_b$. *The result of this operation can be seen in Fig. 4.*

Solving the *Refinement Step* in parallel by Algorithm 2 involves several basic computations that we summarize below:

- Two matrix-matrix multiplications of the matrix X^T (7) by the local workspace F. These computations are performed in parallel without communications because matrix X is not explicitly formed and the second matrix is local and replicated in all processors.

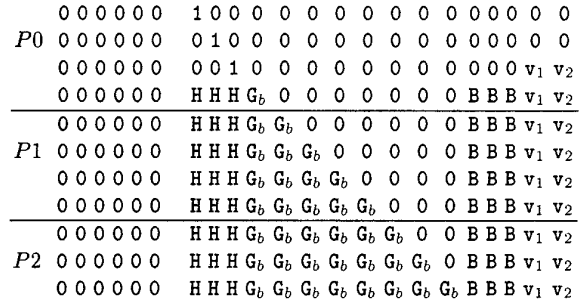


Fig. 4. *Parallel workspace of Fig. 3 after the refinement step has been carried out.*

- Three triangular linear systems involving the distributed matrix denoted in Fig. 3 with entries $\hat{\mathtt{G}}$ and the distributed matrix denoted with entries A. We have applied the corresponding PBLAS routines to perform these operations in parallel.
- Two matrix-matrix multiplications involving the matrix X and the distributed matrix A. Each processor computes a summation and, by means of a global sum, the result is distributed over all processors in the first case, or only stored on processor P_0 in the second one.
- Several parallel Givens rotatios appearing in steps 1 and 5b. This steps have been performed by blocks in order to minimize the number of messages needed to broadcast the Givens rotations.

The main problem of the previous algorithm is the large number of different basic computations to perform and the number of different matrices and vectors involved. We have distributed all data in order to optimize the different steps and to minimize the communication cost. Standard routines from ScaLAPACK and PBLAS have been used for matrix distributions and solving distributed triangular linear systems. For matrix-matrix multiplications and for triangularization of workspaces with Givens rotations described above we have implemented several specific routines to solve these basic tasks. A more detailed description of these last computational kernels can be seen in [2].

5 The Parallel Algorithm

In this section we show a very summarized version of the whole parallel algorithm. Step 2 of Algorithm 3 corresponds to the Parallel Generalized Schur Algorithm described in section 2, while step 3 corresponds to the *Refinement Step* described in the previous one.

Algorithm 3 (Parallel Algorithm for the Toeplitz LS Problem).
Given a Toeplitz matrix $T \in R^{m \times n}$ and an arbitrary vector $b \in R^m$, this algo-

rithm computes in parallel the solution vector $x \in R^n$ of the LS Problem (1) in a stable way.

1. *Compute values κ, ω and $g = \begin{pmatrix} g_{11} & g_r^T \end{pmatrix}$ (8) by a QR decomposition of the first two columns of $[b\ T]$. Save ω in the distributed workspace* $\mathtt{v}_1$ *and g in the distributed workspace* $\mathtt{v}_2$ *(Fig. 3). Form the generator $\mathcal{G}$ of the displacement matrix ∇_Z (4) distributed over the workspace denoted by entries* G *in Fig. 3.*
2. *Compute the triangular factor $G_t \in R^{(n-1)\times(n-1)}$ (5) on to the distributed workspace* $\hat{\mathtt{G}}$ *using the Parallel Generalized Schur Algorithm described in section 2.*
3. *Compute the triangular factor $G_b \in R^{(n-1)\times(n-1)}$ (5) on to workspace* $\mathtt{G}_b$ *(Fig. 4) applying the parallel* Refinement Step *described in section 4.*
4. *Using the scalar κ, the vectors ω and g stored on* $\mathtt{v}_1$ *and* $\mathtt{v}_2$ *respectively, and the* refined *factor* $\mathtt{G}_b$*, compute the operation described in (3) via a product of parallel Givens rotations J and solve the triangular linear system $Rx = b$ in parallel (using PBLAS) in order to obtain the vector solution x of the LS problem (1).*

6 Experimental Results

First of all, we have tested our parallel algorithm by using the matrices proposed in [8], concluding that the parallel version preserves the stability properties of the sequential algorithm.

In this section we show the experimental results we have obtained with our parallel algorithm using a cluster of 32 personal computers with a Pentium II microprocessor connected through a Myrinet network using point to point communications. This environment has a good relation between computation and communication cost and also between prize and performance. Besides, this type of multicomputer can be easily actualized and it allows the use of standard libraries like MPI and ScaLAPACK over the Linux Operating System.

In Table 1 we compare the results of our parallel algorithm with PDGELS, a parallel routine for solving the general least squares problem included on the ScaLAPACK library. We can see that efficiency of PDGELS is better, but time is worst because the algorithm included in ScaLAPACK does not have into account the structure the Toeplitz matrices.

We also show time and efficiency of Algorithm 3 with matrices of different sizes and relations between m and n versus different number of processors (Table 2). All the results are obtained using the best block size (v) in each case. We can observe that, while there is a good efficiency with a few processors, this factor decreases when the number of processors increases. Indeed, the results improve when we increase the size of the matrices and when $m >> n$ (Fig. 5). This behaviour of the parallel algorithm is due, mainly, to the low computational cost of the sequential algorithm. We are trying to parallelize and algorithm that exploits the special structure of the Toeplitz matrices and that reduces the cost from $O(mn^2)$ to $O(mn)$. It is widely known that in a distributed system, a

Table 1. Comparison between PDGELS routine and Algorithm 3. Table shows time in seconds and efficiency for 1, 2, 4, 8 and 16 processors. The best block size and logical grid have been used for PDGELS routine.

Algorithm 3									
$m \times n$	1	2		4		8		16	
1000 × 100	0.143	0.105	68%	0.086	42%	0.075	24%	0.083	11%
1000 × 500	0.868	0.674	64%	0.463	47%	0.407	26%	0.341	16%
1000 × 1000	2.144	1.715	62%	1.173	46%	0.988	27%	1.016	13%
PDGELS routine from ScaLAPACK									
$m \times n$	1	2		4		8		16	
1000 × 100	0.272	0.162	84%	0.112	61%	0.104	33%	0.104	16%
1000 × 500	3.121	1.838	85%	1.263	62%	0.923	42%	0.778	25%
1000 × 1000	8.010	5.272	76%	3.725	54%	2.301	43%	1.650	30%

very important factor to obtain good performance is to reduce the communication cost and/or, at least, to increase the relation between computational and communication costs. In the case of a sequential algorithm with so small computational cost, any communication introduced in the parallel version has an enormous effect.

Table 2. Parallel results in time (seconds) and efficiency for several Toeplitz matrices of different number of rows and columns.

$m \times n$	1	2		4		8		16	
1000 × 100	0.143	0.105	68%	0.086	42%	0.075	24%	0.083	11%
1000 × 500	0.868	0.674	64%	0.463	47%	0.407	26%	0.341	16%
1000 × 1000	2.144	1.715	62%	1.173	46%	0.988	27%	1.016	13%
2000 × 200	0.600	0.383	78%	0.249	60%	0.196	38%	0.178	21%
2000 × 1000	3.630	2.555	71%	1.580	57%	1.223	37%	1.068	21%
2000 × 2000	9.225	6.893	67%	4.151	55%	3.124	37%	2.922	20%

In order to perform a more detailed analysis, we offer the impact of intriducing the *Refinement Step* in the parallel algorithm. In Table 3 we show the Generalized Schur Algorithm cost separate from the *Refinement Step* cost. We can see that the sequential cost of the *Refinement Step* is so much higher than the cost of the Generalized Schur Algorithm. Therefore, there is a better opportunity of obtaining parallel speedup during the Step 3 of the Algorithm 3. The parallelization of the Generalized Schur Algorithm offers worse results due to the greater influence of the communications than the parallelization of the *Refinement Step* as we increase the number of processors. The necessity of broad-

casting the transformation factors on each iteration and the displacement of the first column of the generator involves a large communication cost related to the very small computational cost of applying the transformations.

Table 3. Comparison of time in seconds and efficiency between Step 2 and Step 3 of Algorithm 3.

$m \times n$		1	2		4		8		16	
2000 × 200	Schur	0.016	0.028	29%	0.030	13%	0.036	6%	0.043	2%
	Refi.	0.522	0.331	79%	0.199	65%	0.147	44%	0.123	27%
2000 × 1000	Schur	0.401	0.296	68%	0.249	40%	0.245	20%	0.257	10%
	Refi.	2.915	1.817	80%	1.050	69%	0.718	51%	0.579	31%
2000 × 2000	Schur	1.583	0.972	81%	0.694	57%	0.602	33%	0.579	17%
	Refi.	6.909	4.382	79%	2.542	68%	1.739	50%	1.370	32%

In order to fully exploit the parallel system, we have also analyzed the influence of scaling the problem with the number of processors. Specifically we have used a isotemporal scale, increasing the size of the problem in order to maintain the temporal cost of the parallel algorithm. The cost of the problem is $O(mn)$, so we have scaled both factors, m and n, with the square root of the number of processors. In Fig. 5 we show the scaled speedup that we have obtained in the

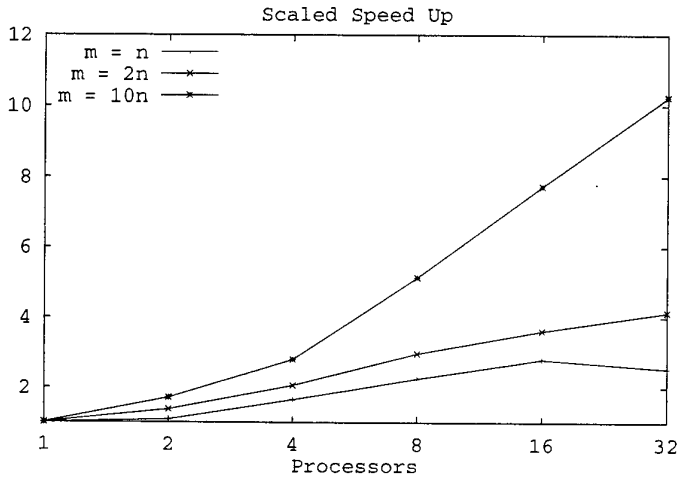


Fig. 5. *Scaled Speed Up of isotemporal cost of different example matrices varying in the relation between m and n.*

case of matrices with a different relation between m and n. Specifically we show the results with square matrices ($m = n$), with matrices in which $m = 2n$ and with very rectangular matrices ($m = 10n$). In all cases, we start with a matrix in which $n = 400$ in the sequential case. The results are not close to the optimum, but we obtain scaled speed up greater than 10 with 32 processors. Given de specific characteristics of the parallelized algorithm, this is a good result.

The figure also shows that we obtain better results with rectangular matrices than with square matrices. This behaviour of the algorithm shows that the computation cost of the parallel algorithm depends on both factors, m and n, while the communication factor increases mainly with n. Indeed, during the first phase of the algorithm the communications only depend on n.

7 Conclusions

In this paper we have presented a new parallel algorithm for solving the least squares problem with Toeplitz matrices. This algorithm exploits the special structure of this class of matrices in order to reduce the cost obtaining the solution of the problem and is mainly based on the parallelization of the method presented in [8].

We have parallelized the two main phases of the method: the Generalized Schur Algorithm and the refinement process to obtain a more accurate result. The parallel algorithm maintains the stability properties of the sequential method and, therefore, offers a similar accuracy than using the QR method based on Householder transformations. The parallel algorithm has been developed using a standard environment, thus producing a portable code to different parallel environments. We have used the ScaLAPACK library based on the MPI message-passing library. However, the specific characteristics of the algorithm does not allow fully exploit the bidimensional parallel model of the ScaLAPACK library. The second step of the Algorithm 3 is based on the transformation of the *generator* of the Toeplitz matrix that has $n - 1$ rows, but only six columns. Therefore, we had had to use a unidimensional grid in order to approach this phase of the algorithm. During the *Refinement Step* we have tried to reduce the communication cost by using an appropriate workspace and we have combined ScaLAPACK routines with other routines designed specifically to approach the different stages of the refinement. In order to apply the Givens rotations it is suitable to use the same logical grid of $p \times 1$ processors. An experimental analysis has been carried out in a cluster of personal computers based on a high performance interconnection network. The results show good efficiencies with a reduced number of processors, but they are not so good when we use a large number of processors. The main reason of this behaviour is the very small computational cost of the algorithm that we have parallelized. However, as we have shown, time of Algorithm 3 is less than the corresponding of standard routine and, using the parallel algorithm proposed, the impact in the computational cost of the correction post-process is reduced.

References

1. Pedro Alonso, José M. Badía, and Antonio M. Vidal. Un algoritmo paralelo estable para la resolución de sistemas de ecuaciones toeplitz no simétricos. In *Actas del VI Congreso de Matemática Aplicada (CMA), Las Palmas de Gran Canaria*, volume II, pages 847–854, 1999.
2. Pedro Alonso, José M. Badía, and Antonio M. Vidal. Algoritmos paralelos para la resolución de sistemas lineales y del problema de mínimos cuadrados para matrices toeplitz no simétricas. Informe Interno (Technical Report) II-DSIC-2/2000, Universidad Politécnica de Valencia, January 2000.
3. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.
4. Richard P. Brent. Stability of fast algorithms for structured linear systems. In T. Kailath and A. H. Sayed, editors, *Fast Reliable Algorithms for Matrices with Structure*, pages 103–116. SIAM, 1999.
5. S. Chandrasekaran and Ali H. Sayed. A fast stable solver for nonsymmetric Toeplitz and quasi-Toeplitz systems of linear equations. *SIAM Journal on Matrix Analysis and Applications*, 19(1):107–139, January 1998.
6. J. Chun, T. Kailath, and H. Lev-Ari. Fast parallel algorithms for QR and triangular factorization. *SIAM Journal on Scientific and Statistical Computing*, 8(6):899–913, November 1987.
7. Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. Technical report, University of Tennessee, Knoxville, Tennessee, March 94.
8. L. Eldén and H. Park. Stability analysis and fast algorithms for triangularization of toeplitz matrices. Technical Report LiTH-MAT-R-95-16, Department of Mathematics, Linköping University, 1995.
9. L. S. Blackford et al. ScaLAPACK: A portable linear algebra library for distributed memory computers — design issues and performance. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press.
10. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*, volume 3 of *Johns Hopkins Series in the Mathematical Sciences*. The Johns Hopkins University Press, Baltimore, MD, USA, second edition, 1989.
11. Thomas Kailath and Ali H. Sayed. Displacement structure: Theory and applications. *SIAM Review*, 37(3):297–386, September 1995.
12. J. Schur. Über Potenzreihen, die im Innern des Einkeitskreise beschänkt sind. *Journal für die reine und angewandte Mathematik*, 147:205–232, 1917.
13. D. R. Sweet. Fast Toeplitz orthogonalization. *Numerische Mathematik*, 43(1):1–21, 1984.

Parallel Preconditioning of Linear Systems Appearing in 3D Plastic Injection Simulation

D. Guerrero, V. Hernández, J. E. Román, and A. M. Vidal

Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia,
Camino de Vera, s/n, E-46022 Valencia, Spain.
Tel. +34-96-3877356, Fax +34-96-3877359
(dguerrer,vhernand,jroman,avidal)@dsic.upv.es

Abstract. Plastic injection has been simulated for a long time. However, this has not been the case for short fiber reinforced thermoplastics, because the injection of these materials is a much more complex process. Until now, numerical simulation of the suspension flows has been carried out by several techniques, but using simplified models or treating particular geometries such as plate moulds. The objective of HIPERPLAST –an EU-funded project– was the development of a much more general HPCN-based simulator valid for generic 3D moulds. The linear systems which appear in these simulations are very badly conditioned. In this work, a combination of solver and preconditioner is sought which is appropriate for this particular application.

Topics. Computational fluid dynamics, numerical methods, parallel and distributed algorithms.

1 Introduction

This work deals with the solution of symmetric indefinite linear systems of equations by means of Krylov-type iterative methods. Indefinite systems are almost always difficult to solve for iterative methods and it seems that finding a general method for this kind of systems is still an open problem. Therefore, several methods proposed by different authors have been used to solve a specific problem with the aim of finding which ones are the most competitive for this particular case.

The comparison has been carried out having in mind that a parallel implementation is highly recommended, due to the dimension of the problems to be solved. Therefore, the study does not take into account those methods or preconditioning techniques, such as ILU, which are known to have less potential parallelism.

In particular, the application addressed by this work is the simulation of the injection of short fiber reinforced thermoplastics. The application and the initial results are described in section 2. The structure of the rest of the text is the following. In section 3, some different Krylov subspace methods are compared

in terms of convergence rate. Sections 4 and 5 deal with sequential and parallel preconditioning techniques, respectively. Finally, section 6 summarises the results.

2 The HIPERPLAST project

This section gives a brief description of the HIPERPLAST project. Further information can be found at the project's web site[1].

2.1 Framework and Objectives

HIPERPLAST was an EU-funded project under the HPCN–TTN Network initiative[2] of the ESPRIT IV programme. In this initiative, more than 100 demonstrators were developed with the main focus on showing to different industrial sectors across Europe how HPCN technology can provide them with substantial benefits.

The particular objective of HIPERPLAST was the development of a parallel code for the simulation of the injection of short fibre reinforced thermoplastics. Although the outcome of the project was a functioning prototype, the fact that it spanned only 18 months prevented from analysing more sophisticated numerical methods which could be more suitable for this specific application. This is the aim of the present work.

In the rest of this section, a description of the project is given, first from the point of view of the industrial application and the benefits expected by the industry, and then from the technical perspective.

2.2 The Industrial Problem

Reinforced plastics have many applications and can be found in a wide range of industrial products, for example in vehicle toys such as cars or bikes. Due to the high stresses supported by these kinds of products it is necessary to make use of some structural parts made of stainless steel or aluminium, which pose problems such as higher production costs and dependence on external suppliers of metallic components. The use of *short fibre reinforced thermoplastics* can alleviate these problems, allowing to define the structure as an assembly of metallic tubes and joints made of this kind of plastics. By doing this, metallic parts become less complex and common to several different toy designs, thus reducing stocks. This last issue is important for the toy industry because of the short life cycle of the product.

Design of this kind of parts is based on a trial-and-error prototyping process. The designer uses intuition and expertise to make an initial guess of the appropriate shape of the part and parameters of the injection process (e.g. the number

[1] http://hiperttn.upv.es/hiperplast
[2] http://www.hpcn-ttn.org

and position of the injectors). After modelling the geometry with a CAD tool, the mould is manufactured and the prototype can be injected and tested to see if it fulfils the required specifications. The initial design is modified according to the results of the tests. The process typically takes several iterations until a satisfactory result is achieved. The main drawbacks of this procedure are its cost (2000-3000 Euro per mould) and its length in time (more than fifteen days per mould).

As in many other situations, this expensive trial-and-error procedure can be replaced by a similar procedure which uses software tools instead of physical experimentation. In this case, the determination of fiber orientation by means of computer programmes can give the operator valuable hints about the injection process. Fiber orientation, caused by processing, has a serious influence on the final mechanical properties of reinforced components. For this reason the prediction of fiber orientation using flow simulation software is an important tool for the designer.

Simulation makes the design and the moulding process control easier, faster and more reliable. It enables the design engineer to study different variants and analyse the influences of input parameters at a very early stage of development. Therefore errors in the process and failures of the final part can be predicted in the simulation, leading to

- shorter development time,
- minimized mould changes,
- enhanced part quality, and
- significant cost reduction.

Computer simulators for injection of plastics are quite common. However, only few commercial packages can cope with reinforced materials. Some of them are specific for a plastic processing technique, such as EXPRESS which is intended for compression moulding or UFOS-GT for the case of thermoforming[3].

With respect to injection moulding, the commercial simulators (MOLDFLOW, CADMOULD) that claim to cope with reinforced materials perform the simulation by considering only the surface of the mould, without taking into account the interior flow.

The surface approach is not sufficient for real applications and a 3D modelling is necessary. This increases considerably the complexity of the problem and requires to solve large systems of linear equations which today can only be solved in a reasonable time by using parallel codes. As far as the authors know, HIPERPLAST is the first multi-disciplinary effort to approach this application.

2.3 The HIPERPLAST Simulator

During the HIPERPLAST project, a parallel simulator was developed to compute the orientation of fibers during the injection moulding process [1].

[3] http://www.rwth-aachen.de/ikv

The injection process can be modelled by the flow equations, which define an anisotropic Stokes problem, coupled with the fibre orientation equation and the volumetric fraction equation. An explicit discontinuous Taylor-Galerkin strategy has been used in order to decouple these equations [9]. With this scheme, the main simulation algorithm consists in a loop which advances the time. In each iteration, the code solves a linear system of equations corresponding to the space discretisation in the mould domain.

The space discretisation has been done by the Finite Element Method, with a non standard interpolation scheme for the velocity components (linear interpolation plus a bubble function with $\mathcal{C}^0$ continuity), a conforming linear approximation for the pressure and a constant approximation for the orientation tensor and the volumetric fraction.

The discretised kinematics equations of the flow give a relation for pressures and velocities. When assembling the stiffness matrix, the degrees of freedom associated to the bubble functions can be easily distinguished since they are decoupled from each other. After applying the boundary conditions, the resulting system of equations can be written in the following form

$$\begin{bmatrix} \boldsymbol{M} & \boldsymbol{J} \\ \boldsymbol{J}^T & \boldsymbol{D} \end{bmatrix} \begin{bmatrix} \boldsymbol{n} \\ \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \end{bmatrix} \ , \tag{1}$$

where the nodal quantities, velocities $\boldsymbol{v}$ and pressures $\boldsymbol{p}$, have been grouped in vector $\boldsymbol{n}$, distinguishing them from the bubble velocities $\boldsymbol{b}$. In (1), $\boldsymbol{D}$ is a symmetric block diagonal matrix with 3×3 diagonal blocks, $\boldsymbol{M}$ is a symmetric sparse matrix and $\boldsymbol{J}$ is a rectangular sparse matrix. In order to benefit from the structure of matrix $\boldsymbol{D}$, a block Gaussian elimination is applied, yielding the following set of equations

$$(\boldsymbol{M} - \boldsymbol{J}\boldsymbol{D}^{-1}\boldsymbol{J}^T)\boldsymbol{n} = \boldsymbol{w}_1 - \boldsymbol{J}\boldsymbol{D}^{-1}\boldsymbol{w}_2 \ , \tag{2}$$

$$\boldsymbol{J}^T\boldsymbol{n} + \boldsymbol{D}\boldsymbol{b} = \boldsymbol{w}_2 \ . \tag{3}$$

The coefficient matrix $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{J}\boldsymbol{D}^{-1}\boldsymbol{J}^T$ has the same pattern as $\boldsymbol{M}$ and can be computed explicitly without difficulty. This matrix is also symmetric and indefinite.

For the parallelisation, a domain decomposition strategy was chosen because it preserves the locality of the problem, which is of great importance for the efficiency of the implementation. Groups of elements are clustered to form aggregations that are mapped to each processor, and inter-processor communications overhead is minimised by appropriate decomposition of the geometric domain. The decomposition process can be seen as an attempt to maximise the ratio of domain volume to surface area of contact between sub-domains. This was accomplished by using the graph partitioning algorithms implemented in the METIS package [7].

For the scalability of the linear system solution task a Krylov iteration scheme was used. In particular, the initial implementation used the Conjugate Gradient method. For matrix-vector products, communication is required for components of the vector corresponding to nodes in the inter-processor interfaces. In order

to avoid performance degradation, an appropriate ordering of internal, local boundary and external boundary unknowns is necessary. In addition to this, dot products and vector norms represent a synchronisation point in the parallel algorithm.

Preconditioning is the most problematic part of parallelising an iterative method. In a first approach, point-Jacobi preconditioning was used because it is trivially parallelisable.

Fig. 1 shows the performance of the MPI implementation in a SGI Origin 2000 multiprocessor for two test cases, TB2 with a 413392 order coefficient matrix and TB1 with size 955929. It can be observed from the picture that efficiency maintains above 70 %.

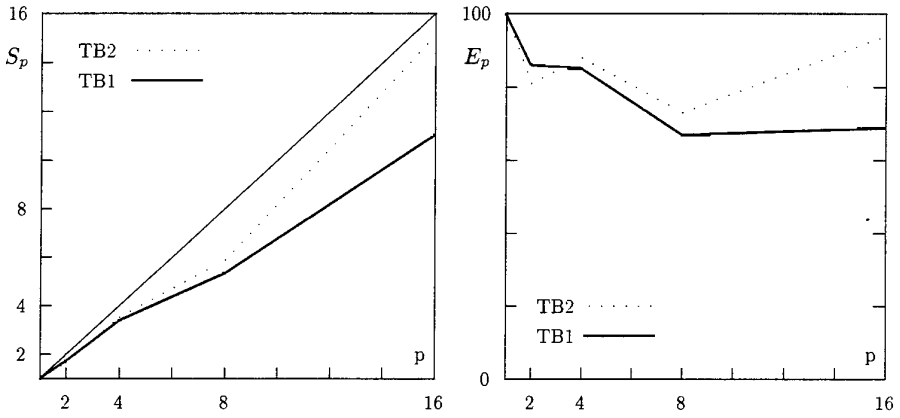


Fig. 1. Speedup and efficiency (in %) achieved in an SGI Origin 2000.

2.4 Limitations of the Initial Developments

The conditioning of the linear systems is quite bad. As an example, the following condition numbers were computed for the first iteration of a simulation with a small test case, 8484 degrees of freedom:

$$\kappa_2(\boldsymbol{M}) = 10^{27}, \;\; \kappa_2(\boldsymbol{A}) = 10^{9}, \;\; \kappa_2(\boldsymbol{\Lambda}^{-1}\boldsymbol{A}) = 10^{3} \;\; , \tag{4}$$

where $\boldsymbol{M}$ is the upper left block in (1), $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{J}\boldsymbol{D}^{-1}\boldsymbol{J}^T$ and $\boldsymbol{\Lambda} = \mathrm{diag}(\boldsymbol{A})$. These numbers suggest that the diagonal preconditioning improves considerably the conditioning of the problem.

The combination of Krylov Subspace Method and Preconditioner used for the simulator prototype were chosen after comparing the results obtained in several experiments carried out in a Matlab-type environment. For practical

reasons, only small-sized problems were treated in these experiments. However, this can be misleading. Small-sized problems are not the kind of problems to be solved by iterative methods in a production environment. Usually, they can be solved much better by direct methods. Moreover, the spectral properties of the coefficient matrices of small test problems are typically not representative of the difficulties present in larger problems that iterative methods are typically applied to. Small-sized systems often arise from coarse grids, but the spectral properties may change completely for finer grids.

In spite of this, the numbers get worse when dealing with large cases —some cases exceed 2 million degrees of freedom— and also as the simulation progresses. Therefore, a more elaborated preconditioning technique must be sought.

3 Comparison of Krylov Subspace Methods

We consider the linear system of equations (2), which can be written as

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \;\; \boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n \,, \tag{5}$$

where $\boldsymbol{A}$ is a large and sparse symmetric indefinite matrix. Due to the size of $\boldsymbol{A}$, direct solvers become prohibitively expensive and iterative methods are considered. Given the initial guess $\boldsymbol{x}_0$, these algorithms compute iteratively new approximations $\boldsymbol{x}_k$ to the true solution $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$. The iterate $\boldsymbol{x}_m$ is accepted as a solution if the residual $\boldsymbol{r}_m = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m$ satisfies $||\boldsymbol{r}_m||/||\boldsymbol{b}|| \leq tol$, where $tol = 10^{-10}$ in all the tests.

The comparison of different Krylov Subspace Methods as well as preconditioners has been carried out with the aid of the PETSc package [2], version 2.0.24. PETSc is a parallel library for the solution of mesh-based algebraic problems including linear and non-linear systems of equations. It allows the user to select different Krylov Subspace Methods and preconditioners at run time with command line options.

Some of the methods discussed in the next sections were not implemented in the PETSc Toolkit so they had to be coded manually following the conventions of this package.

3.1 MINRES and SYMMLQ

The Conjugate Gradient method can break down if the coefficient matrix has both positive and negative eigenvalues. Paige and Saunders [8] proposed two methods to handle large sparse indefinite symmetric matrices. These methods, MINRES and SYMMLQ, are shown in Fig. 2.

MINRES tries to determine $\boldsymbol{x}_k = \boldsymbol{V}_k\boldsymbol{y}_k$, $\boldsymbol{y}_k \in \mathbb{R}^k$, such that $||\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k||_2$ is minimized. This minimization leads to a small system with $\boldsymbol{T}_k$, the $k+1$ by k tridiagonal matrix associated to the Lanczos recurrence. The tridiagonal structure of matrix $\boldsymbol{T}_k$ is exploited to get a short recurrence relation for $\boldsymbol{x}_k$. The advantage of this approach is that only three vectors from the Krylov subspace have to be saved.

Choose $\boldsymbol{x}_0$	Choose $\boldsymbol{x}_0$
$\boldsymbol{x} = \boldsymbol{x}_0,\ \boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x},\ \rho = \|\|\boldsymbol{r}\|\|,\ \boldsymbol{v} = \boldsymbol{r}/\rho$	$\boldsymbol{x} = \boldsymbol{x}_0,\ \boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x},\ \rho = \|\|\boldsymbol{r}\|\|,\ \boldsymbol{v} = \boldsymbol{r}/\rho$
$\beta = 0,\ \widetilde{\beta} = 0,\ c = -1,\ s = 0$	$\beta = 0,\ \widetilde{\beta} = 0,\ c = -1,\ s = 0,\ \kappa = \rho$
$\boldsymbol{v}_{old} = \boldsymbol{0},\ \boldsymbol{w} = \boldsymbol{0},\ \widetilde{\widetilde{\boldsymbol{w}}} = \boldsymbol{v}$	$\boldsymbol{v}_{old} = \boldsymbol{0},\ \boldsymbol{w} = \boldsymbol{0},\ g = 0,\ \widetilde{\widetilde{g}} = \rho$
while $\|\rho\| > tol$ do	while $\kappa > tol$ do
$\widetilde{\boldsymbol{v}} \leftarrow \boldsymbol{A}\boldsymbol{v} - \beta\boldsymbol{v}_{old}$	$\widetilde{\boldsymbol{v}} \leftarrow \boldsymbol{A}\boldsymbol{v} - \beta\boldsymbol{v}_{old}$
$\alpha \leftarrow \boldsymbol{v} * \widetilde{\boldsymbol{v}},\ \widetilde{\boldsymbol{v}} \leftarrow \widetilde{\boldsymbol{v}} - \alpha\boldsymbol{v}$	$\alpha \leftarrow \boldsymbol{v} * \widetilde{\boldsymbol{v}},\ \widetilde{\boldsymbol{v}} \leftarrow \widetilde{\boldsymbol{v}} - \alpha\boldsymbol{v}$
$\beta \leftarrow \|\|\widetilde{\boldsymbol{v}}\|\|,\ \boldsymbol{v}_{old} \leftarrow \boldsymbol{v},\ \boldsymbol{v} \leftarrow \widetilde{\boldsymbol{v}}/\beta$	$\beta \leftarrow \|\|\widetilde{\boldsymbol{v}}\|\|,\ \boldsymbol{v}_{old} \leftarrow \boldsymbol{v},\ \boldsymbol{v} \leftarrow \widetilde{\boldsymbol{v}}/\beta$
$l_1 \leftarrow s\alpha - c\widetilde{\beta},\ l_2 \leftarrow s\beta$	$l_1 \leftarrow s\alpha - c\widetilde{\beta},\ l_2 \leftarrow s\beta$
$\widetilde{\alpha} \leftarrow -s\widetilde{\beta} - c\alpha,\ \widetilde{\beta} \leftarrow c\beta$	$\widetilde{\alpha} \leftarrow -s\widetilde{\beta} - c\alpha,\ \widetilde{\beta} \leftarrow c\beta$
$l_0 \leftarrow \sqrt{\widetilde{\alpha}^2 + \beta^2},\ c \leftarrow \widetilde{\alpha}/l_0,\ s \leftarrow \beta/l_0$	$l_0 \leftarrow \sqrt{\widetilde{\alpha}^2 + \beta^2},\ c \leftarrow \widetilde{\alpha}/l_0,\ s \leftarrow \beta/l_0$
$\widetilde{\boldsymbol{w}} \leftarrow \widetilde{\widetilde{\boldsymbol{w}}} - l_1\boldsymbol{w},\ \widetilde{\widetilde{\boldsymbol{w}}} \leftarrow \boldsymbol{v} - l_2\boldsymbol{w}$	$\widetilde{g} \leftarrow \widetilde{\widetilde{g}} - l_1 g,\ \widetilde{\widetilde{g}} \leftarrow -l_2 g,\ g \leftarrow \widetilde{g}/l_0$
$\boldsymbol{w} \leftarrow \widetilde{\boldsymbol{w}}/l_0$	$\boldsymbol{x} \leftarrow \boldsymbol{x} + (gc)\boldsymbol{w} + (gs)\boldsymbol{v}$
$\boldsymbol{x} \leftarrow \boldsymbol{x} + (\rho c)\boldsymbol{w},\ \rho \leftarrow s\rho$	$\boldsymbol{w} \leftarrow s\boldsymbol{w} - c\boldsymbol{v},\quad \kappa \leftarrow \sqrt{\widetilde{g}^2 + \widetilde{\widetilde{g}}^2}$
end while	end while

Fig. 2. The MINRES (left) and SYMMLQ (right) algorithms.

SYMMLQ determines $\boldsymbol{x}_k = \boldsymbol{A}\boldsymbol{V}_k\boldsymbol{y}_k,\ \boldsymbol{y}_k \in \mathbb{R}^k$, such that the error $\boldsymbol{x} - \boldsymbol{x}_k$ has minimum Euclidean length. It may come as a surprise that $\|\boldsymbol{x} - \boldsymbol{x}_k\|_2$ can be minimized without knowing $\boldsymbol{x}$, but this can be accomplished by restricting the choice of $\boldsymbol{x}_k$ to $\boldsymbol{A}\mathcal{K}_k(\boldsymbol{A}; \boldsymbol{b})$.

These two methods had to be implemented in order to carry out some test, since they are not included in PETSc. The results of the tests (see below) show that for the matrices tested the methods do not reach convergence in a reasonable amount of steps without the aid of a preconditioner.

3.2 Symmetric QMR

Both SYMMLQ and MINRES are based on the Lanczos process for symmetric matrices. Consequently, when preconditioning is used, then the coefficient matrix of the preconditioned system needs to be symmetric. This condition implies that the preconditioner itself needs to be a symmetric positive definite matrix. This restriction for the choice of possible preconditioners for SYMMLQ and MINRES is rather unnatural when the coefficient matrix itself is highly indefinite.

In [5], Freund and Nachtigal proposed an iterative method for solving symmetric indefinite linear systems with arbitrary symmetric preconditioners. The algorithm can be interpreted as a special case of the quasi-minimal residual (QMR) method for general non-Hermitian linear systems and generates iterates defined by a quasi-minimal residual property. The proposed method, which can be seen in Fig. 3, has the same work and storage requirements per iteration as SYMMLQ and MINRES, but it usually converges in considerably fewer iterations.

This method was also implemented because it is not included in PETSc. The convergence history of the three methods, MINRES, SYMMLQ and Symmetric

```
Choose x_0
x = x_0,  r = b − Ax,
Solve M_1 t = r,  τ = ||t||_2,  Solve M_2 q = t
θ = 0,  ρ = r^T q
while not converged do
    t ← Aq,  σ ← q^T t
    if σ = 0, stop
    α ← ρ/σ,  r ← r − αt
    Solve M_1 t = r,  θ ← ||t||_2/τ,  c ← 1/√(1 + θ^2),  τ ← τθc
    d = c^2 θ^2 d + c^2 αq,  x ← x + d
    if x converged, stop
    if ρ = 0, stop
    Solve M_2 u = t,  ρ_old = ρ,  ρ ← r^T u
    β ← ρ/ρ_old,  q ← u + βq
end while
```

Fig. 3. The Symmetric QMR algorithm.

QMR, with no preconditioning can be seen in Fig. 4. The graphs show the relative error $\|\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{b}\|/\|\boldsymbol{b}\|$ in a logarithmic scale for the first 2000 iterations. It can be seen that both MINRES and Symmetric QMR produce converge sequences very close to each other. On the other hand, SYMMLQ shows a very erratic history which at the end seems to diverge.

Since Symmetric QMR can accept preconditioners which are not necessarily positive definite, then it can be considered the best of the three methods for the particular application.

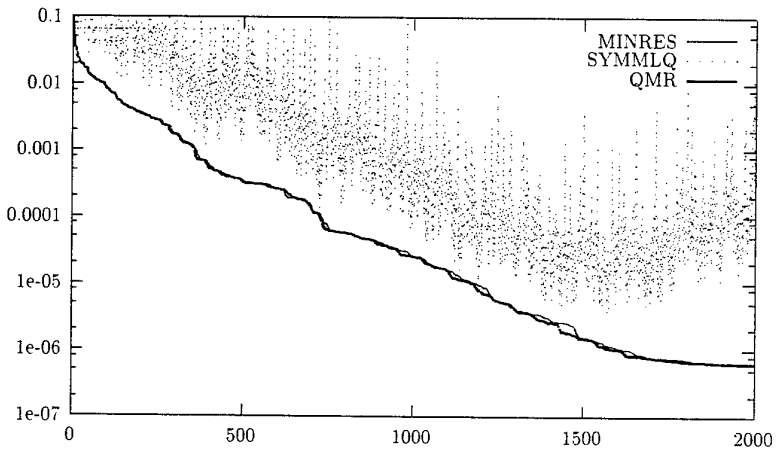


Fig. 4. The convergence behaviour of MINRES, SYMMLQ and Symmetric QMR.

4 Sequential Preconditioning

In general, when applying an iterative method to (5), convergence is not guaranteed or may be extremely slow, as it has been demonstrated in the previous section. Hence, the original problem must be transformed into a more tractable form, by applying a preconditioning matrix M either to the right or to the left of the linear system

$$\boldsymbol{AMy} = \boldsymbol{b}\,, \quad \boldsymbol{x} = \boldsymbol{My}\,, \quad \text{or} \quad \boldsymbol{MAx} = \boldsymbol{Mb}\,. \tag{6}$$

$\boldsymbol{M}$ should be chosen such that $\boldsymbol{AM}$ (or $\boldsymbol{MA}$) is a good approximation of the identity $\boldsymbol{I}$.

One preconditioner which can give reasonably good acceleration without much programming effort is Successive Over Relaxation (SOR) and its symmetric counterpart, SSOR. The tests show that for this particular application, a good choice for the relaxation factor is $\omega = 1$, in which case the method is equivalent to the Gauss-Seidel method.

In the case of SSOR, some optimisation can be done by using the so-called Eisenstat trick [3]. By using both left and right preconditioning of the linear system, this variant of SSOR requires about half of the floating point-operations for conventional SSOR.

Table 1 shows some results related to different sequential preconditioners. For the comparison, two Krylov-subspace methods have been used: GMRES and Symmetric QMR. GMRES is a valid method for general non-symmetric matrices and does not present problems due to indefiniteness. The combination Symmetric QMR and SOR did not reach convergence in less than 2000 iterations.

	GMRES		S-QMR	
	iter	time	iter	time
Jacobi	202	34,31	90	9,37
SOR	75	17,77	–	–
SSOR	75	18,63	47	12,32
ILU	32	20,01	43	28,3

Table 1. Comparison of different sequential preconditioners.

Both in GMRES and Symmetric QMR, the ILU preconditioning achieves convergence in less iterations. However, since a lot of computation is necessary in the initial factorisation step, then the comparison with respect to the total time is favourable to other preconditioning methods. In particular, SSOR seems to work quite well in both cases and this suggests that it can give good results in the parallel setting when combined with a block-Jacobi scheme.

5 Parallel Preconditioning

Due to the size of the problems to be solved, it is necessary to perform the computations in parallel and therefore to use a preconditioner suitable for parallel execution.

As the ultimate goal is to reduce the total execution time, both the computation of the preconditioner matrix $\boldsymbol{M}$ and its application to a vector $\boldsymbol{M}^{-1}\boldsymbol{y}$ should be done in parallel. Since the matrix-vector product must be performed at each iteration, the number of nonzero entries in $\boldsymbol{M}$ should not greatly exceed that in $\boldsymbol{A}$.

The most successful preconditioning methods in reducing solver iterations, e.g., incomplete LU factorizations or SSOR, shown in the previous section, are notoriously difficult to implement on a parallel architecture, especially for unstructured matrices. ILU, for example, can lead to breakdowns. In addition, ILU computes $\boldsymbol{M}$ implicitly, namely in the form $\boldsymbol{M} = \boldsymbol{U}_{approx}^{-1}\boldsymbol{L}_{approx}^{-1}$, and its application therefore involves solving upper and lower triangular sparse linear systems, which are inherently sequential operations.

Polynomial preconditioners with $\boldsymbol{M} = p(\boldsymbol{A})$, on the other hand, are inherently parallel, but do not lead to as much improvement in the convergence as ILU. In particular, the authors have implemented the polynomial preconditioning for the cases of Neumann polynomials and least squares polynomials. However, the tests performed with these two preconditioners have not shown a noticeable improvement in convergence rate. Also, some preliminary test were carried out with the SPAI preconditioner, [6], but it seems to achieve an acceleration similar to Jacobi preconditioning.

The preconditioners which have been compared with the test matrices are the following:

Jacobi. This is the point-Jacobi or diagonal preconditioning. It is the easiest preconditioner to implement but the reduction in the number of iterations is moderate. It is included as a reference for the other methods.

BJ-ILU. This is block-Jacobi with the incomplete LU in each block. Neither overlapping nor relaxation parameters have been used. There is one block per processor.

BJ-SSOR. This method is the same as BJ-ILU but carrying out a SSOR step in each block instead of the incomplete LU factorisation.

ASM. This is the Additive Schwarz Method. There is one sub-domain per processor.

Table 2 shows the number of iterations required for convergence with GMRES and each of these preconditioners. Also the execution times in an IBM SP system are shown for several values of the number of processors, p. The experiments were carried out with the test case TB2, with a coefficient matrix of order 413392. The lower part of the table shows similar results for test case TB1, with order 955929.

TB2	iter	$p=2$	$p=4$	$p=8$	$p=16$	$p=24$
Jacobi	107	9,74	5,21	3,01	1,83	1,64
BJ-ILU	30	5,72	3,22	1,45	0,82	0,61
BJ-SSOR	33	7,15	3,71	1,88	1,09	0,77
ASM	30	-	7,39	5,18	1,33	2,48

TB1	iter	$p=4$	$p=8$	$p=16$
Jacobi	279	29,92	16,32	9,34
BJ-ILU	119	24,16	12,11	6,18
BJ-SSOR	85	20,12	10,23	5,51
ASM	119	-	20,42	11,89

Table 2. Execution time with different parallel preconditioners for TB2 (up) and TB1 (down).

As it can be seen from the table, block-Jacobi with ILU is the best preconditioner in the medium case while for the large case block-Jacobi with SSOR reduces the number of iterations even more.

With respect to the scalability of the different solutions, Table 3 shows speed-up and efficiency in % corresponding to the execution times shown in Table 2. The figures have been computed relative to the times associated to the execution with less processors, $p = 2$ or $p = 4$. The execution times with $p = 1$ or $p = 2$, respectively, were not representative because the local memory allocated by the individual processes exceeded the physical memory available in each processor.

It can be seen in this table that block-Jacobi with SSOR scales quite well and block-Jacobi with ILU even obtains a better efficiency. The reason for this is that when the number of blocks (i.e. processes) increases, then the amount of work associated with the incomplete factorisation within the blocks decreases.

6 Conclusions

In this work, an appropriate method is sought for the solution of large sparse symmetric indefinite linear systems of equations which arise in the simulation of short fibre reinforced thermoplastics injection processes. The initially developed solver used the Conjugate Gradient method with Jacobi preconditioning, which provided with poor convergence rate.

Several Krylov subspace iterative methods, as well as sequential and parallel preconditioners, have been compared with a test battery corresponding to the particular application. To make the comparison, three iterative methods and two preconditioners were implemented in the PETSc Toolkit. The study presents the Symmetric QMR method as a good choice for the iterative method.

With respect to the preconditioner, block-Jacobi preconditioners have shown a good behaviour in terms of acceleration of convergence and also in terms of scalability. Other alternatives such as SPAI must be further investigated.

TB2	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 24$
Jacobi	1,00 (100%)	1,87 (93%)	3,24 (81%)	5,32 (66%)	5,93 (50%)
BJ-ILU	1,00 (100%)	1,78 (89%)	3,94 (99%)	6,97 (87%)	9,38 (78%)
BJ-SSOR	1,00 (100%)	1,93 (96%)	3,80 (95%)	6,56(82%)	9,28 (77%)
ASM	-	1,00 (100%)	1,43 (71%)	5,56 (>100%)	2,98 (50%)

TB1	$p = 4$	$p = 8$	$p = 16$
Jacobi	1,00 (100%)	1,83 (92%)	3,20 (80%)
BJ-ILU	1,00 (100%)	1,99 (99%)	3,91 (98%)
BJ-SSOR	1,00 (100%)	1,97 (98%)	3,65 (91%)
ASM	-	1,00 (100%)	1,72 (86%)

Table 3. Speedup and efficiency in % with different parallel preconditioners for TB2 (up) and TB1 (down).

Acknowledgements. The work described in this paper was partially supported by the European Commission through the HIPERPLAST project (ESPRIT 24003) and the Spanish Government Commission of Science and Technology under grant TIC96-1062-C03-01. We would like to thank the High Performance Computing Center North (Sweden), for providing us access to their IBM SP system.

References

1. E. Arias, V. Hernández, J. E. Román, A. M. Vidal et al. HIPERPLAST: An HPCN Simulator for Reinforced Thermoplastics Injection Processes. To appear in *Proceedings of the ParCo '99 Conference*, Delft, The Netherlands (1999).
2. S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pp. 163–202, Birkhauser Press (1997).
3. S. C. Eisenstat. Efficient Implementation of a Class of Preconditioned Conjugate Gradient Methods. *SIAM J. on Sci. and Stat. Comp.*, 2, pp. 1–4 (1984).
4. R. W. Freund. Preconditioning of Symmetric, but Highly Indefinite Linear Systems. 15th IMACS World Congress on Scientific Computation Modelling and Applied Mathematics, pp. 551-556 (1997).
5. R. W. Freund, and N. M. Nachtigal. A New Krylov-Subspace Method for Symmetric Indefinite Linear Systems. In Proc. of 14th IMACS World Congress (1994).
6. M. J. Grote, and T. Huckle. Parallel Preconditioning with Sparse Approximate Inverses. *SIAM Journal on Scientific Computing*, 18(3), pp. 838–853 (1997).
7. G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Par. and Dist. Comp.*, 48(1):96–129 (1998).
8. C. C. Paige, and M. A. Saunders. Solution of Sparse Indefinite Systems of Linear Equations. *SIAM Journal on Numerical Analysis*, 12(4), pp. 617–629 (1975).
9. A. Poitou *et al.* Numerical Prediction of Flow Induced Orientation in Anisotropic Suspensions. Application to Injection Molding of Fibers Reinforced Thermoplastics. *J. Mat. Proc. Tech.*, 32:429–438 (1992).

Measuring the Performance Impact of SP-restricted Programming in Shared-Memory Machines

Arturo González-Escribano[1], Arjan J.C. van Gemund[2], Valentín Cardeñoso-Payo[1], Judith Alonso-López[1], David Martín-García[1], and Alberto Pedrosa-Calvo[1]

[1] Dept. de Informática, Universidad de Valladolid.
E.T.I.T. Campus Miguel Delibes, 47011 - Valladolid, Spain
Phone: +34 983 423270, eMail:arturo@infor.uva.es
[2] Dept. of Information Technology and Systems, Delft University of Technology.
P.O.Box 5031, NL-2600 GA Delft, The Netherlands
Phone: +31 15 2786168, eMail:a.vgemund@et.tudelft.nl

Topic: Languages and Tools
Keywords: programming paradigms, compilation, performance evaluation, graph analysis

Abstract. A number of interesting properties for scheduling and/or cost estimation arise when parallel programming models are used that restrict the topology of the task graph associated to a program to an SP (series-parallel) form. A critical question however, is to what extent the ability to express parallelism is sacrificed when using SP coordination structures only. This paper presents new application parameters that are the key factors to predict this loss of parallelism at both, language modelling and program execution levels, when programming for shared memory architectures. Our results indicate that a wide range of parallel computations can be expressed using a structured coordination model with a loss of parallelism that is small and predictable.

1 Introduction

In high-performance computing currently the only programming methods that are typically used to deliver the huge potential of high-performance parallel machines are methods that rely on the use of either the data-parallel (vector) programming model or simply the native message-passing model. Given current compiler technology, unfortunately, these programming models still expose the high sensitivity of machine performance on programming decisions made by the user. As a result, the user is still confronted with complex optimization issues such as computation vectorization, communication pipelining, and, most notably, code and data partitioning. Consequently, a program, once mapped to a particular target machine is far from portable unless one accepts a high probability of dramatic performance loss.

It is well-known that the use of *structured* parallel programming models of which the associated DAG has series-parallel structure (SP), has a number of advantages [17], in particular with respect to cost estimation [15, 5, 16], scheduling [4, 1], and last but not least, ease of programming itself. Examples of SP programming are clearly found throughout the vector processing domain, as well as in the parallel programming domain, such as in the Bird-Meertens Formalism [16], SCL [3], BSP [19], NestStep [11], LogP [2], SPC [5], OpenMP [13][1]

Despite the obvious advantages of SP programming models, however, a critical question is to what *extent* the ability to express parallelism is sacrificed by restricting parallelism to SP form. Note that expressing a typically non-SP (NSP) DAG corresponding to the original parallel computation in terms of an SP form essentially involves *adding* synchronization arcs in order to obey all existing precedence relations, thus possibly *increasing* the critical path of the DAG. For instance, consider a *macro-pipeline* computation of which the associated NSP DAG is shown at the left of Fig. 1, representing a programming solution according to an explicit synchronization or message-passing programming model. The figure on the right represents an SP programming solution, in which a full barrier is added between every computational step. While for a normally balanced workload situation the SP solution has an execution time similar to the NSP one, in the pathological case where only the black nodes have a delay value of τ while the white ones have 0, the critical path of the SP solution has highly increased. Due the high improbability of such workload situations in a normal computation, SP solutions are generally accepted in vector/parallel processing. They are an easy and understable way to program, useful for data-parallel synchronization structures as well as many other task-parallel applications, and provide portability to any shared or distributed-memory system.

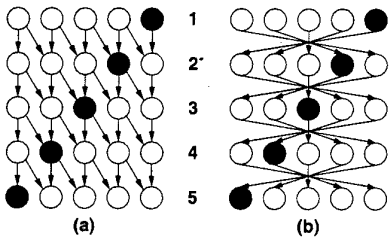


Fig. 1. NSP and SP solutions for a *macro-pipeline* (Inner part)

Let $\gamma \geq 1$ denote the ratio between the critical path of the DAG associated with the original (NSP) algorithm and the DAG of the closest SP program approximation, yet without exploiting any knowledge on individual task work-

[1] Although OpenMP directives are oriented to SP programming, it provides a library for variable blocking that can be used to produce NSP synchronizations.

loads (i.e., only topology is known). Recently, empirical and theoretical results have been presented that show that γ is typically limited to a factor of 2 in practice [5, 6, 8]. In addition, empirical evidence [8] has been presented that for a wide range of parallel applications, especially those within the data parallel (vector) model, γ is strongly determined by simple characteristics of the problem DAG pertaining to topology and workload.

Let Γ denote the ratio between the actual execution times of both solutions when implemented and optimized in a real machine. While $\gamma \geq 1$ at program level, the *actual* performance loss (Γ) as measured at machine level will be positively influenced by the SP programming model as mentioned earlier (See Figure 2). Thus, the initial performance loss when choosing an SP-structured parallel programming model may well be compensated or even outweighed by the potential gains in portability and in performance through superior scheduling quality (in terms of both cost and performance).

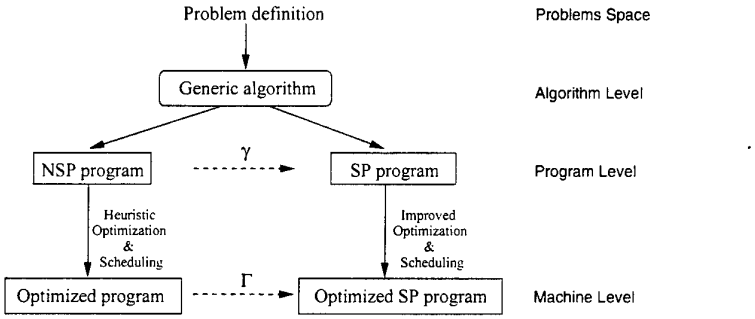


Fig. 2. Measures of performance loss at different levels of abstraction.

In this paper we present the results of a study into the properties of γ and, in particular, Γ. More specifically,

- we extend our earlier study on γ, which was primarily based on synthetic DAGs, with new results for real parallel applications, which confirm the applicability of simple algorithm metrics (such as application scalability, expected number of iterations, synchronization complexity), as prediction parameters for γ.
- we present the results of a study on the relationship between Γ and γ based on the implementation of representative applications on two different shared memory architectures: CC-NUMA (Origin2000) and vector machine (CrayJ90), using different parallel language models (OpenMP or native parallel compiler-directives and message passing. in SP and NSP versions).

This paper is organized as follows. In Section 2 we present a model to measure the loss of parallelism at program abstraction level (γ). Section 3 contains

an overview of the program parameters that determine γ, with some supporting theoretically derived formulae. Section 4 introduces the experiments design to measure the loss of parallelism in algorithms and applications implemented on real machines. The results obtained for representative algorithms when measuring Γ are explained in Section 5.

2 Program level model

The decision to use an NSP or an SP language is taken at the program abstraction level. At this point, the programmer is not concern about the cost of the parallelization mechanics (communications, creation and destruction of tasks, mutual exclusion). He uses a parallel language or library to express the algorithm he designed, taken advantage of the semantics to exploit the parallelism inherent to the problem. Although SP-restricted languages are easier to understand and program, being constrained to SP structures, some of the parallelism could be lost due to added synchronizations. At this level we investigate to what extent one can expect high losses to appear, and what *parameters* related to the algorithm and workload distribution are responsible for this loss.

For our model of programs we use *AoN* DAGs (*Activity on Nodes*), denoted by $G = (V, E)$, to represent the set of tasks (V) and dependencies (E) associated with a program when run with a specific input-data size. Each node represent a task and has an associated *load* or *delay* value representing its execution time. At this level edges represent only dependencies and have no delay value. If required, communication delays should be included in terms of their own specific tasks. *SP* DAGs are a subset of tasks graphs which have only *series* and *parallel* structures, which are constructed by recursively applying Fork/Join and/or series compositions [6].

Let $W : V \rightarrow R$ denote the *workload distribution* of the tasks. For a given graph G and W, we define $C(G)$ (*Critical path* or *Cost*) to be the maximum accumulated delay over all full paths of the graph.

A technique to transform an NSP graph to SP structure without violating original precedence relations is called an *SP-ization* technique. It is a graph transformation ($T : G \rightarrow G'$) where G is an NSP graph and G' has SP form, and all the dependencies expressed in G are directly or transitively expressed in G'. Due to the new dependencies an SP-ization introduces, the critical path may be increased. Let γ denote the relative increment in the critical path produced by a given T, and in general by the best possible T, according to

$$\gamma_T = \frac{C(G')}{C(G)} \quad , \quad \gamma = \min_T(\gamma_T)$$

respectively. Clearly, γ is a function of DAG topology and workload W. For a given W, there exists a transformation T such that γ_T is minimal. However, in a usual programming situation the exact W is either not known or highly data-dependent, and can therefore not be exploited in determining the optimal SP program. Thus, of more interest in our study into γ are the upper bound

$\hat{\gamma}_T$ and the mean value $\bar{\gamma}_T = \mathrm{E}(\gamma_T)$ for *any* possible W. Although there does not exist a generic optimal SP-ization for any G and W, we have conjectured in previous work that typically it holds

$$\forall G, \exists T : \gamma_T \leq 2$$

except for pathological (extremely improbable) values of W [5].

In general, in absence of real W information, a fair assumption is to use i.i.d. (independent, identically distributed) task loads. This model seems especially suitable for huge regular problems and topologies, and generally accurate enough for fine or medium grain parallelism.

Let $G = (V, E)$ be a DAG, and let $t \in V$ denote a task. Then we define a number of DAG properties as listed in table 1.

Task in-degree	$i(t) = \lvert\{(t', t) \in E\}\rvert$
Task out-degree	$o(t) = \lvert\{(t, t') \in E\}\rvert$
Task depth level	$d(t) = 1 + max(d(t')) : (t', t) \in E$
Graph layers	$L(G) = \{l : l \subseteq V; t, t' \in l \Rightarrow d(t) = d(t')\}$
Graph depth	$D(G) = \max_{t \in V}(d(t))$
Graph parallelism	$P(G) = \max_{l \in L(G)}(\lvert l \rvert)$
Synchronization density	$S(G) = \frac{1}{N} \sum_{t \in V} \frac{i(t)+o(t)}{2}$

Table 1. Graph properties

3 Effect of program parameters

Previous empirical studies [6] with random i.i.d. W identified specific structured topologies that present worse γ values than random unstructured topologies with the same number of nodes. Mainly the inner part of Pipelines (see Fig. 1) and Cellular Automata programs. Consequently, we shall chose the above topologies as starting point for our research on the topological factors that are responsible for the loss of parallelism as a result of SP-ization. Other interesting algorithms are also included: LU reductions, Cholesky factorizations, FFT, and several synthetic topologies.

Graphs generated from these algorithms, for different input-data sizes, has been constructed in the program level model. Using random task loads and simple SP-ization techniques [7], an estimation of γ has been derived for them, studying the effect of several graph parameters. Only some of them show to be relevant.

The metrics scalability (or number of processors), and number of iterations are measured with the graph size parameters: *Maximum degree of parallelism* (P) and *Depth level* (D). The effect of P and D, measured in several representative

applications, is shown to be under-logarithmic on γ, and bounded by simple functions [8]. See an example in Fig. 3, with the curves generated for a 2D cellular automata when one parameter is fixed. The contribution of D to the loss of parallelism is limited when $D > P$. This applies for all topologies tested except a special case, which the inner part of a pipeline is an example [8].

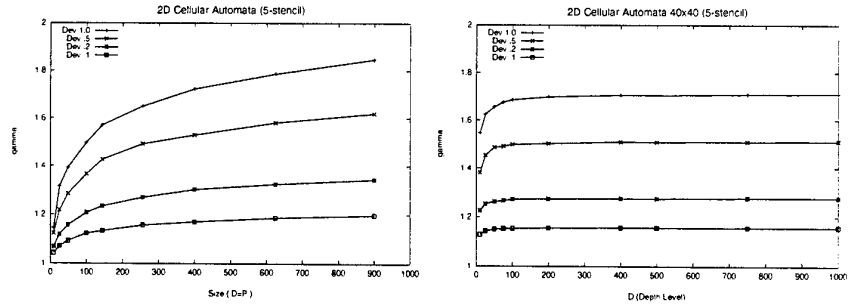


Fig. 3. Effect of P and D - 2D Cellular Automata

Furthermore, the synchronization activity inherent in an algorithm, represented by the *Synchronization Density* parameter (S), limits the γ growing even for small values. In Fig. 4*(a)* is shown how the increment of the number of edges in a synthetic regular topology immediately limits the possible loss of parallelism. In the other hand, small values of S ($S < 2$) indicate the presence of task series. SP-ization techniques that take advantage of these structures lead to great decreases of γ. Fig. 4*(b)* present this effect on a fine grain parallelization of a Cholesky factorization. The values of γ are influenced by the small and variable S parameter, that in this case depends on the input-data size, growing for very small sizes and decreasing afterwards.

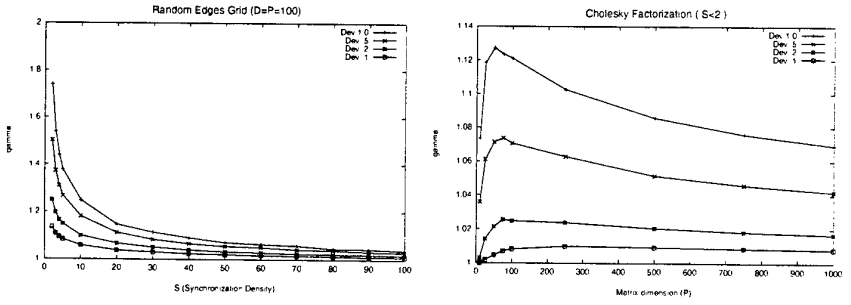


Fig. 4. Effect of P and D - 2D Cellular Automata

Some theoretically derived formulae, using coarse SP aproximations of the original DAGs and order statistics. support these results [18]. An example follows:

Let $P = D$ and let W be modeled by an i.i.d. delay per node according to a Gaussian(μ, σ) distribution. The critical path (C_{SP}) of SP graphs can be derived using a well-known approximation of the cost of a P-node parallel section from order statistics [10], given by

$$C_P = \mu + \sigma\sqrt{2\log(0.4P)}$$

It follows

$$C_{SP} = D(\mu + \sigma\sqrt{2\log(0.4P)})$$

Since generally the cost estimation of NSP stochastic DAGs is analytically intractable, we approximate the NSP DAGs by SP DAGs that capture the main inner features of the original. It appears that the parallelism P' of the SP DAG approximation is directly related to S and P of the original NSP DAG, according to $P' = S + \log(P/2)$ for cellular automata and pipeline topologies. (For 1D cellular automata $S = 3$, for pipeline $S = 2$ [7]). C_{NSP} is then approximated within 10% error [7].

Subsequently applying the SP DAG critical path approximation from order statistics yields

$$C_{NSP} = D(\mu + \sigma\sqrt{2\log(0.4(S + \log(P/2)))})$$

Consequently

$$\bar{\gamma} = \frac{C_{SP}}{C_{NSP}} \approx \frac{D(\mu + \sigma\sqrt{2\log(0.4P)})}{D(\mu + \sigma\sqrt{2\log(0.4(S + \log(P/2)))})}$$

This formula agrees with our experiments within 25% [18]. A coarse, but meaningful simplification of the formula for (typically) large P is given by

$$\bar{\gamma} \approx \frac{\mu + \sigma\sqrt{\log(P)}}{\mu + \sigma\sqrt{\log(S)}}$$

Indeed, the asymptotic influence of P is clearly logarithmic, while the effect of S is exactly inverse, which is in agreement with the results. Also the effect of the workload distribution is in agreement with our measurements (considering the typical case where $P \gg S$).

4 Program execution level

At implementation level a parallel program is compiled and optimized for an specific machine. When executed, it uses costly mechanisms to spawn, synchronize

and communicate tasks. At this level the underlying architecture of the machine becomes important.

The exact cost added to the real execution time is highly dependent on the implementation, for a particular machine, of the parallel mechanisms provided at the language level. The advantages of the architecture must be exploited.

An SP version of a program typically needs to add dependencies, and new delays inherent to the more complex synchronization and communication scheme may increase execution times. The cost of any parallel mechanism is different for each architecture. Nevertheless, the better data partitioning and scheduling techniques, only possible when SP-restricted programming is used, can minimize the communication needings and compensate this effect.

In this study we focus in *shared-memory* architectures. The programming techniques used in these machines are straightforward, and the programmer is not normally facing the data distribution or scheduling details directly.

Our study is focused in two basic shared-memory architectures: CC-NUMA (Origin2000) and Vector-machine (CrayJ90). Both have representative properties for performance evaluation of synchronization techniques. CrayJ90 has a non-hierarchical memory structure. Thus, synchronizations and data access have more predictable delay times. Automatic optimizations deployed by the compiler are mainly oriented to the efficient use of the vector processing units and coarse grain parallelization. Origin2000 is a CC-NUMA machine. The use of memory hierarchy improves performance, while cache-coherence protocols and automatic process migration try to hide machine level details to the programmer. Nevertheless, the efficient use of memory locality is not an easy task even with compiler assistance. Delay times for data access and synchronizations are less stable, specially when full communications or barriers are used across the whole system.

Experiments design: For the experiments presented in this paper we have chosen three of the most representative and easy to program algorithms studied at the previous level:

- 2D Cellular automata (1750x1750 grid, 1750 iterations)
- Pipeline (Inner part, see Fig. 1) (30000 cells vector, 30000 iterations)
- LU reduction (1750x1750 matrix)

The algorithms has been implemented straightforward, mainly from documentation examples and text books, in different programming models:

- MPI, NSP version (Cray and Origin2000)
- MPI with added Barriers, SP version (Cray and Origin2000)
- OpenMP directives, SP version (Origin2000)
- OpenMP variable blocking, NSP version (Origin2000)

For performance comparing, our reference model is the MPI implementation. The second model is generated adding barriers to the original MPI code, to

transform it to SP form. This let us compare the real effect of a direct transformation form NSP code to SP, using the same parallel tool implementation. In most examples two versions of OpenMP are presented. The first one uses simple OpenMP directives, mainly parallel loops, sections and barriers to produce an SP program. In most cases the use of barriers has been intentionally included to compare with the MPI barrier version. The second one, not developed in every example, is a complicated NSP code where every synchronization is implemented as variable blockings. They are used as semaphores in the most possible efficient way.

We avoid compiler aggressive optimization except in the last version. Compiler code manipulation (mainly loop reordering and unrolls) could change the synchronization patterns. Cray versions are compiled with vector optimizations enabled. Specific SP data partitioning and scheduling techniques has not been exploited; we rely on the machine native system.

Measures include the total execution time of the parallel section of each code, as well as the mean and deviation of task times. We consider a task to be a continuous serial computation, from the point after a wait for synchronization is issued (one or more communication receptions, blockings or barriers) to the next one. The experiments were conducted with 2,4,6 and 8 processors. Results, codes and tools used are available [14].

5 Results

In this section we present the results of the machine level experiments. Comparisons with the γ predictions obtained for these problems with the program level model are discussed.

Workload distribution: Parameters from real workloads distributions, obtained when running the programs, are studied. When the problem size is fixed, and the number of processors is incremented, the workload per task is partitioned, leading to smaller tasks. In general, pieces of computation of different size present different mean and deviation on the same machine. Thus, for a fixed problem size, the deviation is variable when the number of processors is changing, and the ratio differs for every machine. Fortunately, the differences are small enough, and measures obtained in the program level model, with the same workload deviation, introduce a minimum error.

Balanced computations: As expected, the task loads of Cellular Automata and Pipeline problems present a minimum deviation in both machines.

$$\begin{array}{ll} \text{Cellular Automata:} & \delta/\mu \in (0.01, 0.06) \\ \text{Pipeline:} & \delta/\mu \in (10^{-5}, 10^{-3}) \end{array}$$

In Fig. 5, Fig. 6 and Fig. 7 is perfectly clear that SP versions (specially the MPI-SP one) has negligible increment on the execution time comparing with

the MPI reference model. The small slope of the Γ curves is perfectly compliant with γ predictions of our program level model, although the values are slightly higher due to the added communication costs, not previously considered.

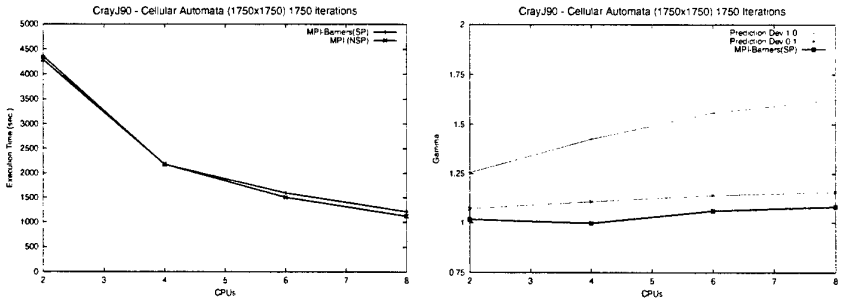


Fig. 5. CrayJ90 - 2D Cellular Automata

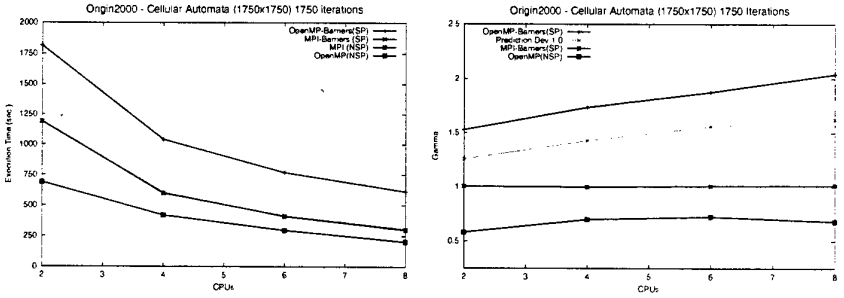


Fig. 6. Origin2000 - 2D Cellular Automata

Each barrier (on each iteration) produce a constant increment on the execution time, apart from the loss of parallelism. It is remarkable that the Origin2000 MPI barrier implementation is almost perfectly efficient, while the OpenMP system adds significantly cost to the execution time [9]. More efficient nested fork/join techniques in the Origin2000 are being researched [12].

As shown in Fig. 6, the OpenMP-SP program generated for the cellular automata problem has shown to be rather inefficient due to poor serial code optimization, that adds a constant time to each iteration. Although another improved version has been programmed that shows much better results (plot moved down), it is interesting to notice the similarities between the high Γ curve and the predictions from the program level model. At the same time, in every figure the OpenMP-NSP version, aggressively optimized, produces lesser task delays

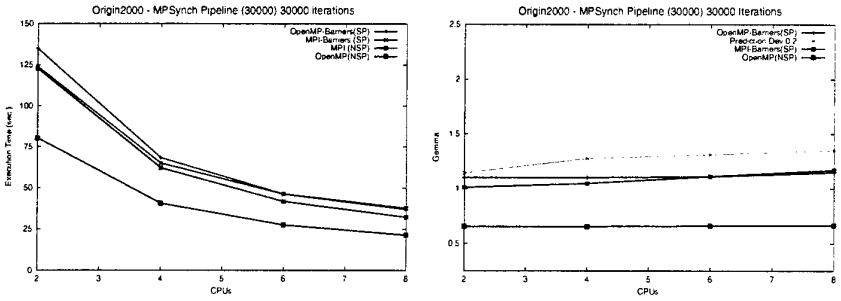


Fig. 7. Origin2000 - Pipeline (inner part)

and therefore lesser final execution time. Nevertheless, in Γ terms the results are comparable with the MPI reference model as the synchronization pattern has not been changed. This reveals that even if Γ could be highly affected by serial programming practices and optimization, the effect of SP-synchronization generated in the program level is preserved.

Unbalanced computations: LU reduction programs typically distribute the rows of the matrix to the processors, synchronizing after rows update. This scheme do not exploit fine grain parallelism inherent to each element update. The amount of work for each task will be too small, and the communication cost will overcome the computation.

Medium-coarse grain parallelism provides higher deviations, as the tasks do not run similar computations. In this case, processing only the triangular part of the matrix, the number of elements that are updated in each row are different, and changing in each iteration. This effect is slightly reduced in the MPI implementation through rows interleaving. The deviations are still high:

$$\text{LU reduction: } \delta/\mu \in (1.5, 1.7)$$

In Fig. 8 the Γ curve obtained in the CrayJ90 is compared with γ predictions for highly unbalanced situations. The Γ high values are not still fully explained. A new increasing effect is produced by the variable cost of a barrier when the number of processors is growing. See Table 2. Adding the predicted effect of γ to the cost of 1750 barriers (one for each iteration) a narrow approximation of the real execution time is obtained. This effect less noticeable in programs with higher task loads like cellular automata (see Fig. 5) and in the Origin2000, where the barrier relative cost is much lesser (see Fig. 9). For Origin2000 it is only detectable for applications with really low task loads, as pipeline (see Fig. 7).

Efficient SP programming: For LU reduction algorithm the OpenMP-NSP version is quite complicate and has not been programmed. Instead, in Fig. 9 we

NPROC	CrayJ90	Origin2000
2	.002	.00005
4	.007	.00049
6	.011	.00053
8	.016	.00083

Table 2. Barrier cost estimation

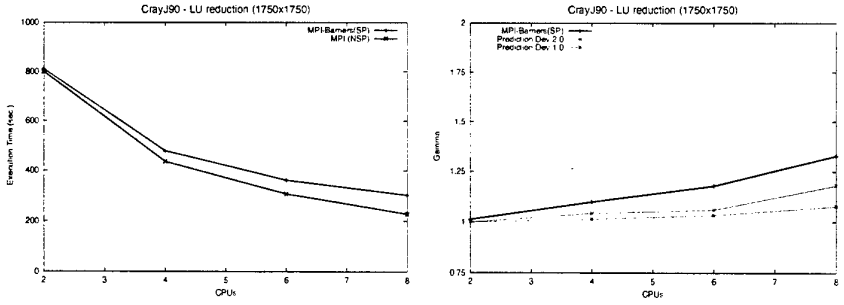


Fig. 8. CrayJ90 - LU reduction

show the results obtained with a manually parallelized OpenMP version, based on SP parallel loops. Automatic optimizations has been applied. An efficient SP version produces good results in Γ terms, with no relative increase when compared with the MPI reference model.

Iterations: Reduced executions of the algorithms has been run with small but growing number of iterations. Results obtained are compliant with the program level predictions. In Fig. 10 is shown how in a cellular automata, for a fixed number of processors, the execution time grows with a given ratio for each programming model. Consequently, Γ is not dependent on the number of iterations D (provided $D > P$ as predicted on the program level).

6 Conclusion

In this paper we present a study on the relationship between the loss of parallelism inherent to SP-restricted programming at program or algorithm design level and the real performance loss as measured at machine execution level. These results point out that the difficult task of mapping a code to a real parallel machine is indeed much easier with a restricted SP language model, while the actual performance loss (Γ) due to the lack of expressiveness is only small and predictably related to the properties of the algorithm (γ).

One should note that SP particular techniques for scheduling or data partitioning have not been exploited. The results presented in this paper can be improved with the use of specific SP programming environments.

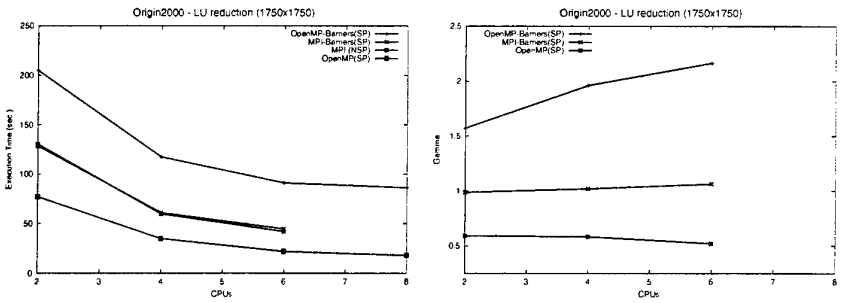


Fig. 9. Origin2000 - LU reduction

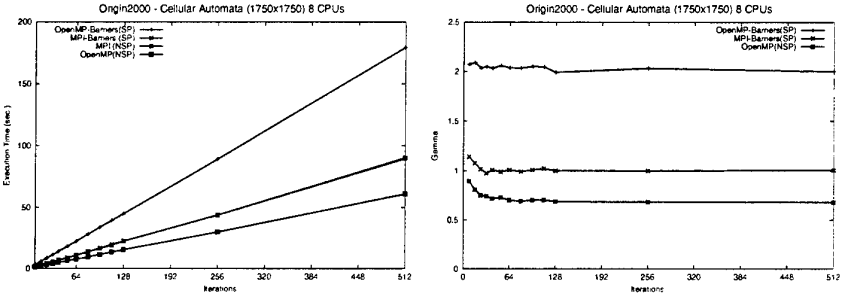


Fig. 10. Origin2000 - 2D Cellular Automata - Iterations

In summary, we show that many parallel computations can be expressed using a structured parallel programming model with limited loss of parallelism, both at the algorithm level and the machine execution level. To the best of our knowledge such a comparative study between NSP and approximated SP implementations of real applications has not been performed before. The significance of the above results is that the optimizability and portability benefits of efficient cost estimation in the design and/or compilation path can indeed outweigh the initial performance sacrifice when choosing a structured programming model.

Future work includes a further study into more irregular, data-dependent or dynamic algorithms and applications, to determine DAG properties to accurately measure both γ and Γ, as well as exploration of the benefits of specific SP programming environments for both, shared-memory and distributed memory architectures.

Acknowledgements

We thank the *Rekencentrum, Rijksuniversiteit Groningen (The Netherlands)*, for let us use their CrayJ90 machine. The Origin2000 machine is a property of the *University of Valladolid. (Spain)*.

References

[1] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. Annual Symposium on FoCS*, pages 356–368, nov 1994.

[2] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Proc. 4th ACM PPoPP*, pages 1–12, San Diego, CA, USA, May 1993.

[3] J. Darlington, Y. Guo, H.W. To, and J. Yang. Functional skeletons for parallel coordination. In *Europar'95*, LNCS, pages 55–69, 1995.

[4] L. Finta, Z. Liu, I. Milis, and E. Bampis. Scheduling UET–UCT series–parallel graphs on two processors. *DIMACS Series in DMTCS*, 162:323–340, aug 1996.

[5] A.J.C. van Gemund. The importance of synchronization structure in parallel program optimization. In *Proc.* 11th *ACM ICS*, pages 164–171, Vienna, 1997.

[6] A. González-Escribano, V. Cardeñoso, and A.J.C. van Gemund. On the loss of parallelism by imposing synchronization structure. In *Proc. 1st Euro-PDS Int'l Conf. on Parallel and Distributed Systems*, pages 251–256, Barcelona, July 1997.

[7] A. González-Escribano, V. Cardeñoso, and A.J.C. van Gemund. Loss of parallelism on highly regular DAG structures as a result of SP-ization. Technical Report 1-60340-44(1999)-04, TU Delft, The Netherlands, July 1999.

[8] A. González-Escribano, A.J.C. van Gemund, V. Cardeñoso-Payo, H-X. Lin, and V. Vaca-Díez. Expressivenes versus optimizability in coordinating parallelism. In *Proc. ParCo'99*, Delft, August 1999.

[9] Paul Graham. OpenMP a parallel programming model for shared memory architecture. Techical-watch report, EPCC, The University of Edinburgh, Mar 1999. available from http://www.epcc.ed.ac.uk/epcc-tec/documents/.

[10] E.J. Gumbel. *Statistical Theory of Extreme Values (Main Results).* chapter 6. pages 56–93. Wiley Publications in Statistics. John Wiley & Sons, 1962.

[11] C.W. Kessler. NestStep: nested parallelism and virtual shared memory for the BSP model. In *Int. Conf. on Parallel and Distrbuted Processing Techniques and Applications (PDPTA'99)*, Las Vegas (USA), June-July 1999.

[12] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. González, and J. Labarta. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors. In *ICS'99*, pages 294–301, Rhodes, Greece, 1999.

[13] OpenMP organization. WWW. on http://www.openmp.org.

[14] PGamma. Measuring the performance impact of SP programming: Resources and tools. WWW. on http://www.infor.uva.es/pnal/arturo/pgamma.

[15] R.A. Sahner and K.S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Trans. on Software Eng.*, 13(10):1105–1114, Oct 1987.

[16] D.B. Skillicorn. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[17] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[18] A. Vaca-Díez. Tools and techniques to assess the loss of parallelism when imposing synchronization structure. Tech.Rep. 1-68340-28(1999)02, TU Delft, Mar 1999.

[19] L.G. Valiant. A bridging model for parallel computation. *C.ACM*, 33(8):103–111, Aug 1990.

A SCOOPP Evaluation on Packing Parallel Objects in Run-time*

João Luís Sobral, Alberto José Proença

Departamento de Informática - Universidade do Minho
4710 - 057 BRAGA – PORTUGAL
{jls, aproenca}@di.uminho.pt

Abstract The SCOOPP (Scalable Object Oriented Parallel Programming) system is an hybrid compile and run-time system. SCOOPP dynamically scales OO applications on a wide range of target platforms, including a novel feature to perform a run-time packing of excess parallel tasks. This communication details the methodology and policies to pack parallel objects into grains and method calls into messages. The SCOOPP evaluation focus on a pipelined parallel algorithm - the Eratosthenes sieve - which may dynamically generate a large number of fine-grained parallel tasks and messages. This case study shows how the parallelism grain-size - both computational and communication - has a strong impact on performance and on the programmer burden. The presented performance results show that the SCOOPP methodology is feasible and the proposed policies achieve efficient portability results across several target platforms.

1 Introduction

Most parallel applications require parallelism granularity decisions: a larger number of fine parallel tasks may help to scale up the parallel application and it may improve the load balancing. However, if parallel tasks are too fine, performance may degrade due to parallelism overheads, both at the computational and communication level.

Static granularity control, performed at compile-time, can be efficiently applied to fine grained tasks [1][2], whose number and behaviour is known at compile-time. HPF [3], HPC++ [4], and Ellie [5] are examples of environments that support static granularity control. However, parallel applications where parallel tasks are dynamically created and whose granularity can not be accurately estimated at compile-time require dynamic granularity control to get an acceptable performance; this also applies when portability is required across several platforms.

Granularity control can lead to better performance when performed by the programmer, but it adds an extra burden on the programmer activity: it requires knowledge of both the architecture and the algorithm behaviour, and it also reduces the code clarity, reusability and portability.

* This work was partially supported by the SETNA-ParComp project (Scalable Environments, Tools and Numerical Algorithms in Parallel Computing), under PRAXIS XXI funding (Ref. 2/2.1/TIT/1557/95).

The SCOOPP system [6] is an hybrid compile and run-time system, that extracts parallelism, supports explicit parallelism and dynamically serialises parallel tasks in excess at run-time, to dynamically scale applications through a wide range of target platforms. This paper evaluates the application of the SCOOPP methodology to dynamically scale a pipelined application - the Eratosthenes sieve - on three different generations of parallel systems: a 7 node Pentium II 350MHz based cluster, running Linux with a threaded PVM on TCP/IP, a 16 node PowerPC 601 66 MHz based Parsytec PowerXplorer and a 56 node T805 30Mhz based Parsytec MultiCluster 3, both running PARIX with proprietary communication primitives, functionally identical to PVM. The cluster nodes are inter-connected through a 1 GBbit Myrinet switch, the PowerXplorer nodes use a 4x4 mesh of 10Mbit Transputer-based connections and the MultiCluster Transputers are interconnected through a 7x8 mesh.

Section 2 presents an overview of the SCOOPP system and its features to dynamically evaluate the parallelism granularity and to remove excess parallelism. Section 3 introduces the Eratosthenes sieve and presents the performance results. Section 4 concludes the paper and presents suggestions for future work.

2 SCOOPP System Overview

SCOOPP is based on an object oriented programming paradigm supporting both active and passive objects. Active objects are called parallel objects in SCOOPP (//obj) and they specify explicit parallelism. These objects model parallel tasks and may be placed at remote processing nodes. They communicate through either asynchronous or synchronous method calls.

Passive objects are supported to take advantage of existing code. These objects are placed in the context of the parallel object that created them, and only copies of them are allowed to move between parallel objects. Method calls on these objects are always synchronous.

Parallelism extraction is performed by transforming selected passive objects into parallel objects (more details in [7]), whereas parallelism serialisation (i.e. grain packing) is performed by transforming parallel objects into passive ones [8].

Granularity control in SCOOPP is accomplished in two steps. At compile-time the compiler and/or the programmer specifies a large number of fine-grained parallel objects. At run-time parallel objects are packed into larger grains - according to the application/target platform behaviour and based on security and performance issues – and method calls are packed into larger messages.

Packing methodologies are concerned on "how" to pack and "which" items to pack; this subject is analysed in section 2.1. These methodologies rely on parameters, which are estimated to control granularity at run-time; these are analysed on section 2.2. Packing policies focus on "when" and "how much" to pack, and they heavily rely on the structure of the application; this subject is analysed in section 2.3.

2.1 Run-time Granularity Control

Conventional approaches for run-time granularity control are based on fork/join parallelism [9][10][11][12][13]. The grain-size can be increased by ignoring the fork and executing tasks sequentially, avoiding spawning a new parallel activity to execute the forked task.

The SCOOPP system dynamically controls granularity by packing several //obj into a single grain and serialising intra-grain operations. Additionally, SCOOPP can reduce inter-grains communication by packing several method calls into a single message.

Packing Parallel Objects. The main goal of object packing is to decrease parallelism overheads by increasing the number of intra-grain operations between remote method calls. Intra-grain method calls - between objects within the same grain - are synchronous and usually performed directly as a normal procedure call; asynchronous inter-grain calls are implemented through standard inter-tasks communication mechanisms.

The SCOOPP run-time system packs objects when the grain-size is too fine and/or when the system load is high. The SCOOPP system takes advantage of the availability of granularity information on existing //obj. When parallel tasks (e.g. //obj) are created at run-time, it uses this information to decide if a newly created //obj should be used to enlarge an existing grain (e.g. locally packed) or originate a new remote grain.

Packing Method Calls. Method call packing in SCOOPP aims to reduce parallelism overheads by packing several method calls into a single message.

The SCOOPP run-time system packs method calls when the grain-size is too fine. On each inter-grains method call, SCOOPP uses granularity information on existing objects to decide if the call generates a new message or if it is packed together with other method calls into a single message.

Packing Parallel Objects and Method Calls. The two types of packing complement each other to increase the grain-size. They differ in two aspects: (i) method calls can not be packed on all applications, since the packing relies on repeated method calls between two grains, and may lead to deadlock when calls are delayed for an arbitrary long time; this delay arises from the need to fulfil the required number of calls per message; (ii) method calls in a message can be more easily unpacked than objects in a grain. Reversing object packing usually requires object migration, whereas packs of method calls can be sent without waiting for the message to be fully packed. In SCOOPP, packs of methods calls are sent either on programmer request or when the source grain calls a different method on the same remote grain.

2.2 Parameters Estimation

To take the decision to pack, two sets of parameters are considered: those that are application independent and those that are application dependent. The former includes the latency of a remote "null-method" call (α) and the inter-node communication bandwidth. The later includes the average overhead of the method parameters passing (ν), the average local method execution time (μ), the method fan-out (ϕ) (e.g., the average number of method calls performed on each object per method execution) and the number of grains per node (γ).

Application independent parameters are statically evaluated by a kernel application, running prior to the application execution; parameters that depend on the application are dynamically evaluated during application execution. The next two subsections present more details of how these two types of parameters are estimated.

Application Independent Parameters. Application independent parameters include the latency of a remote "null-method" call (α) and the inter-node communication bandwidth. Both parameters are defined for a "unloaded" target platform. They are estimated through a simple kernel SCOOPP application that creates several //obj on remote nodes and performs a method call on each object.

The remote method call latency (α) is the time required to activate a method call on a remote //obj. It is estimated as half the time required to call and remotely execute a method that has no parameters and only returns a value.

The inter-node communication bandwidth is estimated by measuring the time required to call a method with an arbitrary large parameters size. It is half of the division of the parameters size by the time required to execute the method call.

On some target platforms, these two parameters depend on the pair source/destination nodes, namely on the interconnection topology. In such cases, the SCOOPP computes the average from the parameters taken between all pairs of nodes. Moreover, these parameters tend to increase when the target platform is highly loaded, due to network congestion and computational load. However, this effect is taken into account on the SCOOPP methodology through the γ parameter (number of grains per node), which is a measure of the load on each node.

These two parameters are statically estimated to reduce congestion penalties at run-time, since they require inter-node communication, which is one of the main sources of parallelism overheads. Their evaluation at run-time, during application execution, may introduce a significant performance penalty.

Application Dependent Parameters. SCOOPP monitors granularity by computing, at run-time, the average overhead of the method parameters passing (ν), the average method execution time (μ) and the average method fan-out (ϕ). SCOOPP computes these parameters, at each object creation, from application data collected during run-time.

The overhead of the method parameters passing (ν) is computed from the inter-node communication bandwidth multiplied by the average method parameter

size. This last one is evaluated by recording the number of method calls and adding the parameter sizes of each method call.

The average method execution time (μ) is evaluated by recording the time required to perform each local method execution. When a method does not perform other calls, this value is just the elapsed time. When a method contains other calls, the measurement is split into the pre-call and after-call phases, and the previous procedure is applied to each phase. Moreover, the time required to perform the pre-call phase is used has a first estimate of the average method execution time, so that the average method execution time data is available for the next method call, even if the first method execution one has not completed yet.

The average method fan-out (ϕ) is measured by a global program analysis through object and method calls statistics. The run-time system marks each //obj with its depth on the object creation tree. The depth of the root object is one and the depth of all other //obj is equal to the depth of its creator plus one. The run-time system maintains a table for the number of call performed on each depth, which is incremented on each local method call. The method fan-out is derived from this table through the overall ratio between consecutive depths.

SCOOPP minimises the run-time impact of the parameters estimation overhead in three ways. First, granularity information is collected at class level, e.g., the ν, μ and ϕ parameters are measured for each class of parallel objects. This approach is clearly less costly than an instance-based approach and more accurate than a global one. Second, when the overhead introduced to access the system clock to measure the average method execution time is high (usually more than 1%) the frequency of information retrieval is reduced; this excludes, however, the application start up phase, since on that phase no information is available. Third, the parameters that are estimated at run-time do not require inter-nodes communication, since the estimation is locally performed and parameters information is only exchanged within requests for remote object creation.

2.3 Packing Policies

Packing policies define "when" and "how much" to pack, e.g. the number of //obj that should be packed in each grain, and the number of method calls to pack on each message. These policies are usually grouped according to the structure of the application: object pipelines, static object trees (e.g. object farming) and dynamic object trees (e.g. work split and merge). The work here presented focus on packing policies for pipelined algorithms and the next section evaluates its application to a case study, the Eratosthenes sieve.

Packing Parallel Objects. The decision "when" to pack is taken based on the average method execution time (μ), the average latency of a remote "null-method" call (α) and the overhead of the method parameters passing (ν). When the average method execution time is excessively short, //obj should be packed, which occurs when the overhead of a remote method call is higher than the average method

execution time, e.g., $(\alpha+\nu)>\mu$. This is the turnover point to pack //obj, where the parallelism overhead becomes longer than the time spent on locally "useful work".

The decision of "how many" //obj to pack into a single grain (e.g., degree of object packing or computation grain-size, C_p) is related to the α, ν and μ parameters as seen before, and also on the method fan-out (ϕ) and on the system computational load, e.g., the number of grains per node (γ). The computation grain-size should be increased when the system presents high parallelism overhead (e.g., high α and ν) and be decreased on high average method execution time. The degree of object packing should also be decreased when fan-out increases, since each method call performs several intra-grain calls, and it can be increased when the number of grains per node is high, to decrease parallelism overheads.

On pipelined applications, packing adjacent //obj makes the number of intra-grain calls equal to the average number of objects in each grain, since the fan-out is close to 1. When C_p //obj are packed together, each remote method call generates C_p method calls, executing on $C_p\mu$ time. Under these conditions, the turnover point to decide when to pack is reached when $C_p=(\alpha+\nu)/\mu$. This expression defines the minimum number of //obj to pack on each grain to overcome the parallelism overheads. To decrease parallelism overheads even more, SCOOPP increases the number of //obj on each grain linearly with γ by using the expression $C_p=\gamma(\alpha+\nu)/\mu$.

Packing Method Calls. The decision "when" to pack method calls follows the same rule as the one applied to pack objects, e.g., when $(\alpha+\nu)>\mu$; this condition reflects that the overhead to place a single remote call is higher than the remote method execution time. In this case, several inter-grains calls should be packed to reduce communication overheads.

The decision "how many" method calls to pack into a single message (e.g., degree of method call packing or communication grain-size, C_m) is computed from the α, ν and μ parameters. Sending a message that packs C_m method calls has a time overhead of $(\alpha+C_m\nu)$ and the time to locally execute this pack is $C_m\mu$. Packing should be performed such that $(\alpha+C_m\nu)<C_m\mu$, e.g., when the overhead to place a remote call is lower than the time to locally execute the pack of method calls. Resolving the equation gives the turnover point $C_m=\alpha/(\mu-\nu)$.

When the average method execution time is close or smaller than the overhead of the parameter passing (e.g., $\mu<=\nu$), method calls should not be packed. However, this rule can be relaxed if both method calls and //obj are packed.

Packing Parallel Objects and Method Calls. SCOOPP can simultaneously pack method calls and //obj. However, when method calls are packed, the application performance may benefit from a less //obj packing degree. In this case, SCOOPP scales down the computation grain-size by using the expression $C_p=\gamma(\alpha+C_m\nu)/(\mu C_m)$.

When the overhead of the parameters passing (ν) is longer than the average method execution time (μ), e.g., $\nu>\mu$, the method calls packing factor should be decreased. In this case, the method call packing is estimated as $C_m=\alpha/\nu$.

To summarise, on pipelined applications, the μ/ν ratio is the key to choose between object and method calls packing. When $\nu<\mu$ the communication packing

degree, in number of method calls per message is $C_m=\alpha/(\mu-\nu)$, and the object packing degree is decreased by the C_m factor, e.g., the number of //obj on each grain is $C_p=\gamma(\alpha+C_m\nu)/(\mu C_m)$. When $\nu>\mu$, the communication packing degree is $C_m=\alpha/\nu$ and the same C_p expression can be used to compute the number of //obj per grain.

3 SCOOPP Evaluation with the Eratosthenes Sieve

The Eratosthenes sieve is an algorithm to compute all prime numbers up to a given maximum. The original algorithm is well known; although several faster algorithms have been proposed [14][15][16], the original one is still the most adequate to illustrate relevant features in parallel algorithms, since a parallel version is intuitively obtained from the original sequential algorithm.

One simple parallel implementation is a pipelined algorithm containing all computed prime numbers, where each element filters its multiples. Numbers are sent to the pipeline on an increasing order. Each number that gets to the end of the pipeline is a prime number and is appended as a new filter. Fig.1 presents the sieve processing flow for the numbers 3, 4 and 5.

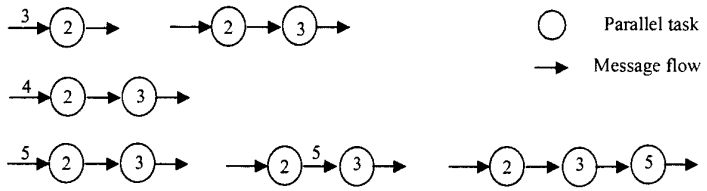


Fig. 1. Parallel sieve of Eratosthenes processing numbers 3, 4 and 5

The Eratosthenes sieve has been chosen to show the relevant features of the SCOOPP since it has a totally predictable behaviour, making it adequate to evaluate the separate impact on the execution time of each parameter and packing approach. Furthermore, it is scalable to large environments, if a large number is selected.

The Eratosthenes sieve has a large parallelism potential since each element of the pipeline (e.g., sieve filter) can be a parallel task (e.g., a parallel object), which originates a large number of fine-grained parallel tasks. It dynamically creates parallel tasks and their number is dependent of the problem size. Table 1 presents the parallelism degree of the sieve of Eratosthenes for several problem sizes.

Table 1. Parallelism degree of the Eratosthenes sieve several problem sizes

Problem size	Number of parallel tasks	Number of messages
100	24	290
1.000	167	14292
10.000	1228	762862
100.000	9591	46224072

On a naive implementation of the sieve, each parallel task has a computation to communication ratio of one integer division operation per message received, which is a too low ratio for the generality of distributed memory machines. A slightly optimised sieve was developed to increase this ratio and decrease the sieve sequential workload, which sends blocks of 10 values between sieve filters on a single method call. Each sieve filter marks the numbers that it filters and a block is merged with another block when it has more 5 values marked. This optimisation decreases the number of messages by a factor close to 10 and increases the computation to communication ratio to a value close to 10 integer divisions per message received.

The next subsection discusses how a programmer based static grain-size adaptation can increase this ratio. A second subsection shows performance results measured using the SCOOPP dynamically grain-size adaptation. Both subsections present performance results for an optimised sieve on a problem size of 100 000 values.

3.1 Programmer Based Grain-size Adaptation

This section shows how a programmer can adapt the grain-size of the sieve to improve performance on several platforms. It presents the impact of the grain-size choices on the number of //tasks and inter-//task messages. Finally, it presents the execution times of the sieve for a number of grain-size choices and analyses the impact of grain-size choices on the tested three platforms.

To adapt the grain-size in the sieve algorithm a parallel programmer may merge sieve filters into a single parallel object and/or pack several parallel objects into a single grain (e.g., a parallel task). Merging filters into a //obj requires some code rewrite, while packing //obj into a grain is less demanding: minor code modifications, mainly to adapt the load distribution policy to perform a block distribution. Merging filters into a //obj removes overheads of intra-grain object creation and method calls, leading to lower execution times (e.g., sequential workload). However, it requires complex code to support dynamic grain-size modifications.

Both approaches adapt the computational grain-size, increasing the average number of operations per received value on each //task (e.g., //task computation to communication ratio) and reducing the overall number of //tasks. On the sieve, this number of operations is directly proportional to the number of filters on each //task and is hereafter referred to as the //task computation granularity, in number of filters per parallel task. However, this increase may not lead to an acceptable performance, namely there may be not enough //tasks and the sieve may generate an excessive number of messages. Packing several method calls into a single message reduces the messages traffic, decreasing the communication overhead. On the optimised sieve under study, the number of values per message is tenfold the number of method calls per message, since each method call sends a block of 10 values, and is hereafter referred to as the inter-//task communication granularity.

Table 2 presents the number of parallel tasks and inter-tasks messages required to compute the prime numbers up to 100 000, for several //task computation and communication granularities. The grain-sizes values were selected to show representatives values of the sieve execution times.

Table 2. Sieve parallelism degree for several computation and communication granularities

Task computation granularity (filters per //task)	// tasks	Inter-tasks communication granularity (values per message)				
		10	50	100	500	1000
1	9591	4 894 536	1 005 717	518 063	118 860	67 100
6	1599	845 518	174 272	89 192	19 569	10 856
25	384	236 692	45 144	24 013	6 364	3 354
100	96	72 750	16 175	8 318	1 873	889
400	24	21 406	4 678	2 395	526	282
1600	6	8 572	1 802	915	194	102
6400	2	5 480	1 110	558	115	59
9591	1	0	0	0	0	0
	// tasks	Inter-tasks messages				

Fig.2 presents the sieve execution times as a function of both the computation and communication granularities. These figures present the execution times on 4 and 7 cluster nodes, on 4 and 16 PowerXplorer nodes and on 14 and 56 MultiCluster nodes. On these experiments the measured values were obtained by using one sieve filter per //obj and grain packing was performed by packing several //obj into a single //task. The MultiCluster can not run sieves with grains smaller than 3 sieve filters, due to memory space limitations. All graphs are scaled to the sieve execution time on a single node.

On all these targets platforms the computation granularity has a strong impact on the sieve performance: when the computation grain-size is too fine or too large the performance penalties are considerably heavy. Too fine grains can lead to a large number of //tasks and the associated overhead costs; too large grains may not use all the available processing nodes.

Communication grains also have an impact on the overall performance: on smaller systems, fine grains (short messages) introduce a penalty, since they generate an excessive number of messages between pairs of nodes; on large systems, shorter and more frequent messages favour load balancing and reduce start-up times.

These results show how relevant is the right choice for both the computation and communication grain-size. However, they also show how time consuming a programmer based approach can be due to the dynamic nature of the parallel tasks of the sieve; it requires long experimental work (to test a wide range of computation and communication grain-sizes) and/or a deep analysis of both the algorithm and target platform features.

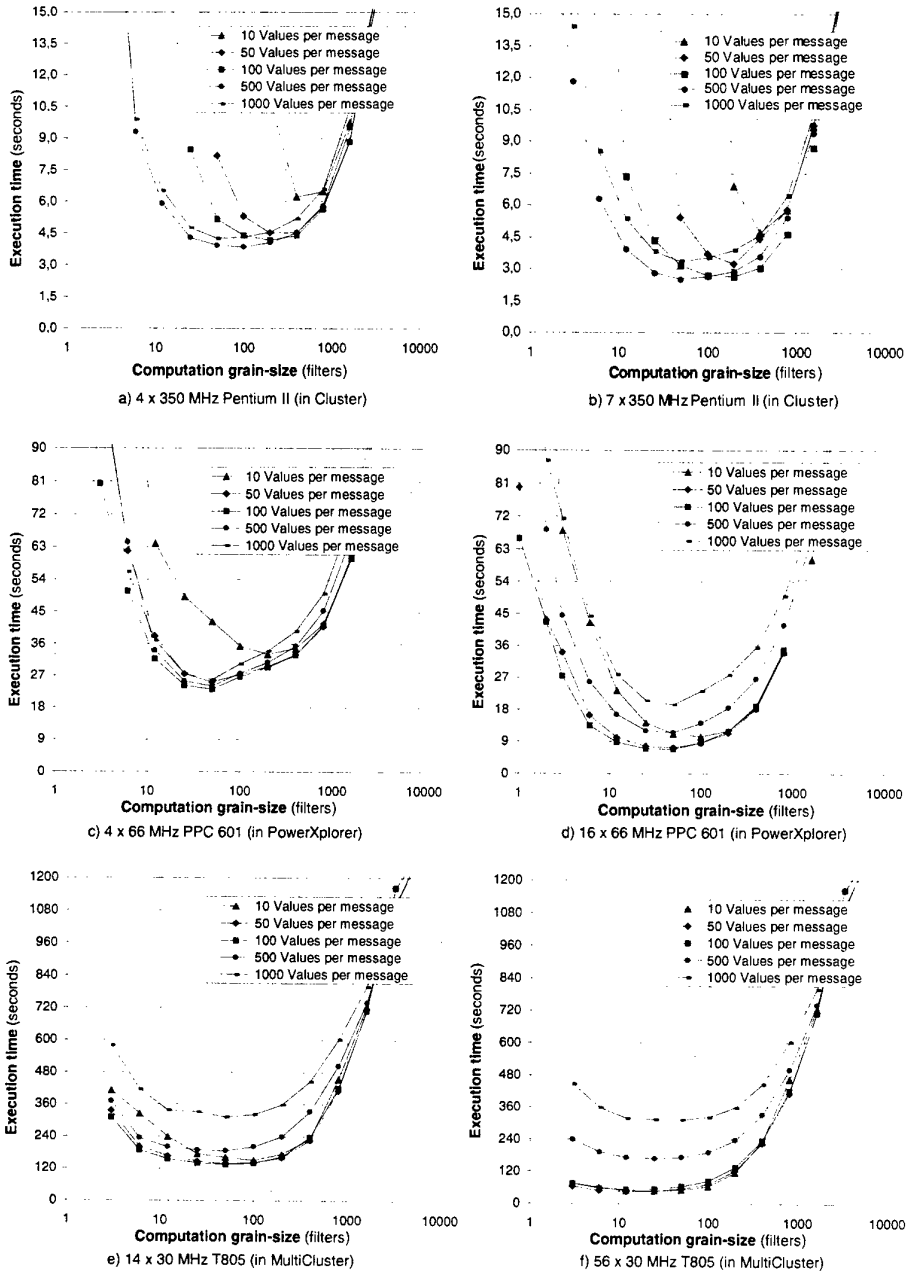


Fig. 2. Sieve execution times for a programmer based grain-size adaptation

3.2 Dynamic Grain-size Adaptation

One of the main goals of SCOOPP is dynamic scalability through grain packing. To tests de effectiveness of the SCOOPP granularity control, an evaluation of the ability to dynamically pack sieves filters was performed. This evaluation compares the lowest execution times – experimentally measured in the previous section - with the execution times obtained by running the sieve on several target platforms, without any code change and relying on the granularity control mechanisms in the SCOOPP run-time system.

Table 3 summarises the measures above mentioned. On the SCOOPP strategy, the computation and communication grain-sizes were obtained by computing the number of //obj on each //tasks and the number of method calls on each message, according the expressions on section 2.3. A circular load balancing strategy spreads grains through the nodes and can place several grains per node.

The P column shows the number of processors used for each test. For both the programmer based and SCOOPP grain-size adaptation several parameters are given: T is the execution time in seconds; Sp is the speedup obtained comparing with the same sieve on a single node; γ is the number of grains placed on each node; C_p is the degree of the computation packing, in number of //obj (filters) per //task (e.g., the computation grain-size) and C_m is the degree of the communication packing, in values per message (e.g., the communication grain-size). C_m is tenfold the number of method calls per message, since each method call sends a block of 10 values. On the SCOOPP methodology C_p and C_m are mean values, since they are computed dynamically and change during run-time.

The SCOOPP methodology results also include 3 columns with the estimated parameters, in microseconds: the remote method call latency (α), the overhead of the method parameters passing (ν) and the average method execution time (μ). The latter two parameters are also mean values.

Table 3. Comparing sieve execution times: programmer based and SCOOPP

		Programmer based					SCOOPP							
	P	T	Sp	γ	C_p	C_m	T	Sp	γ	C_p	C_m	α	ν	μ
Cluster	4	3.86	3.8	24	100	500	3.96	3.7	28	86	560	500	10	5
	7	2.52	5.9	27	50	500	2.66	5.6	21	65	560			
PowerXplorer	4	23.2	3.9	48	50	100	28.0	3.2	19	126	50	300	72	18
	16	6.9	13.0	12	50	100	8.0	11.3	12	50	50			
MultiCluster	14	135.6	9.6	14	50	100	162.3	8.0	21	34	20	530	82	440
	56	44.5	29.2	14	12	50	44.8	29.0	11	16	20			

These results show the effectiveness of the SCOOPP methodology to scale the sieve application on several target platforms. The methodology was able to dynamically increase grain-sizes to obtain speedups of the same order of magnitude as a programmer-based approach. Moreover, execution times obtained through the SCOOPP methodology are often in a 20% range of the optimal values, showing that this methodology successively removes most of the parallelism overheads. The remaining overhead is usually due to a choice of a too large or too small number of

grains. However, removing this overhead requires the knowledge of the full number of //task or some guessing through experiments, as the ones performed on the previous section. These alternatives increase development costs and are not feasible on applications where the number of //tasks is strongly dependent on input data.

When computation and communication grain-sizes are controlled through packing (both object and method calls packing) the total number of created objects and method calls remains the same. To reduce this sequential workload - due to the object oriented paradigm – the programmer can "pack" by merging several //obj into a single //obj (e.g., pack several filters into a single //obj) and by grouping blocks of values on a single method call. Fig.3a and 3b show the impact of merging several filters into a single //obj and increasing the block size on method calls. The graphs show execution times on a single cluster node and the ideal execution time on 4 cluster nodes, for several computation and communication grain-sizes.

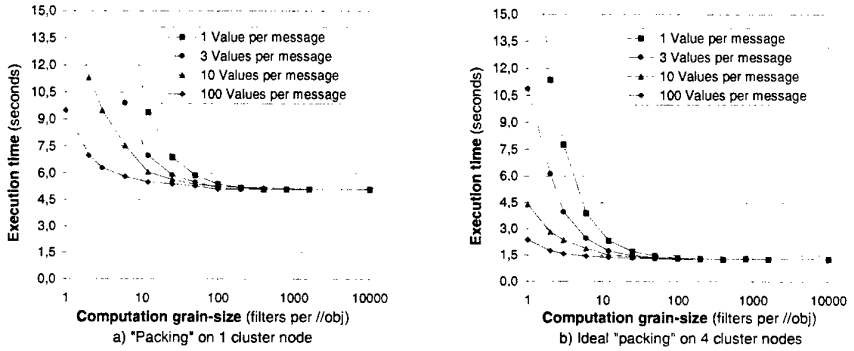


Fig. 3. Execution times for partially optimised sieves through method call and object merging

When these optimisation approaches are followed to supply SCOOPP with pre-optimised parallel versions, SCOOPP is also able to improve the overall performance. Fig.4a presents the times obtained on programmer partially optimised sieves, with communication grain-sizes of 10 and 100 values per message; Fig.4b shows their behaviour on the SCOOPP system.

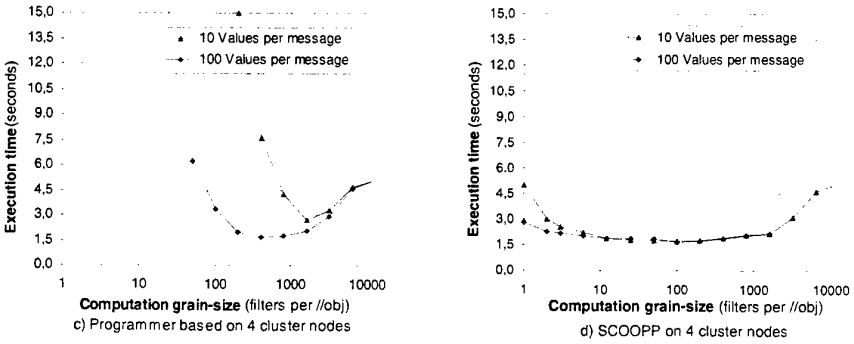


Fig. 4. Programmer based and SCOOPP implementation on partially optimised sieves

These execution times show that the SCOOPP values are very close to the "ideal" ones in Fig.3b. These results reinforce the effectiveness of the SCOOPP methodology, showing that SCOOPP can efficiently scale pipelined applications, even when these are previous and partially optimised by the programmer.

Conclusion

The commercial success of massively parallel systems was slowed down mainly due to the lack of adequate tools to support automatic mapping of the applications into distinct target platforms, without significative loss of efficiency. The overhead on programmers was too high and the available tools were inefficient. SCOOPP attempts to overcome these limitations: it provides dynamic and efficient scalability of object oriented parallel applications across several target platforms, packing grains and messages, without any code modification.

The presented results show the effectiveness of the SCOOPP methodology when applied to pipelined applications on several target platforms. The methodology is able to dynamically increase grain-sizes and to obtain speedups of the same order of magnitude as a programmer-based approach. Moreover, execution times obtained through the SCOOPP methodology are often in a 20% range of the optimal values, showing that this methodology successively removes most of the parallelism overheads.

Programmer based grain-size adaptation is not a competitive alternative to SCOOPP, it requires a wide range of tests on each target platform and each test is highly time consuming (as presented in Fig.2).

The performance penalties imposed by SCOOPP have a low impact on application execution time, and they are mainly due to the run-time requirements to estimate the application dependent parameters to adapt the computation and communication grain-sizes. A static adaptation can provide the correct grain-size at the beginning of the running, but a dynamic strategy requires some time to evaluate the application features and to react accordingly.

Dynamic scalability of the parallel code version largely overcomes this small performance cost. It is the most promising approach to scale applications where task granularity is strongly dependent on input data. When compile time estimates of task granularity are not accurate, it may decrease the cost of the parallel code development and improve the code reutilization on multiple target platforms.

Current work includes development of packing policies for static and dynamic object trees, and applied to less controlled application environments (such as computer vision applications).

References

[1] Kruatrachue, B., Lewis, T.: Grain Size Determination for Parallel Processing, IEEE Software, Vol. 5(1), January (1988)

[2] Gresoulis, A., Yang, T.: On the Granularity and Clustering of Direct Acyclic Graphs, IEEE Transactions on Parallel and Distributed Systems, Vol. 4(6), June (1993)

[3] High Performance Fortran Forum: HPF language specification, Technical Report CPRPC-TR92225, Center for Research on Parallel Computation, Rice University, Tex., (1993)

[4] Beckman, P., Gannon, D., Johnson, E.: HPC++ and the HPC++ Lib. Toolkit, White Paper, www.extreme.indiana.edu/hpc++, (1997)

[5] Andersen, A.: A General, Fine-Grained, Machine Independent, Object-Oriented Language, ACM SIGPLAN Notices, Vol. 29(5), May (1994)

[6] Sobral, J., Proença, A.: Dynamic Grain-Size Adaptation on Object-Oriented Parallel Programming - The SCOOPP Approach, Proceedings of the 2nd Merged IPPS/SPDP 1999, Puerto Rico, April (1999)

[7] Sobral, J., Proença, A.: ParC++: A Simple Extension of C++ to Parallel Systems, Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Applications (PDP'98), Madrid, Spain, January (1998)

[8] Sobral, J., Proença, A.: A Run-time System for Dynamic Grain Packing, Proceedings of the 5th International EuroPar Conference (Euro-Par'99), Toulouse, France, September (1999)

[9] Mohr, E., Kranz, A., Halstead, R.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, IEEE Transactions on Parallel and Distributed Processing, Vol. 2(3), July (1991)

[10] Goldstien, S., Schauser, K., Culler, D: Lazy Threads: Implementing a Fast Parallel Call, Journal of Parallel and Distributed Computing, Vol. 37(1), August (1996)

[11] Karamcheti, V., Plevyak, J., Chien, A.: Runtime Mechanisms for Efficient Dynamic Multithreading, Journal of Parallel and Distributed Computing, Vol. 37(1), August (1996)

[12] Taura, K., Yonezawa, A.: Fine-Grained Multithreading with Minimal Compiler Support – A Cost Effective Approach to Implementing Efficient Multithreading Languages, Proceedings of the ACM SIGPLAN Conference on Programming Language Design & Implementation (CPLDI'97), Las Vegas, July (1997)

[13] Lopez, P., Hermenegildo, M., Debray, S.: A Methodology for Granularity Based Control of Parallelism in Logic Programs, Journal of Symbolic Computation, Vol. 22, (1998)

[14] Pritchard, P.: Linear Prime-Number Sieves: A Family Tree, Science of Computer Programming, Vol. 9, (1987)

[15] Xuedong, L.: A Practical Sieve Algorithm Finding Prime Numbers, Communications of the ACM, Vol. 32(3), (1989)

[16] Dunten, B., Jones, J., Sorenson, J.: A Space-Efficient Fast Prime Number Sieve, Information Processing Letters, Vol. 59, (1996)

The Distributed Engineering Framework TENT

Thomas Breitfeld[2], Tomas Forkert[1], Hans-Peter Kersken[1]
Andreas Schreiber[1], Martin Strietzel[1], Klaus Wolf[2]

[1] DLR, Simulation- and Softwaretechnology, 51170 Cologne, Germany,
<first name>.<last name>@dlr.de, http://www.sistec.dlr.de,
Phone: +49-2203/601-2002, Fax: +49-2203/601-3070
[2] GMD, Institute for Algorithms and Scientific Computing, Schloss Birlinghoven,
53754 Sankt Augustin, Germany
<first name>.<last name>@gmd.de, http://www.gmd.de/scai,
Phone: +49-2241/14-2557, Fax: +49-2241/14-2181

Abstract. The paper describes TENT, a component-based framework for the integration of technical applications. TENT allows the engineer to design, automate, control, and steer technical workflows interactively. The applications are therefore encapsulated in order to build components which conform to the TENT component architecture. The engineer can combine the components to workflows in a graphical user interface. The framework manages and controls a distributed workflow on arbitrary computing resources within the network. Due to the utilization of CORBA, TENT supports all state-of-the-art programming languages, operating systems, and hardware architectures. It is designed to deal with parallel and sequential programming paradigms, as well as with massive data exchange. TENT is used for workflow integration in several projects, for CFD workflows in turbine engine and aircraft design, in the modeling of combustion chambers, and for virtual automobile prototyping.

1 Introduction

The design goal of TENT is the integration of all tools which belong to the typical workflows in a computer aided engineering (CAE) environment. The engineer should be enabled to configure, steer, and control interactively his personal process chain. The workflow components can run on arbitrary computing resources within the network. We achieved this goals by designing and implementing a component-based framework for the integration of technical applications.

TENT is used as integration platform in several projects. In the SUPEA project we developed a simulation environment for the analysis of turbocomponents in gas turbines. An integrative environment for the development of virtual automobile prototypes is under construction at the AUTOBENCH project at GMD. In the AMANDA project, TENT is used for the integration of multidisciplinary tools for the simulation of the static aeroelastic of a complete aircraft. Finally, at the BKM project several physical models for the simulation of combustion chambers are coupled with the central simulation code using TENT.

From the requirements of these projects we can derive the main requirements of the integration system:

- Easy integration of existing CFD-codes, finite element-codes, pre- and post-processors;
- Integration of sequential and parallel codes (MPI, PVM, or HPF);
- Efficient data exchange between the tools;
- Integration of tightly coupled codes;
- Interactive control and design of the workflows;
- Interactive control of the simulation.

1.1 Related Work

The development of frameworks, integration, or problem solving environments (PSE) is focussed in many scientific projects. The interactive control of simulations in combination with virtual reality environments is demonstrated within the collaborative visualization and simulation environment COVISE [8]. Actual projects concentrate on the utilization of CORBA [4] for the organization of distributed systems. PARDIS [5] introduced an improved data transfer mechanism for parallel applications by defining new parallel data objects in CORBA. A very similar approach is addressed in the ParCo (parallel CORBA) project [11]. The integration framework TENT is a CORBA-based environment, which defines its own component architecture for a distributed environment and a new data exchange interface for efficient parallel communication on top of CORBA.

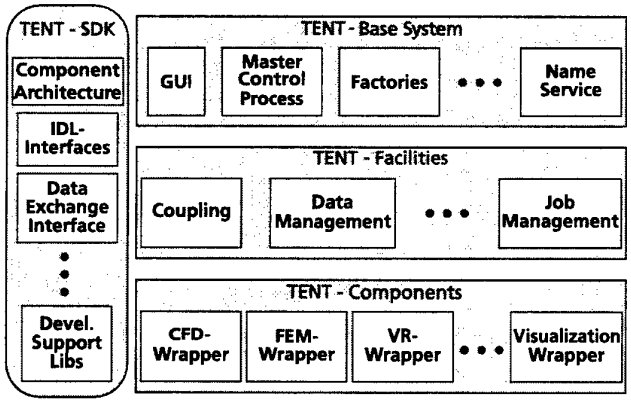


Fig. 1. Packages of the Integration System TENT

2 Base concepts of the framework

The TENT framework consists of four different packages. This structure is displayed in figure 1. The Software delopment kit summarizes all interface definitions and libraries for the software development with TENT. The base system

includes all basic services needed to run a system integrated with TENT. The facilities are a collection of high-level services and the components consist of wrappers and special services for the integration of applications as TENT - Components.

2.1 The Software Development Kit

TENT defines a component architecture and an application component interface on top of the CORBA object model. The TENT component architecture is inspired by the JavaBeans specification [3] and the Data Interchange and Synergistic Collateral Usage Study framework (DISCUS) [15]. The interface hierarchy is shown in figure 2. The data exchange interface as part of the SDK allows the parallel data exchange between two TENT - components. It can invoke a data converter, which automatically converts CFD-data between the data formats supported by the integrated tools.

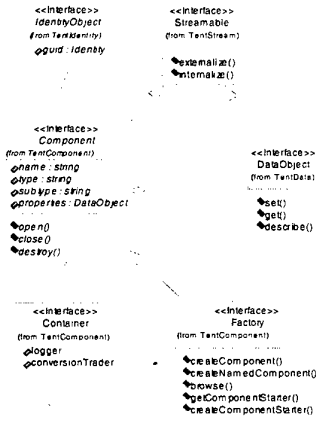


Fig. 2. TENT Framework Interface Hierarchy (UML notation)

2.2 TENT - Base System

The TENT system components form an engineering environment out of a bunch of stand alone tools. All system components are implemented in Java. The Master Control Process (MCP) is the main entity within TENT. It realizes and controls system tasks such as process chain management, construction and destruction of components, or monitoring the state of the system.

The factories run on every machine in the TENT framework. They start, control, and finish the applications on their machine. The name service is the standard CORBA service. The relations between the services are shown in figure 3.

2.3 TENT Facilities

In order to support high-level workflows the system offers more sophisticated facilities. For coupling multi-disciplinary simulation codes, working on numerical grids, TENT will include a coupling server. This server offers the full functionality of the coupling library MpCCI[1] (former CoColib) [1]. MpCCI coordinates the exchange of boundary informations between simulation applications working on neighbouring grids. We are working on a datamanagement service, which stores and organizes the simulation data of typical CFD or finite element simulations. For the virtual automobile prototypes a dataserver is already in production, that can hold the data sets for allowing a feedback-loop between a virtual reality environment, a crash simulator and the appropriate preprocessing tools.

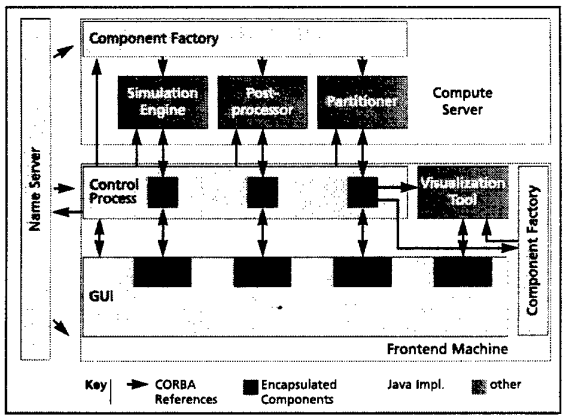


Fig. 3. Architectural Overview of TENT

2.4 TENT Components

The applications must be encapsulated by wrappers to access their functionality in TENT by CORBA calls. Due to the level of the accessibility of the sources, the wrapper can be tightly coupled with the application, e.g. linked together

[1] MpCCI is a trademark owned by GMD.

with it, or must be a stand alone tool, that starts the application via system services. Depending on the wrapping mechanism the communication between the wrapper and the application is implemented by, e.g., direct memory access, IPC mechanisms, or file exchange.

3 Applications

In several projects the TENT framework is used for the integration of engineering workflows. Originally TENT was developed in the SUPEA project for integrating simulation codes to build a numerical testbed for gas turbines.

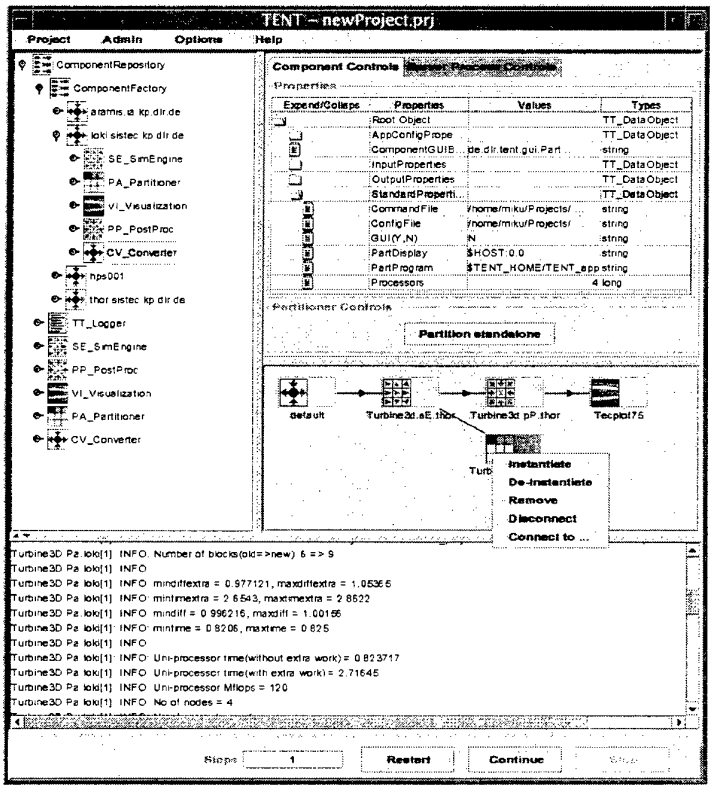


Fig. 4. TENT Control GUI with (from left to right, beginning in the upper left corner) Component Repository, Property Panel, Component Controls, Wire Panel, and Logger Information

In this project several preprocessors, simulation tools, postprocessors, and visualization tools are integrated by TENT, and can be combined to form workflows. A typical SUPEA workflow is designed in the GUI snapshot at figure 4. The workflow is displayed at the Wire Panel. Each icon shows one application in the workflow. The wires between the icons describe the control flow. The most left icon is the master control process which starts and control the flow. It starts the simulation engine, which requests the decomposed grid data from the preprocessor. After a user defined number of iteration steps the simulation engine sends the actual physical data set to a postprocessor, which finally sends a visualizable data file to the visualizer at the end of the workflow. After the first run the workflow repeats without calling the preprocessor again as often as the user choose in the bottom line of the GUI.
The workflow can be freely designed by dragging the applications displayed in the Component Repository and dropping them to the Wire Panel. Here the control flow is defined by wiring the icons interactively. Clicking on the icons in the Wire Panel shows their editable properties in the Property Panel in the upper right corner of the GUI.

In the AMANDA project TENT will be used for the integration of several simulation, pre-, postprocessing, control, and visualization tools. The aim is to form an environment for the multi-disciplinary aeroelastic simulation of an aircraft in flight manœuvres. Therefore the CFD-simulation of the surrounding flow must be coupled with a finite element analysis of the wing structure. Logic components will be included in TENT to allow dynamic workflows. In the next paragraph the aspect of software integration at the AMANDA project is described in more detail.

The virtual automobile prototype is the aim of the project AUTOBENCH [13]. In this project TENT is used to allow the interactive control of a crash simulation from within a virtual reality environment.

3.1 The AMANDA–Workflows

Airplane Design For the simulation of a trimmed, freely flying, elastic airplane the following programs have to be integrated into TENT:

- a CFD code, TAU [12] and FLOWer [6],
- a structural mechanics code (NASTRAN[10]) and a multi-body program (SIMPACK [7]), to calculate the deformation of the structure,
- a coupling module, to control the coupled fluid-structure computation,
- a grid deformation module, to calculate a new grid for the CFD simulation,
- a flight mechanics/controls module (build using an Object-Oriented modelling approach [9]), to set the aerodynamic control surface positions,
- visualization tools.

Figure 5 shows a complete AMANDA airplane design workflow to be integrated in TENT. In this case TAU is choosen for the CFD simulation. The process chain is hierarchically structured. The lowest level contains the parallel CFD

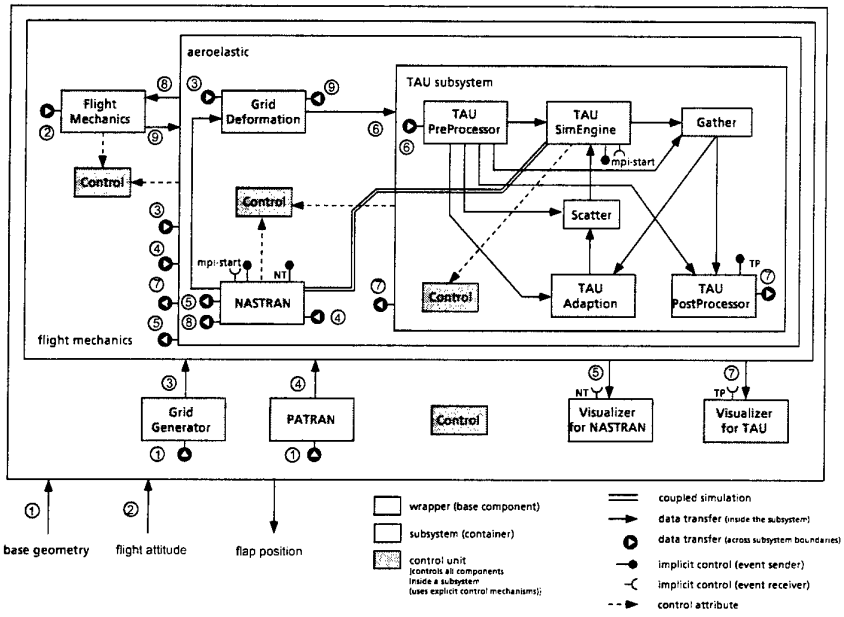


Fig. 5. Process chain for coupled CFD/structural mechanics/flight control system.

solver only. This CFD-subsystem consists of the flow solver itself and auxiliary programs to handle mesh adaptation. The next level, the aeroelastic-subsystem, comprises the CFD solver, the structural mechanics or multi-body code, and the grid deformation module. For the solution of the coupled non-linear aeroelastic equations a staggered algorithm is implemented in the control process [2]. The highest level consists of the flight mechanics module coupled to the aeroelastic-subsystem. Each single code is additionally accompanied by its own visualization tool. The computation of a stable flight state typically proceeds as follows: Starting by calculating the flow around the airplane, the pressure forces on the wings and the nacelle are derived. Theses forces are transfered to the structural mechanics code and interpolated to the structural mechanics grid using the MpCCI library. This code calculates the deformation of the structure which in turn influences the flow field around the airplane. At a final stage it is planned to extend the system and feed the calculated state into a flight mechanics/controls module to set the control surfaces accordingly and to obtain a stable flight position. This changes the geometry of the wings and requires therefore the recalculation of the flow field and the deformation.

Turbine Design A new aspect in virtual turbine design is the simulation of flow inside the turbine in consideration of the heat load on the blades and the

cooling. The numerical modeling of the position and size of cooling channels and holes in the blades are essential for the layout of an air-cooled turbine.

The CFD code TRACE [14], a 3D-Navier-Stokes-Solver for the simulation of steady and unsteady multistage turbomachinery applications, and a heat conduction problem solver (NASTRAN) are coupled to simulate the airflow through the turbine together with the temperature distribution inside the blades. For the coupling of the flow simulation and the heat conduction a stable coupling algorithm as been developed where TRACE delivers the temperatures of the air surrounding the blade and the heat transfer coefficients as boundary conditions to NASTRAN which in turn calculates the temperature distribution inside the blade. The temperature at the surface of the blade is returned to TRACE as boundary condition for the airflow.

4 Conclusions

The integration framework TENT allows a high-level integration of engineering applications and supporting tools in order to form static as well as dynamically changeable workflows. TENT sets up, controls, and steers the workflow on a distributed computing environment. It allows the integration of parallel, or sequential code in most common programming languages and on most common operating systems. CORBA is used for communication and distributed method invocation. Additionally a fast parallel data exchange interface is available. At the end of 1999 TENT will be available as a freely distributed software via the Internet. TENT is already used for integration in several scientific engineering projects with an industrial background.

Acknowledgments

The simulation codes in the AMANDA project are provided by Johann Bals and Wolf Krüger (SIMPACK), Armin Beckert (coupling of TAU and NASTRAN), Thomas Gerholt (TAU), Ralf Heinrich (FLOWer), and Edmund Kügeler (TRACE), all working at the DLR.
The MpCCI library is developed by Regine Ahrem, Peter Post, and Klaus Wolf (all at GMD).
The simulation codes in the SUPEA project are provided by Frank Eulitz (TRACE, at DLR), Harald Schütz (TRUST, at DLR), Georg Bader (at TU Cottbus), and Ernst von Lavante (at Uni/GH Essen).

References

1. Regine Ahrem, Peter Post, and Klaus Wolf. A communication library to couple simulation codes on distributed systems for multiphysics computations. In *Proceedings of ParCo99*, Delft, August 1999.

2. Armin Beckert. Ein Beitrag zur Strömung-Struktur-Kopplung für die Berechnung des aeroelastischen Gleichgewichtszustandes. ISRN DLR-FB 97-42, DLR, Göttingen, 1997.
3. Graham Hamilton (ed.). *JavaBeans API Specification.* Sun Microsystems Inc., July 1997.
4. Object Management Group. *The Common Object Request Broker: Architecture and Specification.* OMG Specification, 2.2 edition, February 1998.
5. Katarzyna Keahey and Dennis Gannon. PARDIS: a parallel approach to CORBA. In *Proceedings of IEEE 6th International Symposium on High Performance Distributed Computing*, Portland, OR., August 1997.
6. Norbert Kroll. The National CFD Project Megaflow - status report. In H. Körner and R. Hilbig, editors, *Notes on numerical fluid mechanics*, volume 60. Braunschweig, Wiesbaden, 1999.
7. Wolf Krüger and Wilhelm Kortüm. Multibody simulation in the integrated design of semi-active landing gears. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference.* Boston, 1998.
8. Ulrich Lang and Dirk Rantzau. A scalable virtual environment for large scale scientific data analysis. In *Future Generation Computer Systems*, volume 14, pages 215–222. Elsevier Science, 1998.
9. D. Moormann, P. J. Mosterman, and G. Looye. Object-oriented computational model building of aircraft flight dynamics and systems. *Aerospace Science and Technology*, 3:115–126, April 1999.
10. Nastran. http://www.macsch.com.
11. Thierry Priol and Christoph René. Cobra: A CORBA-compliant programming environment for high-performance computing. In David Protchard and Jeff Reeve, editors, *Euro-Par'98 Parallel Processing*, number 1470 in Lecture Notes in Computer Science, Southampton, UK, September 1998. Springer.
12. Dieter Schwamborn, Thomas Gerholt, and Roland Kessler. DLR TAU code. In *Procedings of the ODAS-Symposium*, June 1999.
13. Clemens-August Thole, Sven Kolibal, and Klaus Wolf. AUTOBENCH: Environment for the Development of Virtual Automotive Prototypes. In *Proceedings of 2nd CME-Congress (to appear)*, Bremen, September 1999 1999.
14. Dirk Thorsten Vogel and Edmund Kügler. The generation of artificial counter rotating vortices and the application for fan-shaped film-cooling holes. In *Proceedings of the 14th ISABE*, ISABE-Paper 99-7144, 1999.
15. Ron Zahavi and Thomas J. Mowbray. *The essential CORBA: Systems Integration Using Distributed Objects.* John Wiley & Sons, New York, August 1997.

Suboptimal Communication Schedule for GEN_BLOCK Redistribution†

Hyun-Gyoo Yook and Myong-Soon Park

Dept. of Computer Science & Engineering, Korea University,
Sungbuk-ku, Seoul 136-701, Korea
{hyun, myongsp}@iLab.korea.ac.kr

Abstract. This article is devoted to the redistribution of one-dimensional arrays that are distributed in a GEN_BLOCK fashion over a processor grid. While GEN_BLOCK redistribution is essential for load balancing, prior research about redistribution has been focused on block-cyclic redistribution. The proposed scheduling algorithm exploits a spatial locality in message passing from a seemingly irregular array redistribution. The algorithm attempts to obtain near optimal scheduling by trying to minimize communication step size and the number of steps. According to experiments on CRAY T3E and IBM SP2, the algorithm shows good performance in typical distributed memory machines.

1. Introduction

The data parallel programming model has become a widely accepted paradigm for programming distributed memory multicomputer. Appropriate data distribution is critical for efficient execution of a data parallel program on a distributed memory multicomputer. Data distribution deals with how data arrays should be distributed to each processor. The array distribution patterns supported in High Performance Fortran (HPF) are classified into two categories — basic and extensible. Basic distributions like BLOCK, CYCLIC, and CYCLIC(n) are useful for an application that shows regular data access patterns. Extensible distributions such as GEN_BLOCK and INDIRECT are provided for load balancing or irregular problems [1].

The array redistribution problem has recently received considerable attention. This interest is motivated largely by the HPF programming style, in which science applications are decomposed into phases. At each phase, there is an optimal distribution of the arrays onto the processor grid. Because the optimal distribution changes from phase to phase, the array redistribution turns out to be a critical operation [2][11][13].

Generally, the redistribution problem comprises the following two subproblems [7][11][13]:

† This research was supported by KOSEF under grant 981-0925-130-2

- Message generation: The array to be redistributed should be efficiently scanned or processed in order to build all the messages that are to be exchanged between processors.
- Communication schedule: All the messages must be efficiently scheduled so as to minimize communication overhead. Each processor typically has several messages to send to all other processors.

This paper describes efficient and practical algorithms for redistributing arrays between different GEN_BLOCK distributions. The "generalized" block distribution, GEN_BLOCK, which is supported in High Performance Fortran version 2, allows contiguous segments of an array, of possibly unequal sizes, to be mapped onto processors [1], and is therefore useful for solving load balancing problems [14]. The sizes of the segments are specified by values of a user-defined integer mapping array, with one value per target processor of the mapping.

Since the address calculation and message generation processes in GEN_BLOCK redistribution are relatively simple, this paper focuses on an efficient communication schedule. The simplest approach to designing a communication schedule is to use nonblocking communication primitives. The nonblocking algorithm, which is a communication algorithm using nonblocking communication primitives, avoids excessive synchronization overhead, and is therefore faster than the blocking algorithm, which is a communication algorithm using blocking communication primitives. However, the main drawback of the nonblocking algorithm is its need for as much buffering as the data being redistributed [6][11][12].

There is a significant amount of research on regular redistribution — redistribution between CYCLIC(n) and CYCLIC(m). Message passing in regular redistribution involves cyclic repetitions of a basic message passing pattern, while GEN_BLOCK redistribution does not show such regularity. There is no repetition, and there is only one message passing between two processors. There is presently no research concerning GEN_BLOCK redistribution. This paper presents a scheduling algorithm for GEN_BLOCK redistribution using blocking communication primitives and compares it with nonblocking one.

The paper is organized as follows. In Section 2, the characteristics of blocking and nonblocking message passing are presented. Some concepts about and definitions of GEN_BLOCK redistribution are introduced in Section 3. In Section 4, we present an optimal scheduling algorithm. In Section 5, a performance evaluation is conducted. The conclusion is given in Section 6.

2. Communication Model

The interprocessor communication overhead is incurred when data is exchanged between processors of a coarse-grained parallel computer. The communication overheads can be represented using an analytical model of typical distributed memory machines, the General purpose Distributed Memory (GDM) model [7]. Similar models are reported in the literature [2], [5], [6] and [11]. The GDM model represents

the communication time of a message passing operation using two parameters: the start-up time t_s and the unit data transmission time t_m.

The start-up time is incurred once for each communication event. It is independent of the message size to be communicated. This start-up time consists of the transfer request and acknowledgement latencies, context switch latency, and latencies for initializing the message header. The unit data transmission time for a message is proportional to the message size. Thus, the total communication time for sending a message of size m units from one processor to another is modeled as Expression 1. A permutation of the data elements among the processors, in which each processor has maximum m units of data for another processor, can be performed concurrently in the Expression 1 time. The communication time of collective message passing delivered in multiple communication steps is the sum of the communication time of each step as shown in Expression 2.

$$T_{STEP} = t_s + m \times t_m \quad (1)$$

$$T_{TOTAL} = \sum T_{STEP} \quad (2)$$

Since the model does not allow node contention at the source and destination of messages, a processor can receive a message from only one other processor in every communication step. Similarly, a processor can send a message to only one other processor in each communication step. Contention at switches within the interconnection network is not considered. The interconnection network is modeled as a completely connected graph. Hence, the model does not include parameters for representing network topology. This assumption is realistic because of architectural features such as virtual channels and cut-through routing in state-of-art interconnection networks. Also, the component of the communication cost that is topology dependent is insignificant compared to the large software overheads included in message passing time.

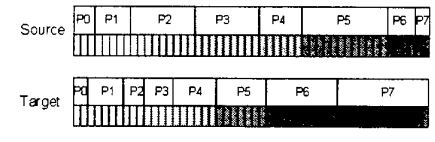
3. GEN_BLOCK redistribution

This section illustrates the characteristics of GEN_BLOCK redistribution and introduces several concepts. GEN_BLOCK, which allows each processor to have different sizes of data blocks, is useful for load balancing [1]. Since, for dynamic load balancing, current distribution may not be effective at the next time, the redistribution between different GEN_BLOCKs like Fig. 1(a) is necessary.

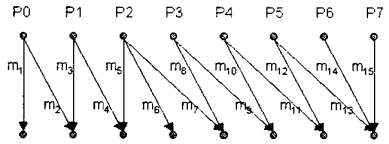
The GEN_BLOCK redistribution causes collective message passing, as can be seen in Fig. 1(b). Unlike regular redistribution, which has a cyclic message passing pattern the message passing occurs in a local area. Suppose there are four processors: Pi, Pl-1, Pl, Pl+1. If Pi has messages for Pl-1 and Pl+1, there should be a message from Pi to Pl. In the same way, if there are messages from Pl-1 to Pi and from Pl+1 to Pi, there should be a message from Pl to Pi. For example, in Fig. 1(b), P2 sends messages to P2, P3, and P4, and P7 receives messages from P5, P6, and P7. We call this "Spatial Locality of Message Passing."

Same sender (or receiver) messages cannot be sent in a communication step. In this paper, we call a set of those messages "Neighbor Message Set." One characteristic of a neighbor message set is that there are, at most, two link messages. As shown in Fig. 1(c), some messages can be included in two neighbor message sets; we call this "Link Message." Another characteristic is that the chain of neighbor message sets is not cyclical, which means that there is an order between them. We define "Schedule" as a set of communication steps. In this paper, we assume that the communication steps are ordered by their communication time. Fig. 1(d) is an example of the schedule. The schedule completes redistribution with three communication steps.

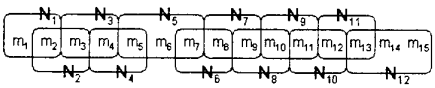
There may be many schedules for a redistribution. Variations of a schedule are easily generated by changing the communication steps of a message. For example, by moving m_7 to Step 1, we can make a new schedule. However, m_6 is already in Step 1, and since m_6 and m_7 are in the same neighbor set, m_6 must be moved to a different step. If m_8 moves from Step 3 to Step 2, then m_2, m_4, and m_5, as well as m_7 and m_8, have to be relocated. We define these two message sets that cannot be located in the same communication step as a "Conflict Tuple." For example, between Step 2 and Step 3, there are two conflict tuples: ($\{m_3, m_7\}$,$\{m_5,m_8\}$) and ($\{m_{14}\}$,$\{m_{15}\}$). Fig. 1(e) shows all conflict tuples of the schedule in Fig. 1(d).



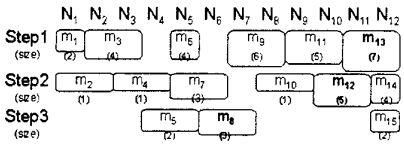
(a) Source and Target Distribution Patterns



(b) Messages and Spatial Locality



(c) Msg. Ordering and Neighbor Msg. Sets



(d) Schedule Table

Step1 - Step2 ({m_1, m_3},{m_2, m_4}), ({m_6},{m_7}), ({m_9, m_{11}, m_{13}},{m_8, m_{10}, m_{12}, m_{14}})
Step1 - Step3 ({m_6},{m_5}), ({m_9},{m_8}), ({m_{13}},{m_{15}})
Step2 - Step3 ({m_4, m_7},{m_5, m_8}), ({m_{14}},{m_{15}})

(e) Conflict Tuples

Fig. 1. GEN_BLOCK redistribution

4. Relocation Scheduling Algorithm

To save redistribution time, it is necessary to reduce the number and the size of communication steps. In this section, we propose two optimal conditions and an optimal scheduling algorithm for GEN_BLOCK redistribution.

4.1 Minimal Step Condition

According to Expressions 1 and 2, the influence of message suffling pattern or non-maximum message sizes in a communication step are not crucial, but the communication time linearly increases as the number of communication steps increase. Therefore, reducing the number of communication steps is important. In this section, we propose a minimal step condition for GEN_BLOCK redistribution.

- **Minimal Step Condition:** Minimum number of communication steps is maximum cardinality of neighbor sets.

Since the messages in a neighbor message set cannot be sent in a communication step, there should be more communication steps than the maximum cardinality of neighbor message sets. According to the characteristics of neighbor message sets, there are at most two link messages in a neighbor message set. All the link messages can be located within two communication steps by using a zigzag locating mechanism like m_1, m_2, m_3 and m_4, shown in Fig. 1(d). The remaining messages in neighbor sets are then all independent and possible to locate in any communication step. Therefore, the minimum number of communication steps is maximum cardinality of neighboring sets.

4.2 Minimal Size Condition

As shown in Expression 1, reducing the sizes of maximum messages in each step is important for efficient redistribution. In this section, we propose another optimal scheduling condition for the GEN_BLOCK redistribution.

- **Minimal Size Condition**: For all conflict tuples *(M, M')* in Schedule *C*, $SZ_M \geq SZ_{M'}$.

The SZ_N is a size of *N*. *N* can be a message, a communication step, or a schedule. If *N* is a message, SZ_N is the size of the message. If *N* is a communication step, SZ_N is the size of maximum message in *N*. And if *N* is the schedule, SZ_N is a sum of the size of communication steps in *N*.

Suppose a communication schedule *C1* has a conflict tuple that does not hold the minimal size condition, which means that there is a conflict tuple *(M, M')* between Communication Steps S_i and S_j $(i > j)$ such that $SZ_M < SZ_{M'}$. Let's consider another schedule, *C2*, that is the same as *C1* except for the Message Set *M* and *M'*. In *C2*, *M* is

in S_j and M' is in S_i. To understand this easily, we express S_i and S_j in $C1$ as $S1_i$ and $S1_j$, and S_i and S_j in $C2$ as $S2_i$ and $S2_j$. Because of the definitions of Communication Step and Schedule, and Expressions 1 and 2, $SZ_{S1i} > SZ_M$ and $SZ_{S1j} \geq SZ_{M'}$. If $SZ_{S1i} > SZ_M$ and $SZ_{S1j} = SZ_{M'}$, then $SZ_{S2i} > SZ_{M'}$, therefore $SZ_{C1} = SZ_{C2}$. In case $SZ_{S1i} > SZ_M$ and $SZ_{S1j} > SZ_{M'}$, if there is a message m in $S2$ such that SZ_m is less than $SZ_{M'}$ but greater than any other messages in $S2$, then $SZ_{C2} = SZ_{C1} - SZ_{M'} - SZ_m$, otherwise $SZ_{C2} = SZ_{C1} - SZ_{M'} - SZ_M$. In both cases, $SZ_{C2} < SZ_{C1}$. Therefore, if there is a schedule $C1$ that does not hold the minimal size condition, then we can always makes a schedule $C2$ that is not worse than $C1$ by changing the communication steps of M and M'.

Unfortunately, this proof does not guarantee that all schedules that satisfy the minimal size condition are optimal. However, according to the various experiments in Section 5, we show that a schedule that satisfies the condition shows better performance.

4.3 Relocation Scheduling Algorithm

Algorithm 1 is a scheduling algorithm that generates an optimal communication schedule. The schedule satisfies minimal step condition and minimal size condition.

The algorithm consists of two phases. In the first phase, it performs size-oriented list scheduling [15] (lines 6-12). In the processing of the first phase, when it cannot allocate a message without violating the minimal step condition, it goes to the next

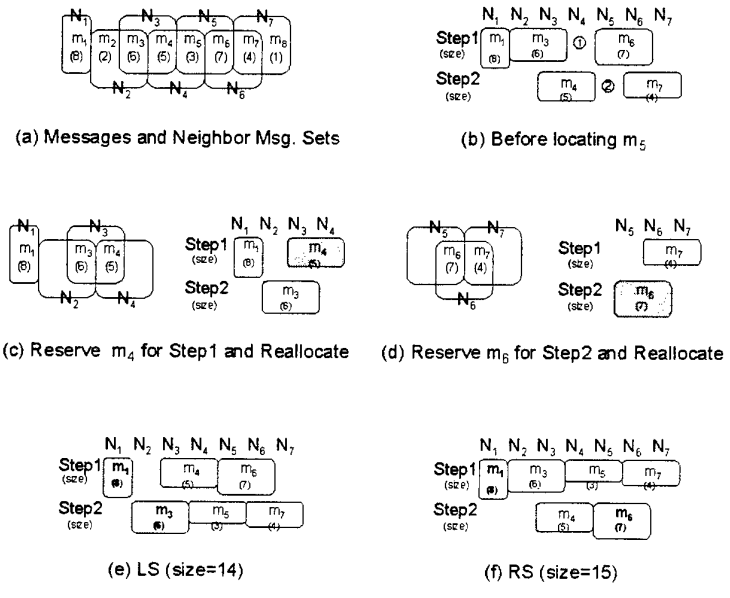


Fig. 2. Relocation Scheduling Algorithm

phase — the relocation phase. In the relocation phase, it relocates already allocated messages and makes space for the new message.

Some redistributions can be scheduled through only the list scheduling phase. In this case, because a larger message is located in an earlier step, the minimal size condition is satisfied. The schedule in Fig. 1(d) is generated through only the list scheduling phase. The redistribution in Fig. 2(a) is an example that cannot be

```
Algorithm: locate
  input M : Message Set sorted by size and reservation
     rn : reserved neighbor set
  output S : Schedule
{
 1    Sort(M);
 2    SNT(1, rn) = 1
 3    for (i=0; i<card(M); i++) {
 4       m = M(i);
 5       for (j=0; j<card(S); j++) {
 6          s = S(j)
 7          If ((SNT(s,nb(m,1))==0) && (SNT(s,nb(m,2))==0)) {
 8             SNT(s,nb(m,1)) = 1;
 9             SNT(s,nb(m,2)) = 1;
 10             Insert(m, s);
 11             Go to next;
 12       } }
 13       LS = replace(S, locate(LM, nb(m,1)));
 14       RS = replace(S, locate(RM, nb(m,2)));
 15       if (sz(LS) < sz(RS)) S = LS;
 16       else S = RS;
 17 next:
 18    }
 19    return S;
  }
-   sort(M) sorts the messages in M by size.
-   SNT(s,n) is flag to indicate "there is n-th neighbor set message in step s."
-   card(S) is a cardinality of set S.
-   nb(m,i) is i-th neighbor set of message m.
-   Insert(m,s) inserts message m into step s.
-   replace(S,S') returns a new schedule in which the position of m in S are
    replaced as that in S'.
-   sz(S) is a redistribution size of schedule S.
```

Algorithm 1. Relocation Scheduling Algorithm

scheduled only through list scheduling. Because the maximum cardinality of its neighbor sets is 2, the messages have to be scheduled within two communication steps.

The input message set M of the relocation scheduling algorithm is $\{m_1, m_6, m_3, m_4, m_7, m_5, m_2, m_8\}$. The algorithm schedules m_1, m_6, m_3, m_4, and m_7 with list scheduling, but because there are already m_6 in Step 1 and m_4 in Step 2, m_5 cannot be located in both Step 1 and Step 2. The control is then passed to the relocation phase. Because the message m_5 is included in neighbor set N4 and N5, it has to be placed in ① or ②. In the relocation process, the schedule is divided into two sub message sets. One, which we term the left set, is composed of the messages before N4, and the other, termed the right set, is composed of the messages after N5. The left set and the right set are kinds of GEN_BLOCK redistributions and can be seen as an input of the relocation scheduling algorithm, recursively. In case of the left set, to make a space in Step 2 for m_5, m_4 is reserved in Step 1 before the recursive call. When the recursive call returns, we get a new subschedule in Fig. 2(c), merge it with the original schedule, and make a new schedule in Fig. 2(e). In case of the right set, we also make another schedule in Fig. 2(f), but we discard it because it does not satisfy the minimal size condition.

5. Performance Evaluation and Experimental Results

This section shows timing results from implementations of our redistribution algorithm on CRAY T3E and IBM SP2. The algorithms and node programs were coded in Fortran 77, and MPI primitives were used for interprocessor communication. We use a random function for a series of GEN_BLOCK distributions in the experiment.

To evaluate the relocation scheduling algorithm in various redistribution environments, we have assumed three changing load situations: stable, moderate, and unstable. In the stable case, the changes are not heavy. Therefore, there are a small number of messages, and the average size of the messages is also relatively small. To make the series of distributions for this case, the standard deviation of block size is limited to less than 10 % of average block size. As the situation becomes moderate and unstable, the changing loads in each processor become heavy. Thus, for the moderate and unstable situations, we assume that the standard deviation of block sizes is between 45 % and 55 % and between 90 % and 100 % of average block size for each.

In order to evaluate the effects of the minimal size condition and the minimal step condition, we use the following five scheduling algorithms.

- NBLK is a schedule for a redistribution that uses non-blocking communication primitives.
- OPT is an optimal schedule for blocking communication that satisfies the minimal step condition and the minimal size condition. The schedule is generated by the relocation scheduling algorithm.
- MIN_STEP is a minimal step schedule using blocking communication primitives. For this schedule, we use a random list schedule, and to keep the minimal step condition, we use the relocation phase of the relocation algorithm.

- MIN_SIZE is a minimal size schedule using blocking communication primitives. For this schedule, we use the size-oriented list schedule.
- RANDOM is a random schedule using blocking communication primitives. For this schedule, we simply use a random list schedule. Therefore, it does not guarantee both the minimal step condition and the minimal size condition.

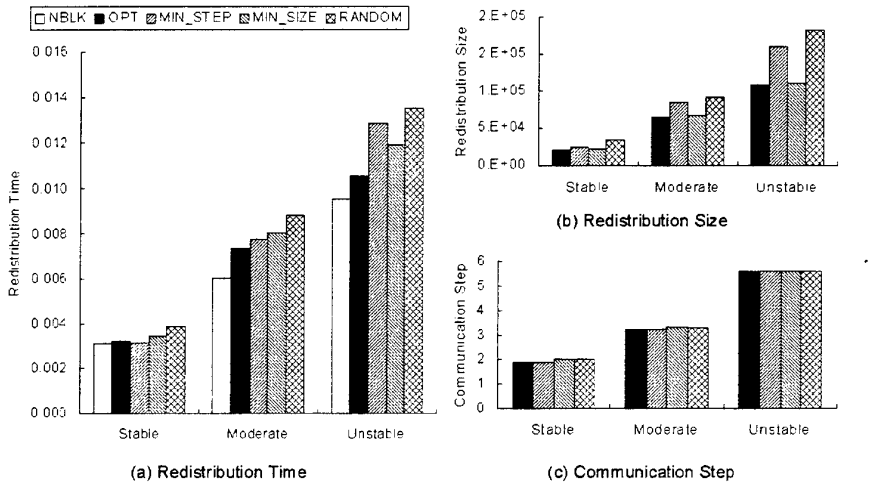


Fig. 3. Redistribution Times at CRAY T3E

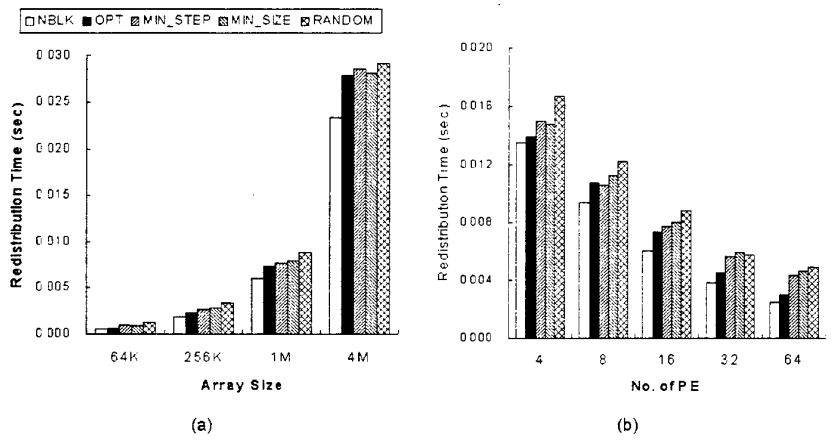


Fig. 4. The effects of number of processor and array size at CRAY T3E

5.1. Results from Cray T3E

Fig. 3 shows the difference between performances through the algorithms on Cray T3E. Here, the performance of the NBLK, OPT, MIN_STEP, MIN_SIZE, and RANDOM schedules are measured on 16 processors. The size of the array used is 1M floating points. From Fig. 3(a), as expected, the NBLK schedule shows the best performance. Between the schedules using blocking primitives, it can be seen that the OPT schedule takes the least amount of redistribution time, whereas the RANDOM schedule takes the longest. The redistribution times for the MIN_SIZE schedule and the MIN_STEP schedule vary.

Fig. 3(b) shows redistribution size of each algorithm. The redistribution size is the sum of the maximum message sizes of each step. According to Fig. 3(b), as the changing load situation becomes worse, the redistribution sizes increase. In the graph, we can see that the gradients of the MIN_STEP schedule and the RANDOM schedules are steeper than those of the OPT schedule and the MIN_SIZE schedule. Because of the influence of the redistribution size, the graph in Fig. 3(a) shows that the redistribution time of MIN_STEP schedule and the RANDOM schedule become longer more quickly than those of the OPT schedule and the MIN_SIZE schedule.

An influence of the communication step is inferred from the gap between the redistribution time of the OPT schedule and the MIN_SIZE schedule. According to Expressions 1 and 2, there are two factors that determine the redistribution time: the number of communication steps, and the redistribution size. As shown in Fig. 3(b), the redistribution size of the two schedules is almost the same. Hence, the difference in redistribution time is caused by the slight gap in the communication step.

The trends in Fig.3 are shown in experiments performed with different processor numbers and different array sizes. In Fig. 4(a), the array size varies from 64K to 4M, and in Fig. 4(b), the number of processors varies from 4 to 64. From these graphs, we observed that, between redistribution schedules for blocking communication primitives, the OPT schedule achieves better performance than the other schedules. The results presented in this section show that for GEN_BLOCK redistribution, it is essential to satisfy the minimal step condition and the minimal size condition.

5.2. Results from IBM SP2

There are slightly different results when the experiment is performed in IBM SP2. In this section, we present and analyze the results from IBM SP2.

According to Fig. 5, OPT shows similar speed with MIN_SIZE, and MIN_STEP schedule is even worse than RANDOM schedule. These mean that, in IBM SP2, Minimal Step Condition is not helpful for redistribution, and sometimes it is harmful.

It is because IBM SP2 does not satisfy the assumed communication model, GDM. As stated in section 2, the interconnection network in GDM is completely connected graph, and contention at switch in interconnection is not considered. To examine IBM SP2 and CRAY T3E satisfy GDM model, we perform an experiment in Fig. 6. The experiment checks communication time of different shuffling situations, such as: one

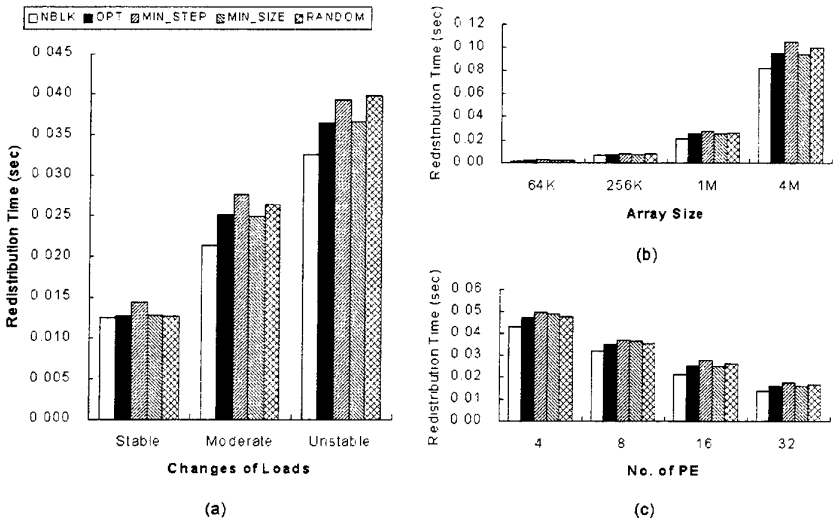


Fig. 5. Redistribution Times in IBM SP2

directional message passing between two processors (1D), bi-directional two message passings between two processors (2D), shift style three message passings between four processors (SH) and cyclic shift style four message passings between four processors, and in case 2D, we changed one message to be 1/4, 2/4, 3/4 and 4/4 of the other. In this experiment, CRAY T3E shows relatively fixed performance, but IBM SP2 takes various communication times according to the number of messages and message size. SP2 takes more time in 2D, SH and C-SH than 1D, and it also takes more time as message size increases in the variation of 2D. These mean that Expression 1 is no longer valid in SP2, and increasing the degree of parallelism is not always good. This might be due to the architectural difference between the two systems. The interconnection network of CRAY T3E is a 3D Torus and that of IBM SP2 is a multistage interconnection network (MIN); it is generally known that the probability of contention in MIN is higher than Mash or Torus structure such as CRAY T3E[16][17].

These experiments demonstrate that the proposed scheduling algorithm shows good performance when the communication model is close to GDM model.

6. Related Work

Many methods for performing array redistribution have been presented in the literature.

Kalns et al. proposed a processor mapping technique to minimize the amount of data exchange for BLOCK-to-CYCLIC(n) redistribution. Using the data to logical processors mapping, they showed that the technique can achieve the maximum ratio between data retained locally and the total amount of data exchanged [9].

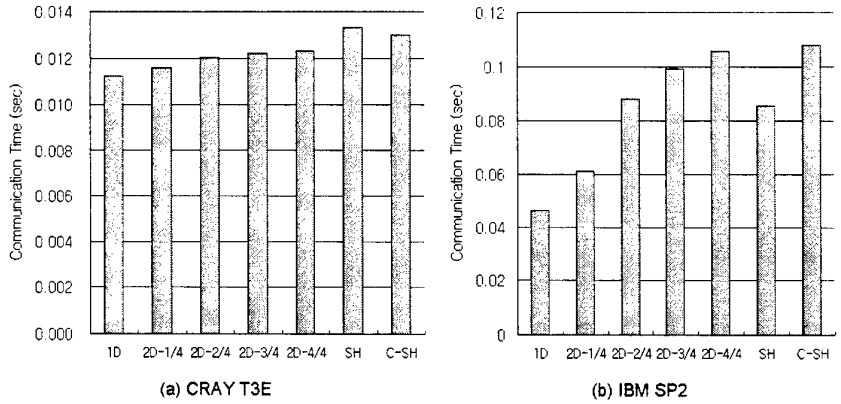


Fig. 6. Communication Time of message sufflings in (a) CRAY T3E and (b) IBM SP2

Gupta et al. derived closed form expressions for determining the send/receive processor/data sets for the BLOCK-to-CYCLIC redistribution. They also provided a virtual processor approach to address the problem of reference index-sets identification for array statements with CYCLIC(n) distribution and formulated active processor sets as closed forms [10].

Thakur et al. proposed algorithms for runtime array redistribution algorithm for BLOCK-to-CYCLIC(m) and CYCLIC(n)-to-CYCLIC(kn). Based on these algorithms, they proposed a two phase redistribution method for CYCLIC(m)-to-CYCLIC(n) using LCM or GCD of m and n [4]. In the same paper, Thakur et al. proposed some ideas that are generally accepted in the later papers. One is a general redistribution mechanism, LCM method, for redistribution between CYCLIC(m)-to-CYCLIC(n). Another is the effect of indirect communication, which was expanded to multiphase redistribution mechanism by Kaushik et al. [5]. The last is an evaluation of the non-blocking communication, which Thakur et al. presented as more efficient than blocking communication because the computation and communication are performed in parallel. Following research has focused on the redistribution between CYCLIC(n)-to-CYCLIC(kn) using non-block communication primitives.

Walker et al. posited that the performance of the redistribution algorithm using block communication primitives was comparable to that of nonblocking redistribution [6].

Lim at el. in [7] proposed a generalized circulant matrix formalism to reduce the communication overheads for CYCLIC(n)-to-CYCLIC(kn) redistribution. Based on the generalized circulant matrix formalism, they proposed direct, indirect, and hybrid communication schedules. They demonstrated that an indirect communication schedule using blocking primitives shows better performance than redistribution using asynchronous communication primitives.

7. Conclusion

In this paper, we propose a new scheduling algorithm for redistribution between different GEN_BLOCKs.

In spite of excessive synchronization overhead, many programs use blocking communication primitives because of the cost of communication buffers. However, to avoid deadlock and poor performance, the messages using blocking communication primitives have to be well scheduled. This paper analyzes the characteristics of communication primitives in MPI and proves that a Minimal Step Condition and a Minimal Size Condition are essential in blocking GEN_BLOCK redistribution. Moreover, by adding a relocation phase to list scheduling, we make an optimal scheduling algorithm: a relocation scheduling algorithm.

In section 5, various experiments on CRAY T3E and IBM SP2 were performed to evaluate the proposed algorithm and analyze the influences of the optimal conditions in a real environment. According to the experiments, it was proven that the relocation scheduling algorithm shows better performance and that the optimal conditions are critical in enhancing the communication speed of GEN_BLOCK redistribution in GDM model.

References

1. High Performance Fortran Forum, *High Performance Fortran Language Specification version 2.0*, Rice University, Houston, Texas, October 1996.
2. Yeh-Ching Chung, Ching-Hsien Hsu, and Sheng-Wen Bai, "A Basic-Cyclic Calculation Technique for Efficient Dynamic Data Redistribution," *IEEE Transaction on Parallel and Distributed Systems*, Vol.9, No.4, April 1998.
3. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, University of Tennessee, Knoxville, Tennessee, May 1994.
4. Rajeev Thakur, Alok Choudhary, and Geoffrey Fox, "Runtime Array Redistribution in HPF Programs," *Proceedings of SHPCC'94*, pp.309-316, 1994.
5. S.D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan, "Multi-Phase Array Redistribution: Modeling and Evaluation," *Proceedings of 9th International Parallel Processing Symposium*, pp.441-445, April 1995.
6. David W. Walker, Steve W. Otto, "Redistribution of Block-Cyclic Data Distribution Using MPI," *Concurrency: Practice and Experience*, Vol.8 No.9, pp.707-728, November 1996.
7. Young Won Lim, Prashanth B. Bhat, and Viktor K. Prasanna, "Efficient Algorithm for Block-Cyclic Redistribution of Arrays," *IEEE Symposium on Parallel and Distributed Process*, October 1996 and will be published in Algorithmica.
8. James M. Stichnoth, David O'Hallaron, and Thomas R. Gross, "Generating Communication for Array Statements: Design, Implementation, and Evaluation," *Journal of Parallel Distributed Computing*, pp.150-159, April 1994.
9. Edger T. Kalns and Lionel M. Ni, "Processor Mapping Techniques Toward Efficient Data Redistribution," *Proceedings of the 8th International Parallel Processing Symposium*, April 1994, Cancun, Mexico

10.S.K.S Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan, "Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines," *Technical Report OSU-CISRC-4.*

11.Frederic Desprez, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert, "Scheduling Block-Cyclic Array Redistribution," *IEEE Transactions on Parallel and Distributed Systems*, Vol.9, No.2, February, 1998 also presented in CRPC-TR97714-S, February 1997.

12.Rajeev Thakur, Alok Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution," *IEEE Transactions on Parallel and Distributed Systems*, Vol.7 No.6, June 1996.

13.Minyi Guo, Ikauo Nakata, and Yoshiyuki Yamashita, "Contention-Free Communication Scheduling for Array Redistribution," *Proceedings of the International Conference on Parallel and Distributed Systems*, pp.658-667, December 1998.

14.High Performance Fortran Forum, *HPF-2 Scope of Activities and Motivating Applications*, November 1994.

15.E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, Jon Wiley & Sons, New York, 1976.

16.G. F. Pfister and V. A. Norton, "Hot spot contention and combining in multistage interconnection networks," *IEEE Transactions on Computers*, vol. 34, pp. 943-948, Oct. 1985.

17.Jose Duato, Sudhakar Yalamanchili, and Lionel Ni, *Interconnection Networks*, IEEE Computer Society Press, pp. 155, 1997.

Finite/Discrete Element Analysis of Multi-fracture and Multi-contact Phenomena

D.R.J. Owen, Y.T. Feng, J. Yu, and D. Perić

Department of Civil Engineering, University of Wales Swansea
Swansea, SA2 8PP, UK
{D.R.J.Owen, Y.Feng, J.Yu, D.Peric}@swansea.ac.uk

Abstract. A dynamic domain decomposition strategy is proposed for the effective parallel implementation of combined finite/discrete element approaches for problems involving multi-fracture and multi-contact phenomena. Attention is focused on the parallelised interaction detection between discrete objects. Two graph representation models, are proposed and a load imbalance detection and re-balancing scheme is also suggested. Finally, numerical examples are provided to illustrate the parallel performance achieved with the current implementation.

1 Introduction

The last several decades have witnessed the great success of the finite element method, along with a tremendous increase of computer power, as a numerical simulation approach for applications across almost every engineering discipline. However, for situations involving *discrete* or *discontinuous* phenomena, a combined finite/discrete element method naturally offers a more powerful solution capability. Typical examples that can considerablely benefit from this combined solution strategy include process simulation (e.g. shot peening, granular flow, and particle dynamics) and fracture damage modelling (e.g. rock blasting, mining applications, and projectile impact).

Besides their discrete/discontinuous nature, these applications are often characterised by the following additional features: highly *dynamic*; rapidly *changing domain configurations*; *sufficient resolution* required; and *multi-physics phenomena* involved. In the numerical solution context, *contact detection and interaction computations* often take more than half of the entire simulation time and the *small time step* imposed in the explicit integration procedure also gives rise to the requirement of a very large number (e.g. millions) of time increments to be performed. For problems exhibiting multi-fracturing phenomena, the necessity of frequent introduction of new physical cracks and/or adaptive re-meshing at both local and global levels adds another dimension of complexity. All these factors make the simulation of a realistic application to be extremely *computational intensive.*

Consequently, parallelisation becomes an obvious option for significantly increasing existing computational capability, along with the recently remarkable

advances in hardware performance. In recent years considerable effort has been devoted to the effective parallel implementation of conventional finite element procedures, mainly based on a *static domain decomposition* concept. The features itemised above associated with the applications of interest make such a parallelisation much more difficult and challenging. Only very recently, have some successful attempts emerged at tackling problems of a similar nature[1–3].

It is evident that a *dynamic domain decomposition* (DDD) strategy plays an essential role in the success of any effective parallel implementation for the current problem. The ultimate goal of this work is therefore to discuss the major algorithmic aspects of dynamic domain decomposition that make significant contributions to enhancing the parallel performance. Our implementation is also intended to be general for both shared and distributed memory parallel platforms.

The outline of the current work is as follows: In the next section, a general solution procedure of the combined finite/discrete element approach for problems involving material discontinuity and failure is reviewed and a more detailed description is devoted to the multi-fracture modelling and the interaction detection; Section 3 presents dynamic domain decomposition parallel strategies for the combined finite/discrete element approach and parallel implementation for both the finite element computation and the global search is also investigated. Issues relevant to the dynamic domain decomposition algorithm for the contact interaction computation, mainly the two graph representation models for discrete objects, is proposed in the next section. Section 5 is devoted to the description of a load imbalance detection and re-balancing scheme. Finally, numerical examples are provided to illustrate the parallel performance achieved with the current implementation.

2 Solution Procedures of Combined Finite/Discrete Element Approaches

Engineering applications involving material separation and progressive failure can be found, among others, in masonry or concrete structural failure, demolition, rock blasting in open and underground mining and fracture of ceramic or glass-like materials under high velocity impact. The numerical simulation of such applications, especially in large scale, has proved to be very challenging. The problems are usually represented by a small number of discrete bodies prior to the deformation process. In the combined finite/discrete element context, the deformation of each individual body is modelled by the finite element discretisation and the inter-body interaction is simulated by the contact. During the simulation process, the bodies are damaged, by, for example, tensile failure, and modelling of the resultant fragmentation may result in possibly two or three orders of magnitude more bodies by the end of the simulation. In addition, the substantial deformation of the bodies may necessitate frequent adaptive remeshing of the finite element discretisation. Therefore the configuration and number

of elements of the finite element mesh and the number of bodies are continuously changing throughout the simulation.

Similarly, process engineering applications often contain a large number of discrete bodies. In many cases, these bodies can be treated as *rigid* and represented by *discrete elements* as simple geometric entities such as disks, spheres or ellipses. Discrete elements are based on the concept that individual material elements are considered to be separate and are (possibly) connected only along their boundaries by appropriate physically based interaction laws. The response of discrete elements depends on the interaction forces which can be short-ranged, such as mechanical contact, and/or medium-ranged, such as attraction forces in liquid bridges.

Contact cases considered in the present work include node to edge/facet and edge/facet to edge/facet. Short-range mechanical contact also include disk/sphere to disk/sphere and disk/sphere to edge/facet. Medium-range interactions can be represented by appropriate attractive relations between disk/sphere and disk/sphere entities.

The governing dynamic equations of the system are solved by explicit time integration schemes, notably the central difference algorithm.

2.1 Procedure description

In the context of the explicit integration scheme, a combined finite and discrete element approach typically performs the following computations at each time step:

1. Finite element and fracture handling:
   - Computation of internal forces of the mesh;
   - Evaluation of material failure criterion;
   - Creation of new cracks if any;
   - Global adaptive re-meshing if necessary;
2. Contact/interaction detection;
   - Spatial search: detection of potential contact/interaction pairs among discrete objects;
   - Interaction resolution: determination of actual interaction pairs through local resolution of the kinematic relationship between (potential) interaction pairs;
   - Interaction forces: computation of interaction forces between actual interaction pairs by using appropriate interaction laws.
3. Global solution: computation of velocities and displacements for all nodes;
4. Configuration update: update of coordinates of all finite element nodes and positions of all discrete objects;

The procedures of finite element internal force computation, equation solution and configuration update in the above approach are all standard operations, and therefore further description is not necessary. However, the fracture modelling and the interaction detection warrant further discussion.

2.2 Multi-fracturing modeling

Two key issues need to be addressed for successful modelling of material failure: (i) the development of constitutive models which reflect the failure mechanism; (ii) the ability of numerical approaches to handle the discontinuities such as shear bands and cracks generated during the material failure and fracture process.

Failure models A variety of constitutive models for material failure have appeared over the years, with softening plasticity and damage theory being the two most commonly adopted in the nonlinear finite element analysis of failure. For brittle materials, a simple Rankine failure model can be employed. After initial yield, a rotating crack formulation may be employed in which the anisotropic damage is modelled by degrading the elastic modulus in the direction of the current principal stress invariant.

For all fracture or localisation models regularisation techniques must be introduced to render the mesh discretisation objective. Optional formulations include non-local damage models, Cosserat continuum approaches, gradient constitutive models, viscous regularisation and fracture energy releasing/strain softening approaches. All models effectively result in the introduction of a length scale and have specific advantages depending on the model of fracture and loading rate.

More detailed description of various fracture models can be found in [4].

Topological update scheme The critical issue of fracture modelling is how to convert the continuous finite element mesh to one with discontinuous cracks and to deal with the subsequent interactions between the crack surfaces. The most general approaches permit fracture in an arbitrary direction within an element and rely on local adaptive re-meshing to generate a well-shaped element distribution.

A particular fracture algorithm is developed in this work to model the failure of brittle materials subject to impact and ballistic loading. The fracture algorithm inserts physical fractures or cracks into a finite element mesh such that the initial continuum is gradually degraded into discrete bodies. However, the primary motivation for utilising the algorithm is to correctly model post-failure interaction of fractures and the motion of the smaller particles created during the failure process.

Within this algorithm, a nodal fracture scheme is employed to transfer the virtual smeared crack into a physical crack in a finite element mesh. The scheme is a three stage procedure: (i) Creation of a failure map for the whole domain; (ii) Assessment of the failure map to identify where fractures should be inserted; (iii) Updating of the mesh, topology and associated data.

In 2-D cases, the failure direction is defined to coincide with the maximum failure strain direction and the crack will propagate orthogonal to the failure direction. Associated with the failure direction, a failure plane is defined with the failure direction as its normal and the failed nodal point lying on the plane. A crack is then inserted through the failure plane. If a crack is inserted exactly

through the failure plane, some ill-shaped elements may be generated. Local remeshing is then needed to eliminate them. Alternatively, the crack is allowed to propagate through the element boundary. In this way, no new elements are created and the updating procedure is simplified. However, this procedure necessitates a very fine mesh discretisation around the potential fracture area. Within the current algorithm a minimum element area criterion is used to ensure that excessively small sliver elements are not created. If the element area obtained by splitting the elements is below this threshold value then the fracture is forced to be along element boundaries. For 3-D situations, the corresponding fracture algorithm is basically the same but the implementation procedure is more complicated.

In addition, an element erosion procedure is also applied to deal with the situation that the material represented by the element to be eroded no longer contributes to the physical response for the problem, such as the case where the material is melted down or evaporated at high temperature or is transformed into very small particles.

2.3 Interaction detection

The interaction contact detection comprises three phases: (global) spatial search, (local) interaction resolution and interaction force computation.

The spatial search algorithm employed is a combination of the standard space-cell sub-division scheme with a particular tree storage structure, the *Argumented Digital Tree* (ADT) [5], and can accommodate various geometrical entities including points, facets, disks/spheres and ellipses. Each entity is represented with a bounding box extended with a buffer zone. The size of the buffer zone is a user-specified parameter. In the case of medium-range interaction, it must not be smaller than the influence distance of each object considered. The algorithm eventually locates for each object (termed the *contactor*) a list of neighbouring objects (termed *potential targets*) that may potentially interact with the contactor.

In the second phase, each potential contactor-target pair is locally resolved on the basis of their kinematic relationship, and any pair for which interaction does not occur, including medium range interaction if applied, is excluded. In the final phase, the interaction forces between each actual interaction pair are determined according to a constitutive relationship or interaction law.

Effects of buffer zone sizes The global spatial search may not necessarily be performed at every time step, as will be described below, while the interaction resolution and interaction force computations should be conducted at each time step. Furthermore, some kinematic variables computed in the interaction resolution phase will be used in the force computation. For these reasons, the interaction resolution and force computations are actually performed together in the current implementation.

The computational cost involved in the interaction force computation phase is fixed at each time step, if no new surfaces are created during the fracturing

process. It may, however, vary significantly for the other two phases if different sizes of buffer zone are specified.

Basically, the size of buffer zone has conflicting effects on the overall costs of the global search and the local interaction resolution. First of all, after performing a global search at one time step, a new search is required to be performed only if the following condition is satisfied

$$l_{max} > l_{buff} \tag{1}$$

where l_{buff} is the size of the buffer zone; and l_{max} is the maximum accumulated displacement of a single object for all discrete bodies in any axial direction after the previous search step:

$$l_{max} = \max_{i,j} \left(\sum_{k=1} v_i^j \Delta t_k \right) \qquad i \in [1, n_{body}], j \in [1, n_{dim}] \tag{2}$$

in which v_i^j is the velocity of object i in the j direction; Δt_k the length of time step at the k-th increment after the global search; n_{body} the total number of objects and n_{dim} the number of space dimensions.

Given a larger buffer zone, the spatial search will create a longer list of potential targets for each contactor, which will increase the amount of work in the phase of interaction resolution in order to filter out those pairs not in potential interaction. On the other hand, the global search will be performed with less frequency which leads to a reduced overall cost in the spatial search phase. With a smaller buffer zone, the potential target list is shorter and the local interaction resolution becomes less expensive at each step, but the global search should be conducted more frequently thus increasing the computational cost in searching.

A carefully selected buffer zone can balance the cost in the two phases to achieve a better overall cost in interaction detection. Nevertheless, an optimal buffer zone size is normally difficult to select for each particular application.

Incremental global search Generally speaking, the spatial search is an expensive task that often consumes a substantial portion of the total simulation time if each new search is performed independently from the previous search. The cost could however be reduced to some extent if the subsequent searches after the initial one are conducted in an *incremental* manner. In this incremental approach, the tree structure (ADT) required in the current search can be obtained as a modification of the previous structure and some search operations can also be avoided.

This approach is based on the observations that even though the configuration may undergo significant changes during the whole course of the simulation, the actual change between two consecutive search steps is bounded by the buffer zone and therefore is local. In addition, the space bisection tree is characterised by the fact that each node in the tree represents a subspace of the whole simulation domain. As long as each node (i.e. object) itself stays in the subspace

it is associated with, the whole tree will not need to be modified at all. Consequently, in the new search, it is possible to avoid building an entirely new tree by modifying only the subtrees that are affected by those objects which have moved out of their represented subspace.

The detail of the above (incremental) global search algorithm can be found in [5]. A more detailed description for interaction detection can be found in [4], while the interaction laws applied to particle dynamics, and shot peening in particular have been discussed in [6, 7].

3 Parallel Implementation Strategies

Domain decomposition based parallelisation has been established as one of the most efficient high level (coarse grain) approaches for scientific and engineering computations, and also offers a generic solution for both shared and distributed parallel computers.

A highly efficient parallel implementation requires two, often competing, objectives to be achieved: a well balanced workload among the processors, and a low level of interprocessor communication overhead.

For conventional finite element computations with infrequent adaptive remeshing and without contact phenomena, a *static* domain decomposition is generally an ideal solution that initially distributes the sub-domain data to each processor and redistributes them after each adaptive remeshing. A well load-balanced situation can be achieved if each processor is assigned an equal number of elements. Interprocessor communications can be reduced to a low level if the interface nodes that are shared by more than one sub-domain are small.

For the current situation involving both finite and discrete elements, to apply a single static domain decomposition for both finite and discrete element domains will apparently not achieve a good parallel performance due to the highly dynamic evolution of the configuration concerned. Therefore a *dynamic* domain decomposition strategy should be adopted.

3.1 Dynamic domain decomposition

The primary goal of the dynamic domain decomposition is to dynamically assign a number of discrete elements or objects to each processor to ensure good load balance as the configuration evolves.

Besides the same two objectives as for a static domain decomposition, the dynamic domain decomposition should also achieve an additional two objectives: *minimum data movement* and *efficiency*.

Firstly, completely independent domain decompositions between the consecutive steps normally give rise to very large amount of data movement involved among the processors, leading to a substantial communication overhead. Therefore, the dynamic domain decomposition should provide efficient re-partitioning algorithms that can keep the domain partitioning constant as much as possible during the simulation. Secondly, since the partitioning may need to be performed

many thousands of times during simulations, the dynamic domain decomposition must be very efficient.

Most dynamic domain decomposers can be classified into two general categories: *geometric* and *topological*. Geometric partitioners divide the computational domain by exploiting the location of the objects in the simulation, while topological decomposers deal with the connectivities of interactions instead of geometric positions. Both methods can be applied to both finite element and discrete element simulations. Generally, topological methods can produce better partitions than geometric methods, but are more expensive. A particular topological based decomposer called (Par)METIS [8] is chosen as the domain partitioner in our implementation as it appears to meet the above criterion for an effective domain decomposition algorithm.

When applying the dynamic domain decomposition strategy to the current finite/discrete element simulation, there are two different approaches. The first approach dynamically decomposes both finite and discrete element domains. Different characteristics of the two parts makes it very difficult to achieve a good load balance.

An alternative solution, which is employed in this work, is to decompose the computations associated with the two domains separately. Within the interaction detection of the discrete elements, slightly modified schemes are employed for the global search, and the combined interaction resolution and force computations. This strategy provides maximum flexibility in the implementation and thus allows the most suitable methodologies and data structures to be applied in different stages of the simulation. This may however cause extra communications in information exchange among processes due to the data structure inconsistencies between different solution stages at each time step.

Dynamic decomposition of finite element computations and a particular parallel implementation version of the global search are discussed below, while the parallel issues for the combined interaction resolution and force computation, including dynamic domain partitioning and dynamic load re-balancing, will be addressed respectively in the next two sections.

3.2 Dynamic decomposition of finite element computations

Compared to the discrete element domain, the configuration change of the finite element mesh is relatively less frequent and insignificant. The major concern for a dynamic domain decomposition algorithm is its ability to adaptively partition the domain so as to minimize the cost due to the data migration among different processors after a new partitioning.

To achieve a well-balanced workload situation often needs a fine tuning. In the case that different elements have different orders (e.g. linear or quadratic), and/or different material models, and/or different stress states (e.g. elastic or plastic), thus having different computational effort, each element should be weighted proportional to its actual cost to ensure a good load balance. In heterogeneous computer clusters, an uneven workload decomposition should be ac-

Box 1: Parallel Implementation of Initial Global Search

1. Globally distribute objects among processors using any available means of domain decomposition;
2. Each processor defines the bounding box of each object with an extended buffer zone in its subdomain;
3. Each processor computes the global bounding box of its domain and broadcasts this information to other processors;
4. Each processor applies the sequential global search algorithm to its domain and constructs its own ADT and potential target lists;
5. Each processor identifies in its domain the objects that overlap with the global bounding boxes of the other domains and then sends the objects to the processors owning the overlapping domains;
6. Each processor receives from other processors the objects that overlap with its own global bounding box; conducts the search for each object in its ADT to build the potential target list, if any; and sends the result back to the processor to which the object originally belongs;
7. Each processor collects the lists created in other processors and merges them to the original lists to obtain the final lists.

complished to take into account different computing powers that each member machine of the group can offer.

3.3 Dynamic parallelisation of global search

Global search is found to be the most difficult operation to be parallelised efficiently due to its global, irregular and dynamic characteristics. In our parallel implementation, some domain decomposition strategies are also employed to both initial and subsequent incremental search steps. Box 1 outlines the algorithmic steps involved in the initial global search.

As the initial search is conducted only once, the performance of the algorithm is not a major concern. Therefore, it can employ any available means to distribute the objects among the processors. In some cases, the algorithm can even be performed sequentially.

For the subsequent incremental searches, an effective parallel implementation of the algorithm becomes critical. Box 2 presents the corresponding parallel algorithm [9] and only the differences from the previous initial search approach are listed.

In the algorithm, it is essential to assume that a good dynamic object distribution is performed in order to achieve a good load balance, and one such distribution scheme will be proposed in the next section.

Box 2: Parallel Implementation of Incremental Global Search

1. Each processor migrates those objects that are assigned to different domains in the current partition to their new processors;
2. Each processor defines/modifies the bounding box of each object with an extended buffer zone in its subdomain;
3. Each processor updates the global bounding box of its domain and broadcasts this information to other processors;
4. Each processor modifies its own ADT and constructs its potential target lists;

5-7. Same as the Implementation in Box 1.

4 Dynamic Domain Decomposition for Interaction Computations

Many scientific and engineering applications can be abstractly expressed as weighted graphs. The vertices in the graph represent computational tasks and the edges represent data exchange. Depending on the amount of computation performed by each task, the vertices are assigned a proportional weight. Similarly, the edges are assigned weights that reflect the data needed to be exchanged.

A domain partitioning algorithm aims to assign each processor a subset of vertices whose total weight is as the same as possible so as to balance the work among the processors. At the same time, the algorithm minimise the edge-cut (subject to load-balance requirements) to minimise the communication overhead.

As the simulation evolves, the computational work associated with an object can vary, so the objects may need to be redistributed among the processors to balance the workload. The objective of dynamic re-partitioning is therefore to compute a balanced partitioning more effectively that minimises the edge-cut, and to minimise the amount of data movement required in the new partitioning.

4.1 *A priori* graph representation of discrete objects

The first important step for the development of an efficient dynamic partitioning for the interaction computation lies in an appropriate graph representation of discrete elements. Unlike a finite element mesh, discrete objects do not have underlying connectivity to explicitly associate with, and thus some form of connections among the objects should be established. An obvious choice is to use the potential target list of each contactor as its connectivity, upon which a graph representation of the interaction computation can therefore be established.

Each vertex is initially assigned a weight that is equal to the number of the targets in its potential list and no weight is placed on the edges. This weighting scheme will work reasonably well if the simulation is dominated by one type

of interaction and the objects have nearly even distribution across the whole domain.

As the above graph model is established based on *a priori* estimation of the interaction relationship and the cost of interaction force computation, it is therefore termed as the *a priori* model.

This special choice of connectivity has the following implications: First, as the potential target list will not be changed in two consecutive global searches, the graph partitioning may need to be performed only once after each global search rather than at each time step. Second, depending on the size of buffer zones specified, the list is only an approximation to the actual interaction relationship between the contactor and target. Furthermore, the actual relationship between objects is also undergoing dynamic changes at each time step. These considerations indicate that the resulting graph may not be a perfect representation of the problem under consideration. Consequently some degree of load imbalance may be expected.

4.2 *A posteriori* graph representation

In order to improve the accuracy of the above graph model, the following inherent features in the current solution procedure for the interaction resolution and force computation should be addressed:

- The target lists created by the global search provide only a potential interaction relationship between contactors and targets, and this relation can only be established after the local resolution, and therefore can not be accurately established *a priori*;
- The computation costs associated with each object, including the kinematic resolution and interaction force computation, are also unknown *a priori*, and are also very difficult to be measured precisely;
- Both the interaction relationship and the computational cost for each object are undergoing constant changes at each time step.

By specifying a very small buffer zone, the potential interaction lists can be much closer to the actual interaction relationship, but this will significantly increase the costs of simulation as indicated earlier, and therefore is not a suitable option.

In view of these facts, an alternative graph representation model is suggested. The basic idea is to use the actual information obtained in the previous time step as the base for modelling the situation at the current step. More specifically, a (nearly) accurate graph model is built for the previous time step. The graph is based on the actual contactor-target list as the connectivity and the (nearly) actual computational cost for each contactor as the vertex weighting. Since the computation domain will not undergo a significant change in two consecutive time steps as a result of a small time increment, it is reasonable to assume that this graph model is also a good representation to the problem at the current time step. Due to the *a posteriori* nature, this model is thus termed the *a posteriori graph representation.*

Another advantage of this *a posteriori* model is that the global search and the domain partitioning can now be performed at different time instances and intervals. This means that the global search is performed when the potential interaction list will be no longer valid; while a dynamic graph partitioning can be conducted when load balancing is required to be maintained. The latter aspect will be further expanded upon in the next section.

4.3 Integration with global search

In principle, both global search and graph partitioning can have totally independent data structures, mainly with different object distribution patterns. In distributed memory platforms, if the same object is assigned to different domains/processors in the search and partitioning phases, some data movement between two processors becomes inevitable. The extra communication results from the data structure inconsistency between the two phases. If the two data structures can be integrated to a certain degree, the communication overhead can also be reduced.

In the parallel implementation, the data associated with a particular object, such as coordinates, velocities, material properties, forces and history-dependent variables, is more likely to reside on the processor determined by the graph partitioning. Therefore, it is a good option for the global search to use the same object distribution pattern as generated in the graph partitioning phase, i.e. a single dynamic domain decomposition is applied to all computation phases in the interaction detection step.

Another advantage of this strategy is that a good load balance may also be achieved in the global search. This can be justified by the fact that a similar total number of potential interaction lists among the processors also implies a similar number of search operations. In addition, the incremental nature of the dynamic repartitioning employed ensures that only a small number of objects is migrated from one domain to another, and hence only a limited amount of information is required to be exchanged among the processors when re-constructing the ADT subtrees and potential target lists in the incremental global search.

5 Dynamic Load Re-Balancing

The domain partitioning algorithm is supposed to generate a well load-balanced partitioning. For various reasons, this goal may, however, be far from easy to achieve, especially when dynamic domain decomposition is involved. It is essential, therefore, to have a mechanism in the implementation that can detect the problem of load imbalance when it occurs and take proper actions to restore, or partially restore, the balance if necessary. As a matter of fact, dynamic load re-balancing is an integrated part of the dynamic domain decomposition which determines the time instances at which a dynamic re-partition should be performed. In this section, a dynamic load balancing scheme will be proposed in an attempt to enhance the performance of the parallelised interaction resolution and force computations.

5.1 Sources of load imbalance

Load imbalance may be caused by the following factors:

- A perfect partition may not be able to achieve a perfect CPU time balance due to sophisticated hardware and software issues;
- The partition produced by the domain decomposer is not perfectly balanced in terms of workload;
- In the finite element computation phases, imbalanced workload may arise when mechanical properties of some elements change, for instance, from elastic to plastic; or when local re-meshing happens that leads to some extra or fewer number of elements for certain domains;
- In the interaction resolution and force computation phases, workload unbalancing may happen for two reasons: The connectivity for each object used in the graph is only an approximation to the real interaction relationship as addressed earlier; and/or the assigned weighting for each object/edge may not represent the real cost associated with the object/edge;

The first two sources of load imbalance are beyond the scope of the present work and thus will not be given further consideration.

An important part of the load rebalancing scheme is to obtain a fairly accurate measurement of the actual cost for each object. This is however not easy to fulfill. The difficulty is due to the fact that the relative difference in computation costs between different types of interaction, such as sphere to sphere and sphere to facet, and sphere to sphere and node to facet, are difficult to define accurately.

A possible solution to this difficulty is by means of numerical experiment. We can design a series of experiments on one particular system to establish a relative cost of each element operation, including each type of interaction resolution and force computation. This relative cost, however, should be updated accordingly if the program is to be run on a new system.

With the relative cost model, we can compute the number of different computation operations an element/object participates in at each time step and then calculate its actual cost upon which the assignment of a proportional weighting to the element/object in the graph is based. This provides a basis on which further operations aimed at maintaining load balancing can take place.

5.2 Imbalance detection and re-balancing

Most load re-balancing schemes consist of three steps: imbalance detection, re-balancing criterion and domain re-decomposition.

The first step of load re-balancing schemes is to constantly monitor the level of load imbalance among the processors during the simulation. The workload of one processor at each time step can be measured, for instance, by summing up the relative costs of all objects in the processor. These costs are already computed during the interaction computation by following the procedure outlined earlier. Alternatively, the workload can also be accurately obtained by measuring its

actual runtime, but this is not trivial. Using either approach, the level of load imbalance can be defined as

$$B = \frac{W_{max} - \overline{W}}{\overline{W}} \tag{3}$$

where

$$W_{max} = \max_i W_i, \quad \overline{W} = \Big(\sum_{i=1}^{n_p} W_i\Big)/n_p$$

and W_i is the workload of the i-th processor; and n_p is the number of processors.

The next step is to decide when to perform a new graph partitioning to re-balance the load. Such a decision requires consideration of complicated tradeoffs between the cost of the graph partitioning, the quality of the new partitioning and the redistribution cost. A simple approach is to start the re-partitioning when the level of imbalance, B, exceeds a prescribed value, τ, i.e.

$$B > \tau \tag{4}$$

Similar to the size of buffer zone, τ may also need to be carefully tuned in order to achieve a better overall performance.

The final step is to perform a domain re-partitioning as described in the previous sections when condition (4) is satisfied.

6 Numerical Experiments

In this section, three examples are provided to illustrate the performance of the algorithms and implementation suggested. As the parallel performance of a domain decomposition for conventional finite elements is well established, the experiments will focus on the interaction resolution and force computations in both 2D and 3D cases.

In addition, in view of the fact that the efficiency of a parallelised program is often affected to some extent by complex hardware and software issues, the contribution to the overall performance from only the algorithmic perspective is therefore identified. More specifically, the following issues will be investigated: 1) the cost of the dynamic domain partitioning and repartitioning in terms of CPU time, and the quality of the partitioning; 2) the behaviour of two proposed graph representation models in terms of load balancing.

The parallelised finite/discrete element analysis program is tested on an SGI Origin 2000 with 8 processors. Due to the shared memory feature of the machine, interprocessor communication overhead plays a much less active role in the parallel performance. Each example is respectively tested with 1, 2, 4 and 6 processors.

The type of interaction considered between discrete elements is standard mechanic contact and the contact cases include node to edge, disk to disk and disk to edge contact in 2D, and sphere to sphere and sphere to 3-noded facet contact in 3D.

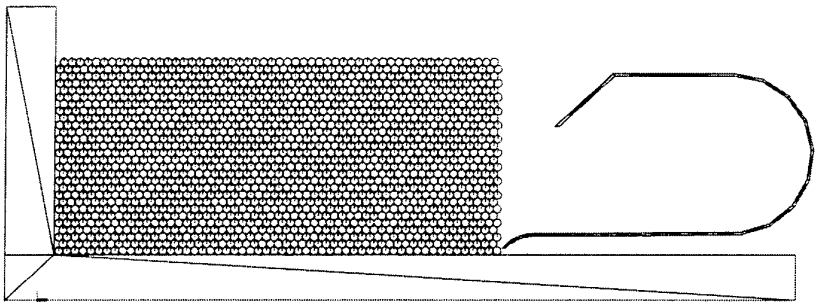


Fig. 1. Example 1 - Dragline bucket filling: Initial configuration

6.1 Example 1: 2D Dragline bucket filling

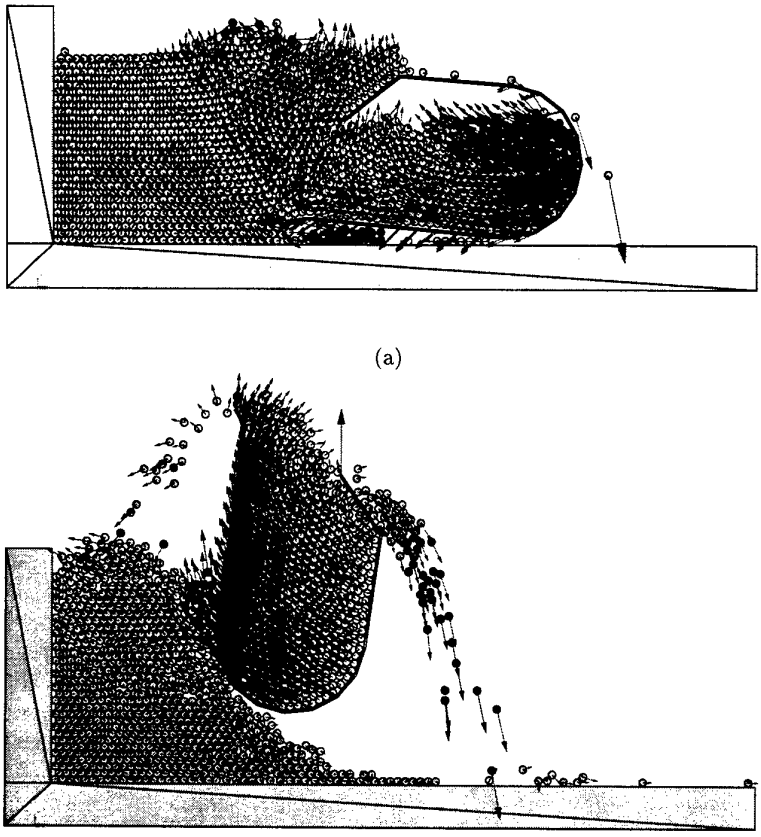
This example simulates a three-stage dragline bucket filling process, where the rock is modelled by discrete elements as disks, while the bucket is modelled as rigid, and the filling is simulated by dragging the bucket with a prescribed motion.

Figs. 1 and 2 respectively illustrate the initial configuration of the problem and two subsequent stages of the simulation. The discrete element model contains over 5000 disks with a radius of 20 units. The size of buffer zone is chosen to be 1 unit, and the time step is set to be 2.6×10^{-6} sec that leads to a total number of 1.6 million time increments required to complete the simulation. With the current buffer zone size, the global (re-)search is performed at about every $30 \sim 50$ steps and takes about 12.9% of the total CPU time in the sequential case.

For the dynamic domain partitioner employed, the CPU time consumed is 0.7% of the total time in the sequential case and 1.2% with 6 processors. These indicate that the re-partitioning algorithm in ParMETIS is efficient. It is also found that the partitioner produces partitioning with up to 3.5% load imbalance in terms of the weighted sub-total during the simulation.

Fig. 3 demonstrates the necessarity of employing a dynamic re-partitioning scheme, in which, the sub-domain distributions obtained by the re-partitioning at two stages are compared with those produced by a completely new partitioning at each occasion. It confirms that a series of consistent partitions that minimise the redistribution cost will not be achieved unless a proper re-partitioning algorithm is adopted in the dynamic domain decomposition. Note that, for better viewing, a much larger disk radius is used in Fig. 3.

The CPU time of each processor for the contact resolution and force computations using the *a priori* graph model at the first 500k time increments is shown in Fig. 4a with various number of processors, where it is clearly illustrated that severe load imbalance occurs. This can be explained, for instance in the 2-processor case, by the fact that although the domain is well partitioned



(a)

(b)

Fig. 2. Example 1 - Dragline bucket filling: Configurations at two stages showing particle velocities: (a) t=2s; (b) t=3.3s

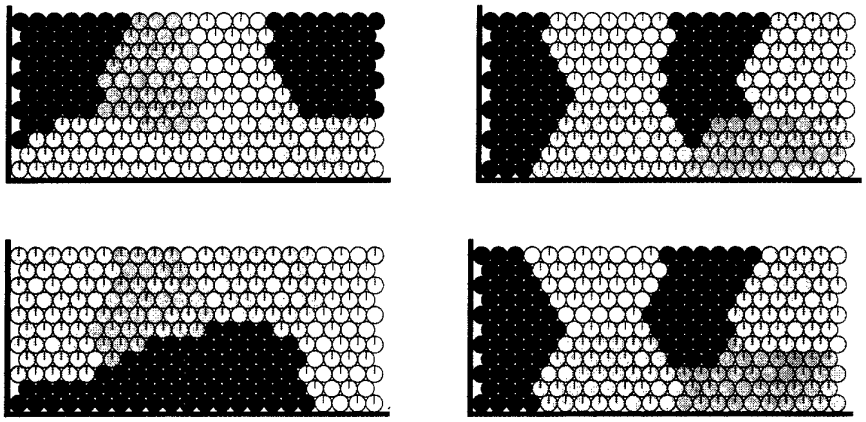
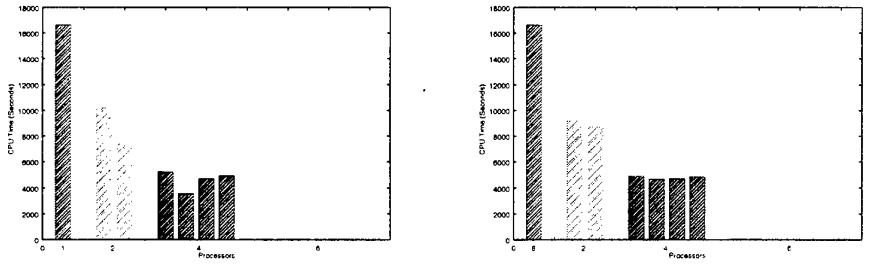


Fig. 3. Example 1 - Dragline bucket filling: Domain partitions at two different time instants: complete partitioning (left column) and re-partitioning (right column)



(a) (b)

Fig. 4. Example 1 - Dragline bucket filling: CPU time of each processor: (a) the *a priori* model; (b) the *a posteriori* model

Table 1. Speedup obtained by two graph models for Example 1

Model	2 processors	4 processors	6 processors
a priori model	1.63	3.20	4.41
a posteriori model	1.86	3.55	5.01

according to the potential contact lists, the actual computation cost on the second sub-domain is much less because the disks in this region are more scattered i.e. more false contact pairs are included in the corresponding lists.

A much better load balancing situation, depicted in Fig. 4b, has been achieved by the *a posteriori* graph model together with the proposed load re-balancing scheme. Table 1 also presents the overall speedup obtained by these two models with different number of processors.

6.2 Example 2: 3D hopper filling

The second example performs simulations of a 3D ore hopper filling process. The ore particles are represented by discrete elements as spheres and the hopper and the wall are assumed to be rigid. The particles are initially regularly packed at the top of the hopper and then are allowed to fall under the action of gravity. The configuration of the problem at an intermediate stage of the simulation is illustrated in Fig. 5.

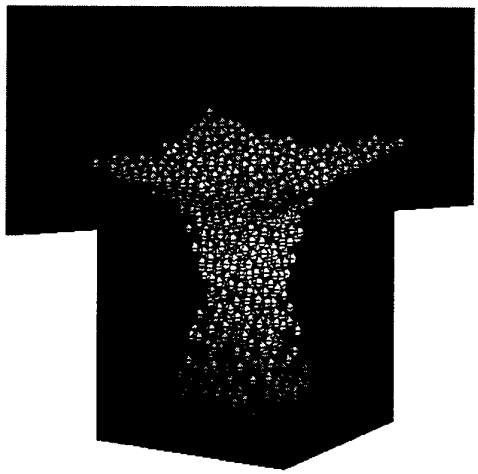


Fig. 5. Example 2 - 3D Hopper filling: Configuration at t=0.75s

Table 2. Speedup obtained by two graph models for Example 2

Model	2 processors	4 processors	6 processors
a priori model	1.86	3.55	4.72
a posteriori model	1.88	3.65	5.23

The radius of the spheres is 0.25 units and the buffer zone is set to be 0.0075 units. The global search is conducted less frequently at the beginning and end of the simulation due to a relatively small velocity of motion.

A similar quality of the partitioning as in the previous example is observed in this example. Table 2 shows the speedup achieved by both graph models with various number of processors. It appears that the *a priori* graph model exhibits a similar performance as the *a posteriori* model for the cases of 2 and 4 processors, but shows a performance degradation in the 6-processor case. The reason for this is because of the symmetry in both x- and y-directions in the problem, the domain decomposer produces a well balanced partitioning, in terms of actual computation cost, in the 2 and 4 processor cases, but fails to achieve the same quality of partitioning in the case of 6 processors. Also note that since the computational cost associated with each contact pair in 3D is more expensive than that in 2D, a slightly better overall parallel performance is achieved in this example.

6.3 Example 3: Axisymmetric layered ceramic target impact

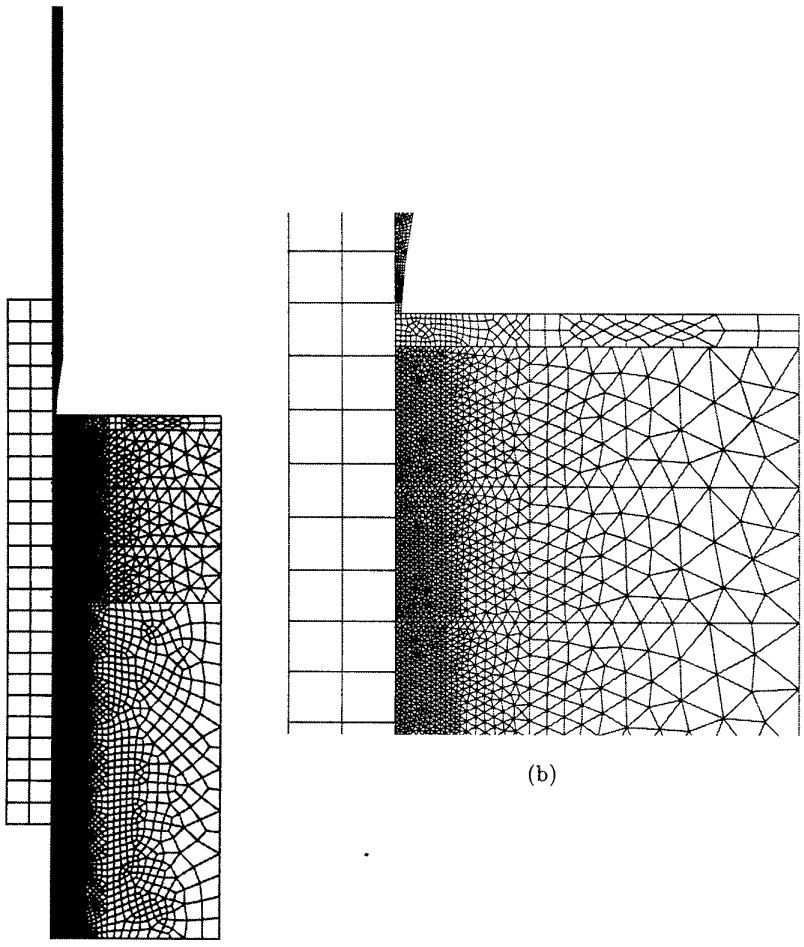
This example consists of a tungsten long rod impacting a composite target comprising an RHA backing block, three ceramic tiles of approximate thickness 25mm, with 6mm RHA cover plate. The ceramic tiles are unconfined. The computational model is axisymmetric with the tile radius set as 74mm. Four noded quadrilateral elements are used to represent the metal components and three noded triangular elements are used for the ceramic. Each component is initially set up as an individual discrete body with contact conditions between the bodies modelled using Coulomb friction. The centreline is modelled using a contact surface as a shield to prevent any object crossing the symmetry axis. The initial finite element discretisation is shown in Fig. 6.

The tungsten is modelled using an Armstrong-Zerilli AZUBCC model whilst the RHA is modelled using an AZMBCC model. Topological changes via erosion due to plastic strain is employed for both materials. The ceramic is treated as a brittle fracturing material and is modelled using the rotating crack model.

Two impact velocities, 1325m/s and 1800m/s, are considered.

The development of damage in the ceramic with increasing penetration at different stages is shown in Figs. 7a -7d. The configuration at $t = 200\mu$s is also depicted in Fig. 8.

Similar to the previous examples, the *a posteriori* graph model for discrete objects achieves better performance, which is demonstrated in Table 3.

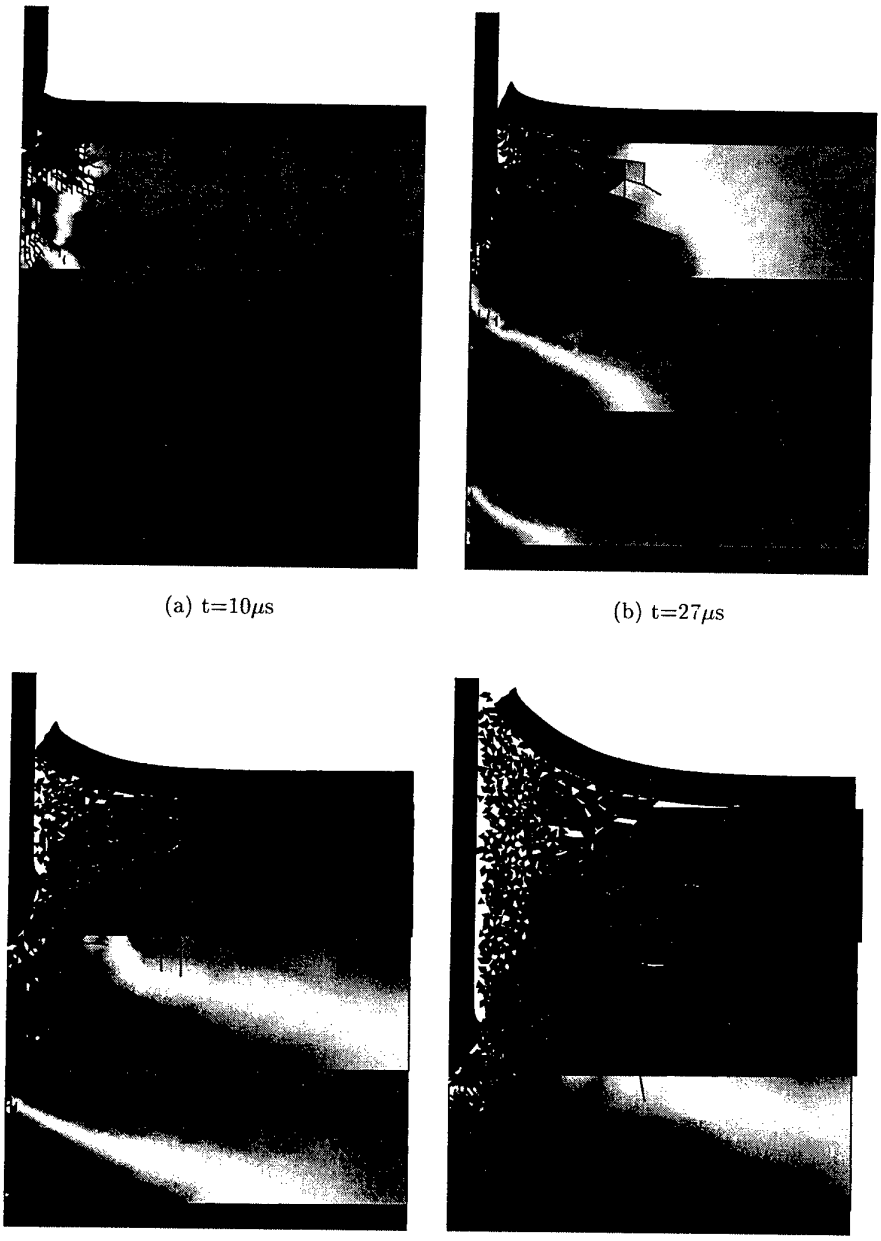


(a) (b)

Fig. 6. Example 3 - Ceramic target impact: (a) initial mesh; (b) zoomed mesh.

Table 3. Speedup obtained by two graph models for Example 3

Model	2 processors	4 processors	6 processors
a priori model	1.82	3.52	4.60
a posteriori model	1.86	3.63	5.33



(a) t=10μs (b) t=27μs

(c) t=40μs (d) t=72.5μs

Fig. 7. Example 3 - Ceramic target impact: Progressive damage indicating regions with radial fractures

Fig. 8. Example 3 - Ceramic target impact: The configuration at t= 200μs

7 Concluding Remarks

The algorithmic aspects of a parallel implementation strategy for a combined finite and discrete element approach are presented in this work. The main features of the implementation include: 1) a dynamic domain decomposition is applied independently to both the finite element computation, the contact detection and discrete element computation; 2) different methodologies can be employed in the global search and the interaction computations; 3) a dynamic graph repartitioning is used for the successive decomposition of the moving configuration; 4) two graph models are proposed for the representation of the relationship between the discrete objects; 5) load imbalance can be monitored and re-balanced by the proposed scheme.

By means of numerical experiment, the performance of the proposed algorithms is assessed. It is demonstrated that the dynamic domain decomposition using the second graph model with dynamic re-partitioning, load imbalance detection and re-balancing scheme can achieve a high performance in applications involving both finite and discrete elements. It is worth mentioning that the strategy suggested can also be applied to other areas such as smooth particle hydrodynamics (SPH), meshless methods, and molecular dynamics.

References

1. Owen, D. R. J., Feng, Y. T., Han, K., and Perić, D.: Dynamic domain decomposition and load balancing in parallel simulation of finite/discrete elements. In *European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS 2000)*, Barcelona, Spain, 11-14 September 2000.
2. Brown, K., Attaway, S., Plimpton, S., and Hendrickson, B.: Parallel strategies for crash and impact simulations. *Comp. Meth. Appl. Mech. Engng.*, 184:375–390, 2000.
3. Hendrickson, B., and Devine, K.: Dynamic load balancing in computational mechanics. *Comp. Meth. Appl. Mech. Engng.*, 184:485–500, 2000.
4. Yu, J.: *A Contact Interaction Framework for Numerical Simulation of Multi-Body problems and Aspects of Damage and Fracture for Brittle Materials.* Ph.D. Thesis, University of Wales Swansea, 1999.
5. Feng, Y. T., and Owen, D. R. J.: Argumented digital tree search algorithm in contact detection. 2000 (to appear).
6. Han, K., Perić, D., Crook, A.J.L., and Owen, D.R.J.: A combined finite/discrete element simulation of shot peening process. part I: Studies on 2D interaction laws. *Engineering Computations*, 2000 (in press).
7. Han, K., Perić, D., Owen, D.R.J., and Yu, J.: A combined finite/discrete element simulation of shot peening process. part II: 3D interaction laws. *Engineering Computations*, 2000 (in press).
8. Karypis, G., and Kumar, V.: METIS 4.0: Unstructured graph partitioning and sparse matrix ordering system. *Technical Report, Department of Computer Science, University of Minnesota*, 1998.
9. Macedo, J.: *An Integrated Approach to the Parallel Processing of Explicit Finite/Discrete Element Problems.* Ph.D. Thesis, University of Wales Swansea, 1997.

Dynamic Multi-Repartitioning for Parallel Structural Analysis Simulations

Achim Basermann[1], Jochen Fingberg[1], Guy Lonsdale[1], Jan Clinckemaillie[2], Jean Marc Gratien[2], Guillaume Thierry[2], and Richard Ducloux[3]

[1] C&C Research Laboratories, NEC Europe Ltd.,
Rathausallee 10, D-53757 Sankt Augustin, Germany
Tel.: +49/(0)2241/9252-0, Fax: +49/(0)2241/9252-99
{basermann, fingberg, lonsdale}@ccrl-nece.technopark.gmd.de
http://www.ccrl-nece.technopark.gmd.de/

[2] Pam System International, Rue Saarinen 20, F-94578 Rungis SILIC 270, France
Tel.: +33-1 49 78 28 20, Fax: +33-1 46 87 72 02
{jc, jmg, gt}@esi.fr
http://www.esi.fr

[3] Transvalor, Les Espaces Delta BP 037, 06901 Sophia Antipolis, France
Tel.: +33 (0) 493 95 52 24, Fax: +33 (0) 493 95 52 84
100604.2665@compuserve.com
http://www.transvalor.com

Abstract. The DRAMA project was initiated to support the take-up of large scale parallel simulation in industry by dealing with the main problem which restricts the effective use of message passing simulation codes — the inability to perform dynamic load balancing. The central product of the project is a library comprising a variety of tools for dynamic repartitioning of unstructured Finite Element or other mesh-oriented applications. The input to the DRAMA library is the computational mesh, and corresponding costs, partitioned into sub-domains. The core library functions then perform a parallel computation of a mesh re-allocation that will re-balance the costs based on the DRAMA cost model. This cost model allows a general approach to load identification, modelling and imbalance minimisation. We present results for the crash analysis code PAM-CRASH which show the necessity for multi-phase/multi-constraint repartitioning components. Moreover, we demonstrate how DRAMA handles imbalance due to adaptive refinement in the case of the stamping simulation code PAM-STAMP and imbalance due to adaptive remeshing in the case of the forging simulation code FORGE3.

Topics: crash and structural analysis; parallel and distributed algorithms.

1 Introduction

The DRAMA project [1] was initiated to support the take-up of large scale parallel simulation in industry by dealing with the main problem which restricts the effective use of message passing simulation codes — the inability to perform dynamic load balancing. The central product of the project is a library

comprising a variety of tools for dynamic repartitioning of unstructured Finite Element (FE) or other mesh-oriented applications. The input to the DRAMA library is the computational mesh and corresponding costs, partitioned into sub-domains. The core library functions then perform a parallel computation of a mesh re-allocation that will re-balance the costs based on the DRAMA cost model. In the following sections, we discuss the basic features of this cost model which allows a general approach to load identification, modelling and imbalance minimisation. Furthermore, results are presented for the industrial crash analysis code PAM-CRASH [5] which show the necessity for multi-phase/multi-constraint repartitioning components [9]. These new partitioning methods allow load balancing for several computational phases — e.g., stress-strain analysis and contact treatment in PAM-CRASH — that are separated by synchronisation points (see also Sect. 4). Moreover, we demonstrate how DRAMA handles imbalance due to adaptive refinement in the case of the stamping simulation code PAM-STAMP [7] and imbalance due to adaptive remeshing in the case of the forging simulation code FORGE3 [6].

2 DRAMA Cost Model

The DRAMA cost model [10] explicitly considers calculation costs w_i per sub-domain i and communication costs $c_{i,j}$ between sub-domains i and j of the parallel application code. For the load-balancing re-partitioning algorithms, it results in an objective cost function F. The model provides a measure of the quality of the current distribution and is used for the prediction of the effect on the computation of moving some parts of the mesh to other sub-domains.

The essential feature is that the cost model is mesh-based, so that it is able to take account of the various workload contributions and communication dependencies that can occur in finite element applications. Being mesh-based, the DRAMA cost model includes both per element and per node computational costs and element-element, node-node, and element-node data dependencies for communication. The DRAMA mesh consists of nodal coordinates and of a list of nodes per element which is a native data structure (element connectivity) in most finite element applications.

In addition to data dependencies between neighbouring elements and nodes in the mesh, dependencies between arbitrary parts of the mesh can occur. For the PAM-CRASH code [5], such data dependencies originate within the contact-impact algorithms when the penetration of mesh segments by non-connected nodes is detected and corrected. The DRAMA cost model allows the construction of *virtual elements* [3, 10] which represent the occurring costs of such dependencies (see also Sect. 4). A virtual element is included in the DRAMA mesh in the same way as a real element: as an additional connectivity list of its constituent nodes.

Types u identify calculation cost parameters per element or per node that refer to different kinds of elements, different material properties, or generally different algorithmic parts of the application code requiring different kinds of

operations. Communication cost parameters per element-element, node-node, and element-node connection depend on the amount of data that potentially have to be transferred for a link between two objects of type u_1 and u_2.

Different algorithmic parts in parallel application codes that are separated by explicit synchronisation points are defined as *phases* within the DRAMA cost model. DRAMA evaluates the costs per phase *iphase*. The PAM-CRASH code, for example, can be considered to consist of essentially two sections; stress-strain computations including time integration (FE phase) and contact treatment (contact phase) with a global synchronisation in between and also at the end of each computing cycle.

Cost parameter determination requires application code instrumentation. Numbers of operations per element/node of type u, $nop_i(u)$, can be specified by counting operations **or** by time measurements. The sum over all phases of total calculation times per phase and counting total numbers of computational operations allow the determination of calculation speeds s_i^{calc}. For communication, the number of bytes $noc(u_1, u_2)$ that have potentially to be transferred for a link between two objects of type u_1 and u_2 and communication speeds $s_{i,j}^{comm}$ have to be specified (latency is not considered). $s_{i,j}^{comm}$ essentially depends on the specific communication protocol. A suited communication model considering the message length has to be chosen. Moreover, a correspondence between types and phases must be given.

With these parameters, the DRAMA cost model has the following form.

$$F = \sum_{iphase} \max_i F_i^{iphase} \quad , \quad F_i^{iphase} = w_i^{iphase} + \sum_j c_{i,j}^{iphase}$$

$$w_i^{iphase} = \sum_u N_i(u) \frac{nop_i(u)}{s_i^{calc}} \quad , \quad c_{i,j}^{iphase} = \sum_{u_1 u_2} N_{i,j}(u_1, u_2) \frac{noc(u_1, u_2)}{s_{i,j}^{comm}}$$

$N_i(u)$ is the number of elements/nodes of type u and $nop_i(u)/s_i^{calc}$ is the computational cost of an object of this type. Since only the ratio is relevant both $nop_i(u)$ and s_i^{calc} may be specified as relative values if this makes instrumentation easier. $N_{i,j}(u_1, u_2)$ is the number of elements/nodes in a sub-domain boundary region, and $noc(u_1, u_2)/s_{i,j}^{comm}$ is the potential communication cost for a link between two objects of type u_1 and u_2.

3 DRAMA Library Interface

The interface between the application code and the library is designed around the DRAMA cost model and the instrumentation of the application code to specify current and future computational and communication costs [3]. Thus the application code has to provide DRAMA, per sub-domain, with the current mesh description, i.e., the element-node connectivity including the type information. The elements can be either real or virtual elements. The nodal coordinates are given in addition.

Moreover, the application code places the calculation and communication cost parameters per type at DRAMA's disposal as well as the correspondence between types and phases.

DRAMA returns the new partition in terms of a new numbering of local elements and nodes together with the relationships between old and new numbering systems and the coordinates of the new set of nodes local to a process. The relationships between old and new numbering systems support the application code in building send and receive lists.

4 Dynamic Load Balancing with DRAMA

The goal of any load balancing method is to improve the performance of applications which have computational requirements that vary with time. The DRAMA library is targeted primarily at mesh-based codes with one or more phases. It offers a multiplicity of algorithms allowing the different needs of a wide range of applications (Finite Element, Finite Volume, adaptive mesh refinement, contact detection) to be covered. The DRAMA library contains geometric (RCB), topological (graph) and local improvement (direct mesh migration) methods [2]. It enables the use of leading graph partitioning algorithms through internal interfaces to ParMetis and PJostle [8, 11, 12].

In comparison with the direct use of graph partitioners, DRAMA has the following advantages.

1. DRAMA's interface is mesh-based. Since an element-node connectivity list is an essential component of mesh-based application codes DRAMA can be easily integrated. Mesh to abstract graph conversion is performed within DRAMA.
2. Beside graph partitioners, DRAMA offers local improvement (migration) and geometric methods. Thus, DRAMA is more general.
3. DRAMA supports cost capturing and cost monitoring.
4. DRAMA supports the application code in building new mailing lists after the re-partitioning.
5. DRAMA allows different element/node type management.

Thus, DRAMA provides pre-defined solutions for most mesh-based codes.

Many applications consist of several phases separated by explicit or implicit global synchronisation points. This is a challenging problem that requires each phase to be balanced independently. Fig. 1 (left) illustrates the situtation for two processors and two phases. Both phases show distinct load imbalance. If both phases depend on each other, as for the stress-strain and contact phases in PAM-CRASH, where the computations refer to the same mesh in both phases, balancing the aggregate costs of both phases is of no use: the two phases have to be balanced separately. There are two approaches to this problem, one is to work with a separate division of objects for each phase [4], the other is to balance each phase on a common partition. The first strategy is advantageous if all computational sections (phases) of the code work on the entire model.

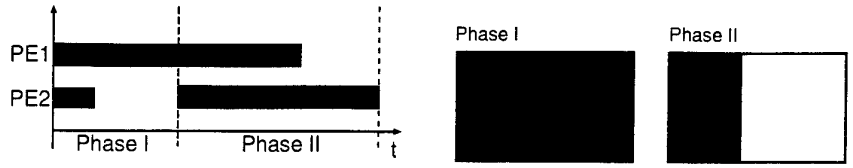


Fig. 1. Left: Load imbalance in two phases separated by two sychronisation points. Right: Operations in two phases on different parts of the same mesh.

It requires fast communication between the different decompositions in each computing cycle. If the code works on different parts of the model (mesh) in different phases it can be favourable to maintain a single mesh decomposition and save communication time. The latter situation is displayed in Fig. 1 (right). The first phase refers to the whole mesh, the second only to the left part of the mesh. For a frontal car crash simulation against a rigid wall with PAM-CRASH, stress-strain is computed for the whole mesh whereas contact detection and correction is mainly performed in the front part of the car model.

Here we follow the single mesh decomposition strategy because it is much easier to implement in existing applications. We show results for the FE phase and contact phase of PAM-CRASH exploiting the new multi-phase/multi-constraint options of PJostle and ParMetis [9, 12].

It should be pointed out that the possible gains that can be achieved with the existing contact-impact algorithmic implementation in PAM-CRASH are limited:

1. for small processor numbers, the contact-impact phase of the calculations has been optimised in recent years to the point where it is very much dominated by the already well-balanced stress-strain calculations;
2. the existence of a non-scaling computational section and (pseudo-) all-to-all communication, which cannot be handled by the DRAMA library's partitioners, produces a dominating cost for larger processor numbers, particularly when the repartitioning attempts to balance the contact phase across many processors.

The graph-partitioning is built upon a combined graph of elements and nodes [2] because a part of the computation is node-based and a part element-based. The basic objects during contact detection are pairs of nodes and segments of a surface, the segment being defined by four nodes. These objects are passed to the DRAMA library as virtual five-node elements in the DRAMA mesh format [3].

5 Evaluation of Multi-Repartitioning Techniques

To evaluate the performance of different repartitoning methods we compare results obtained with a test mesh of an AUDI car model which originate from a

Table 1. Re-partitioning methods.

method	partitioner
1	METIS_mCPartGraphkway, sequential
2	MJostle, sequential
3	MOC_PARMETIS_Partkway
4	MOC_PARMETIS_SR
5	MJostle, parallel
6	PARMETIS_RepartGDiffusion, single phase

PAM-CRASH simulation of a frontal impact with a rigid wall. The mesh data is stored after 10000 cycles from a total of around 80000 cycles. The model consists of 4-node shell and 2-node beam elements. The total load imbalance after 10000 cycles is 12.1% (load imbalance factors: 1.0002 for stress-strain, 10.642 for contact). The initial partition has been generated by the original PAM-CRASH single-phase partitioner at the start of the simulation. Only FE costs have been considered.

The load imbalance factors are defined as

$$\lambda^1 = \frac{\max_{i=0..p-1}(\ w_i^1)}{\overline{w_i^1}} \quad \text{(FE)}\ , \quad \lambda^2 = \frac{\max_{i=0..p-1}(\ w_i^2)}{\overline{w_i^2}} \quad \text{(contact)}\ ,$$

$$\lambda^1_{tot} = \frac{\max_{i=0..p-1}(\sum_j^{nphases}\ w_i^j)}{\overline{\sum_j^{nphases}\ w_i^j}}\ , \quad \lambda^2_{tot} = \frac{\sum_j^{nphases} \max_{i=0..p-1}\ (w_i^j)}{\overline{\sum_j^{nphases}\ w_i^j}}\ .$$

$\overline{x_i}$ denotes the mean value of all x_i, $i = 0..p-1$. λ^1_{tot} neglects synchronisation points, whereas λ^2_{tot}, the real load imbalance, considers them. The value 1 means optimal load balance.

For the graph representation of the AUDI mesh we use a combined graph [2] consisting of elements and nodes where the connections are only between elements and nodes. We consider the methods listed in Table 1 [8, 9, 11, 12]. Method 6 is a single phase partitioner and is added for comparison reasons, all other methods are multi-phase/multi-constraint algorithms. Methods 1 and 2 are sequential multi-partitioners, all other methods are parallel. *MOC_PARMETIS_SR* is a re-partitioner that should minimise load-imbalance and the difference between the current and the new partition. The latter was not investigated here but will be checked in detail in future tests.

Table 2 shows the distribution of shell elements, beam elements, nodes and contact pairs (CPs) per processor (PE) for the AUDI model with 16 sub-domains. Multi-repartitioner 1 is applied. *FE* and *CO* are the costs for the stress-strain phase and the contact phase. The cost for a beam element is about half the cost for a shell element whereas the cost for a contact pair is about one third of the cost for a shell element. These ratios are realistic for PAM-CRASH and were confirmed by timings within the application code. Therefore, the number of shell elements is multiplied by 6, the number of beams by 3, and the number

Table 2. Distribution of shell elements, beam elements, nodes and contact pairs (CPs) per sub-domain, AUDI, repartitioner 1.

PE	shell	beam	CP's	nodes	FE	CO
0	1735	6	67	1652	10428	134
1	1738	0	63	1715	10428	126
2	1737	3	67	1744	10431	134
3	1739	0	67	1661	10434	134
4	1739	0	64	1612	10434	128
5	1730	17	63	1622	10431	126
6	1735	6	67	1696	10428	134
7	1735	6	67	1732	10428	134
8	1725	29	67	1567	10437	134
9	1738	3	67	1719	10437	134
10	1739	0	67	1612	10434	134
11	1739	0	67	1702	10434	134
12	1730	17	66	1670	10431	132
13	1739	0	67	1586	10434	134
14	1739	0	67	1644	10434	134
15	1674	129	67	1710	10431	134
total	27711	216	1060	26644	166914	2120
mean					10432.1	132.5

of contact pairs by 2 to obtain the total costs per processor for the stress-strain phase and the contact phase.

Table 3 displays minimum, maximum, and mean total computational weights per phase of 16 sub-domains for the AUDI model with all partitioning methods considered. Cut edges as well as load imbalance factors per phase and total load imbalance factors are given in addition.

Table 3. Total computational weight per phase, cut edges, and imbalance, AUDI.

meth.	FE [min max]	CO [min max]	edge cut	imbalance [FE CO] tot.
1	[10428 10437]	[126 134]	2193	[1.000 1.011] 1.001
2	[9354 10536]	[132 134]	3116	[1.001 1.011] 1.010
3	[10095 10548]	[128 136]	2308	[1.011 1.026] 1.011
4	[10158 10632]	[126 136]	2682	[1.019 1.026] 1.019
5	[10380 10518]	[130 136]	2641	[1.008 1.026] 1.008
6	[9075 11070]	[0 1128]	4154	[1.061 8.513] 1.155

The results in Table 3 demonstrate that all multi-partitioning methods are able to achieve nearly perfect load balance. The multi-partitioners reach a load imbalance of around 1%, and less, whereas the single phase partitioner ends up with a load imbalance of about 16%.

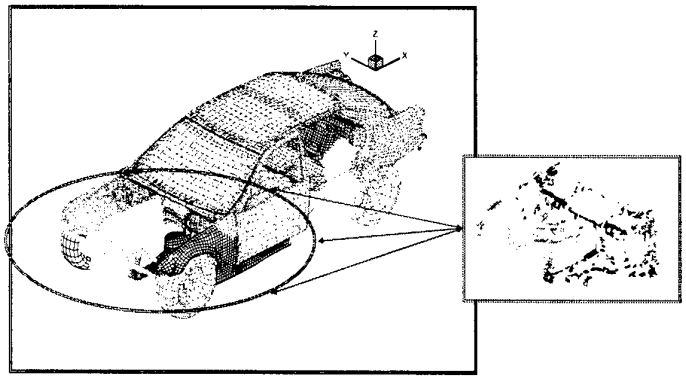


Fig. 2. Multi-constraint Metis (method 3) for a BMW PAM-CRASH model: whole partition and partition of the contact area.

A graphical representation of the handling of the contact phase is given in Fig. 2 for a BMW PAM-CRASH model with 52216 shell elements and 368 beam elements. Multi-constraint Metis (method 3) was applied to partition the BMW into 8 sub-domains. On the left, the whole partition is displayed, whereas the part of the partition where contact occurs is shown on the right. For a frontal crash, contact-impact mainly takes place in the front part of the car. Note that, due to multi-partitioning, all 8 subdomains share the contact area.

Fig. 3 shows contact cost comparisons for a simulation with the BMW model using repartitioning after every 10000 steps, on 8 processors of an NEC Cenju-4 system (R10000 processors, 400 Mflops, 200 MB/s maximum network transfer rate). The elapsed times for contact calculations per step on each of the 8 processors are displayed. Costs in the stress-strain phase are balanced with and without repartitioning.

With DRAMA multi-repartitioning, load imbalance in the contact phase is markedly decreased (red curves, lower bracket). Data redistribution costs 77 s in total. Despite the caveat that major computational gains cannot be expected for PAM-CRASH with the currently existing implementations of the contact-impact algorithms (see 4), it is nevertheless worthwhile to demonstrate that overall computational efficiency is indeed increased: The initial total simulation time of 17430 s was reduced to 17040 s. With better scaling contact algorithms as in [4] and high processor numbers, the total effect of DRAMA multi-repartitioning would be much more distinct.

For the stamping simulation code PAM-STAMP [7], both contact treatment and adaptive meshing are sources of load imbalance. Since the typical contact-

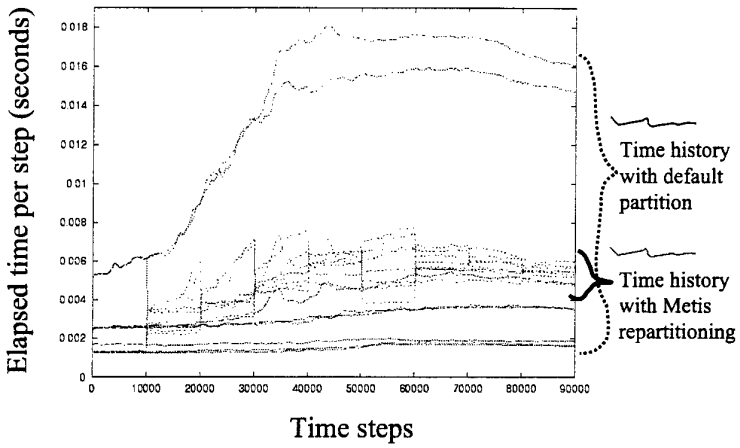


Fig. 3. Multi-constraint Metis (method 3) for a BMW PAM-CRASH model: contact cost comparisons for repartitioning after every 10000 steps (8 processors, Cenju-4).

impact algorithms used within PAM-STAMP applications display better scaling properties than those within PAM-CRASH and the contact phase calculations are fairly well distributed across the elements, it is adaptive refinement that usually makes load imbalance markedly higher than in PAM-CRASH simulations.

The DRAMA library can take account of the new costs due to adaptive meshing either by the new number of elements and nodes per sub-domain or by changed cost parameters. In the former case, the refined mesh with old cost parameters has to be given to the library. The latter case makes sense if the adaptive refinement is performed in a strictly hierarchical way. DRAMA partitions the coarsest mesh with cost parameters that are multiplied according to the refinement of the elements. This strategy decreases DRAMA's memory requirements and increases the speed of the repartitioning. For PAM-STAMP, we follow this strategy.

Unfortunately, the full impact of dynamic load-balancing for PAM-STAMP cannot be demonstrated with the parallel prototype integrated with the DRAMA library during the lifetime of the project. The prototype code includes (physically unnecessary) nodal calculations for the time integration of nodes corresponding to the elements modelling the tools: the *null shells* which are treated as rigid bodies and used to model the motion of the punch, die and blankholder. These calculations are removed in more recent versions of the standard PAM-STAMP code. The impact on the prototype PAM-STAMP with DRAMA was two-fold:

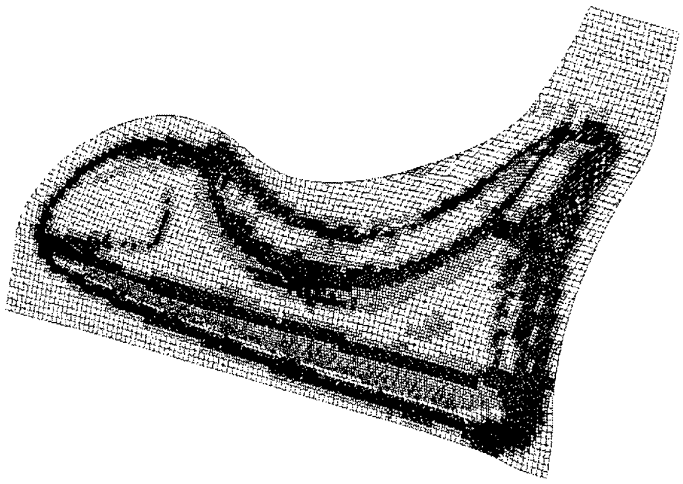


Fig. 4. Fender PAM-STAMP model (Courtesy General Motors): final mesh.

the normally dominant FE stress-strain calculations (upon which the design for the use of the DRAMA library in the PAM-codes is based) become subsidiary to the null shell nodal costs; nodal costs remain imbalanced since restrictions in the data migration and re-partitioning approach within the PAM-codes mean that the DRAMA nodal partitions cannot be used. In addition, the parallel (message-passing) version with adaptive meshing is a recently developed feature and currently subject to robustness problems when more than one level of refinement is introduced (thus additionally limiting the impact of re-partitioning).

Although other options for handling the prototype code are available within the DRAMA library (an example being node-by-node cost modelling), the fact that the future versions of parallel PAM-STAMP will resolve the above issues meant that further investigations with the prototype code was not deemed appropriate, nor was it feasible within the project time-frames.

The results presented in the following are taken from full PAM-STAMP simulations with an industrial benchmark model — the General Motors fender (final mesh) illustrated in Fig. 4 — but with cost analysis only for the FE phase, which includes the use of adaptive mesh refinement.

The performance presented in Fig. 5 is for the fender model with 5180 initial elements on the blank, using 200 mesh refinement steps (based on a 1° angle criteria) and a maximum of one refinement level. DRAMA graph repartitioning was performed at intervals of 4000 computational cycles.

Fig. 5 demonstrates the evolution of imbalance using the ratio of slowest (R_{max}) to average (R_{avg}) times per process spent in the FE routines, R_{bal}:

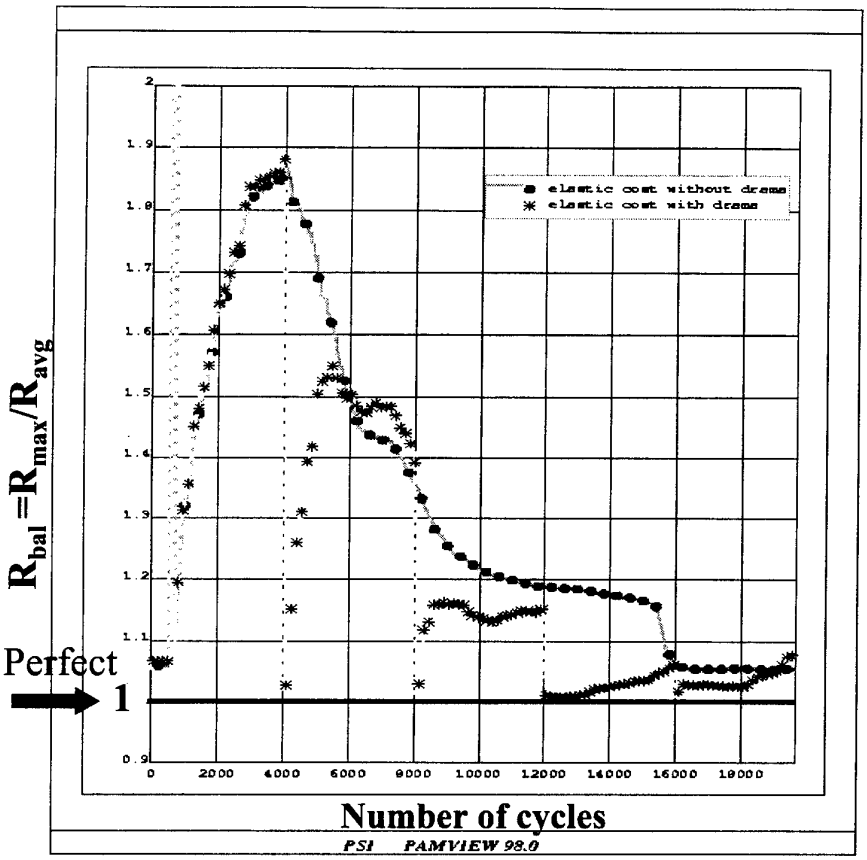


Fig. 5. Fender PAM-STAMP model (Courtesy General Motors): comparison of stress-strain imbalance development (R_{bal} per cycle) with/without DRAMA (repartitioning every 4000 cycles, 8 processors, Cenju-4).

$R_{bal} = R_{max}/R_{avg}$. The comparison is made between a standard run without DRAMA repartitioning (dark curve with bullets) and the run with DRAMA (light curve with stars) — both using 8 compute nodes on the NEC Cenju-4.

Two issues are apparent from Fig. 5: First, the mesh refinement generates high imbalance in the early phases of the simulation that decreases as the simulation proceeds. Second, with values of R_{bal} close to 1.03 at the repartitioning points (the multiples of 4000 cycles), DRAMA is very effectively balancing the FE costs. The reason for decreasing imbalance over the length of the simulation

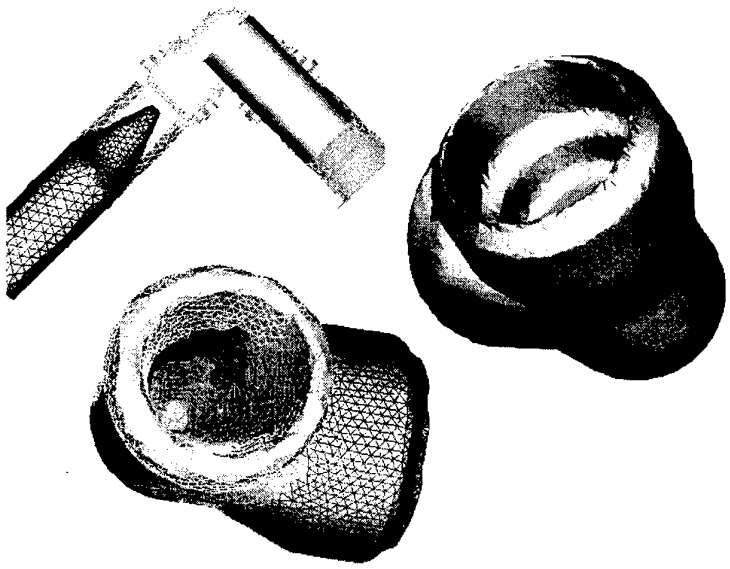


Fig. 6. Pipe connector FORGE3 model (Courtesy Transvalor): final remeshed partition.

is that the final mesh includes refinement which is fairly equally distributed over the model. What is also clear is that one would in practice use more frequent DRAMA repartitioning.

In the following, full code results are presented for the forging simulation code FORGE3. Fig. 6 shows the final remeshed partition for a pipe connector test case.

The new parallel remeshing strategy of FORGE3 including DRAMA is as follows: provide that each sub-domain has been remeshed independently at fixed interfaces in a previous time step. Per subsequent step, the nodes at and around sub-domain boundaries are marked which are to become internal nodes after repartitioning. The criterion is if remeshing around these nodes improves the mesh quality. After this first DRAMA repartitioning, remeshing per sub-domain at fixed interfaces is performed. Finally, DRAMA is called a second time to achieve good load balance. This strategy is possible due to the speed of repartitioning and data migration, which takes only a few seconds while one time step or a remeshing takes several minutes in the pipe connector case.

Fig.7 displays speedups on the pipe connector problem using DRAMA, employing both the mesh migration module and ParMetis single phase graph partitioning, on the LAMP PC cluster at NEC (dual processor Pentium-pro PCs with a Myrinet switch).

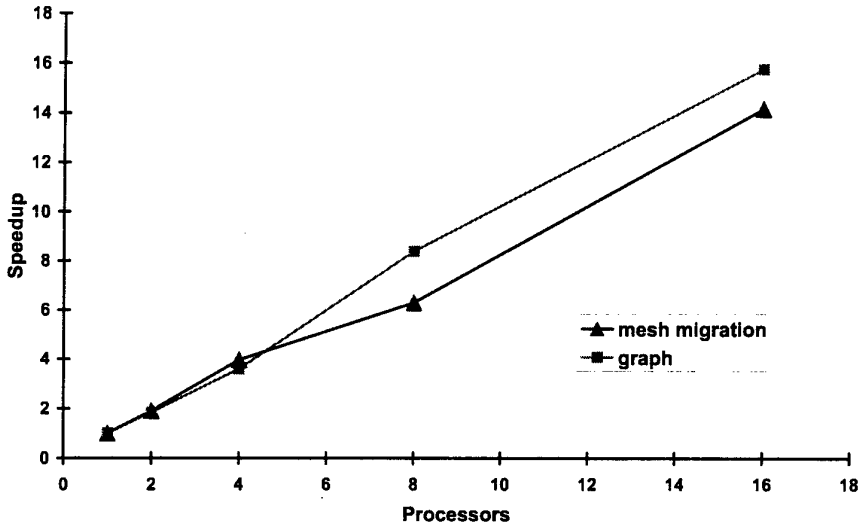


Fig. 7. Pipe connector FORGE3 model (Courtesy Transvalor): speedups with DRAMA mesh migration and graph repartitioning (PC cluster LAMP).

For a conrod, a tap fitting, and the pipe connector model, Table 4 shows the relative performance on 4 processors with respect to the parallel FORGE3 code without DRAMA — as can be seen, significant computational savings are made.

Table 4. Comparisons of total elapsed times for three test cases with the parallel FORGE3 code: original and new DRAMA versions.

Test cases	Conrod	Pipe connector	Tap fitting
Original version	5820 s	43500 s	54500 s
DRAMA version	4620 s	34040 s	36066 s

6 Conclusions

As demonstrated by tests with a mesh and a real simulation run of the industrial code PAM-CRASH, multi-partitioning methods achieve nearly perfect load balance whereas single phase partitioners fail to improve the initial imbalance. The new mesh distributions balance both computational phases simultaneously with

small remaining imbalance. Full code results of the stamping simulation code PAM-STAMP and the forging simulation code FORGE3 showed that repartitioning with the DRAMA library can handle imbalance due to adaptive refinement as well as adaptive remeshing and thus results in significant computation time savings.

Acknowledgements

First, the authors would like to thank all our colleagues from the DRAMA project. The support of the European Commission through the ESPRIT IV (Long Term Research) Programme is gratefully acknowledged.

References

1. The DRAMA Consortium, Project Homepage: `http://www.ccrl-nece.technopark.gmd.de/DRAMA`
2. The DRAMA Consortium: Report on Re-Partitioning Algorithms and the DRAMA Library. DRAMA Project Deliverable D1.3a [1] (1998)
3. The DRAMA Consortium: Updated Library Interface Definition. DRAMA Project Deliverable D1.2b [1] (1999)
4. Attaway, S.A., Barragy, E.J., Brown, K.H., Gardner, D.R., Hendrickson, B.A., Plimpton, S.J.: Transient Solid Dynamics Simulations on the Sandia/Intel Teraflop Computer. Supercomputing '97, Technical Paper (1997)
5. Clinckemaillie, J., Elsner, B., Lonsdale, G., Meliciani, S., Vlachoutsis, S., de Bruyne, F., Holzner, M.: Performance issues of the parallel PAM-CRASH code. Int. J. Supercomputer Applications and High Performance Computing, **11 (1)** (1997) 3–11
6. Coupez, T.: Parallel Adaptive Remeshing in 3D Moving Mesh Finite Element. Numerical Grid Generation in Comp. Field Simulation, B.K. Soni et al., eds., Mississippi University, **1** (1996) 783–792
7. Haug, E., Lefebvre, D., Dammak, Y., Taupin, L., de Luca, P., El Khaldi, F., Mehrez, F., Culière, P., Heath, A., Pickett, A.K., Queckbörner, T.: Numerical Simulation of Industrial Sheet Forming Processes with PAM-STAMP. 4th European Cars/Trucks Symposium , Schliersee (1995)
8. Karypis, G., Kumar, V.: ParMetis: Parallel graph partitioning and sparse matrix ordering library. University of Minneapolis, tech. rep. #97-060 (1997)
9. Karypis, G., Kumar, V.: Multilevel Algorithms for Multi-Constraint Graph Partitioning. University of Minneapolis, tech. rep. #98-019 (1998)
10. Maerten, B., Roose, D., Basermann, A., Fingberg, J., Lonsdale, G.: DRAMA: A library for parallel dynamic load balancing of finite element applications. Proceedings of the *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, CD-ROM (1999)
11. Walshaw, C., Cross, M., Everett, M.: Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. J. Par. Dist. Comput., **47 (2)** (1997) 102–108
12. Walshaw, C., Cross, M., McManus, K.: Multiphase Mesh Partitioning. Univ. Greenwich, London SE10 9LS, UK, tech. rep. 99/IM/51 (1999)

PARALLEL EDGE-BASED FINITE ELEMENT TECHNIQUES FOR NONLINEAR SOLID MECHANICS

Marcos A.D. Martins, José L.D. Alves and Alvaro L.G.A. Coutinho(*)

Center for Parallel Computing
COPPE/Federal University of Rio de Janeiro
PO Box 68506, Rio de Janeiro, RJ 21945-970, Brazil
E-mail: marcos, jalves, alvaro@coc.ufrj.br

Abstract. Parallel edge-based data structures are used to improve computational efficiency of Inexact Newton methods for solving finite element nonlinear solid mechanics problems on unstructured meshes composed by tetrahedra or hexaedra. We found that for tetrahedral meshes, the use of edge-based data structures reduce memory requirements to hold the stiffness matrix by a factor of 7, and the number of floating point operations to compute the matrix-vector product needed in the iterative driver of the Inexact Newton method by a factor of 5. For hexahedral meshes the reduction factors are respectively 2 and 3.

1. Introduction

Predicting the three-dimensional response of large-scale solid mechanics problems undergoing plastic deformations is of fundamental importance in several science and engineering applications. Particularly in the Oil and Gas Industry, solid mechanics is being used to improve the understanding of complex geologic problems, thus helping to reduce risks and operational costs in exploration and production activities (Arguello, 1998).

Traditional finite element technology for nonlinear quasi-static problems involves the repeated solution of systems of sparse linear equations by a direct solution method, that is, some variant of Gauss elimination. The updating and factorization of the sparse global stiffness matrix can result in extremely large storage requirements and a very large number of floating point operations.

Explicit quasi-static nonlinear finite element technologies (Biffle, 1993), on the other hand, may be employed, reducing considerably memory requirements. Although robust and straightforward to implement, explicit schemes, based on dynamic relaxation or nonlinear conjugate gradients may suffer from low convergence rates.

In this paper we employ an Inexact Newton method (Kelley, 1995), to solve large-scale three-dimensional incremental elastic-plastic finite element problems found in geologic applications. In the Inexact Newton Method, at each nonlinear iteration, a linear system of finite element equations is approximately solved by the preconditioned conjugate gradient method. The computational kernels of the Inexact

Newton Methods, besides residual evaluations and stiffness matrix updatings, are the same of the iterative driver, that is, matrix-vector products and preconditioning. Matrix-vector products can be optimized using edge-based data structures, typical of computational fluid dynamics applications (Peraire, 1992, Luo, 1994). For unstructured grids composed by tetrahedra we found that the edge-based data structures reduce memory requirements to hold the stiffness matrix by a factor of 7. Further, the number of floating point operations to compute the matrix-vector product is also reduced by a factor of 5. For grids composed by trilinear hexaedra memory is reduced by a factor of 2, while the number of floating point operations decreases by a factor of 3.

The remainder of this work is organized as follows. In the next section we briefly review the governing nonlinear finite element equations and the Inexact Newton methods. Section 3 describes the edge-based data structures for solid mechanics. Section 4 shows the numerical examples. The paper ends with a summary of the main conclusions.

2. Incremental Equilibrium Equations and the Inexact Newton Method

The governing equations for the quasi-static deformation of a body occupying a volume Ω is,

$$\frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i = 0 \qquad in\ \Omega \tag{1}$$

where σ_{ij} is the Cauchy stress tensor, x_j is the position vector, ρ is the weight per unit volume and b_i is a specified body force vector. Equation (1) is subjected to the kinematic and traction boundary conditions,

$$u_i(x,t) = \overline{u}_i(x,t) \ \ in \ \ \Gamma_u \ ; \ \sigma_{ij} n_j = h_i(x,t) \ \ in \ \ \Gamma_h \tag{2}$$

where Γ_u represents the portion of the boundary where displacements are prescribed ($\overline{u}_i$) and Γ_h represents the portion of the boundary on which tractions are specified (h_i). The boundary of the body is given by $\Gamma = \Gamma_u \cup \Gamma_h$, and t represents a pseudo-time (or increment). Discretizing the above equations by a displacement-based finite element method we arrive to the discrete equilibrium equation,

$$F_{\text{int}} + F_{ext} = 0 \tag{3}$$

where F_{int} is the internal force vector and F_{ext} is the external force vector, accounting for applied forces and boundary conditions. Assuming that external forces are applied incrementally and restricting ourselves to material nonlinearities only, we arrive, after a standard linearization procedure, to the nonlinear finite element system of equations to be solved at each load increment,

$$K_T \Delta u = R \tag{4}$$

where K_T is the tangent stiffness matrix, function of the current displacements, Δu is the displacement increments vector and R is the unbalanced residual vector, that is, the difference between internal and external forces.

Remark. We consider here perfect-plastic materials described by Mohr-Coulomb yield criterion. Stress updating is performed by an explicit, Euler-forward subincremental technique (Crisfield, 1990).

Some form of Newton method generally solves the nonlinear finite element system of equations given by Eq. (4), where the tangent stiffness matrix has to be updated and factorized at every nonlinear iteration. This approach is known as Tangent Stiffness (TS) method. The burden of repeated stiffness matrix updatings and factorizations is alleviated, at the expense of more iterations, by: keeping the tangent stiffness matrix frozen within a load increment; iterating with the elastic stiffness matrix, known as the Initial Stress (IS) method. For solving large-scale problems, particularly in 3D, it is more efficient to solve approximately the linearized problems by suitable inner iterative methods, such as preconditioned conjugate gradients. This inner-outer scheme is known as the Inexact Newton method, and the convergence properties of its variants, the Inexact Initial Stress (IIS) and Inexact Tangent Stiffness (ITS) methods have been analyzed for von Mises materials by Blaheta and Axelsson (1997). We introduce here a further enhancement in IIS and ITS methods, by choosing adaptively the tolerance for the inner iterative equation solver according to the algorithm suggested by Kelley (1995). We also include in our nonlinear solution scheme a backtracking strategy to increase the robustness of the overall nonlinear solution algorithm.

3. Edge-Based Data Structures

Edge-based finite element data structures have been introduced for explicit computations of compressible flow in unstructured grids composed by triangles and tetrahedra (Peraire, 1992, Luo, 1994). It was observed in these works that residual computations with edge-based data structures were faster and required less memory than standard element-based residual evaluations. We have studied edge-based data structures for the implicit finite element solution of potential flow problems (Martins et al, 1997). Following these developments, for solid mechanics problems, we may

derive an edge-based finite element scheme by noting that the element matrices can be disassembled into their edge contributions as,

$$K^e = \sum_{s=1}^{m} K_s^e \tag{5}$$

where K_s^e is the contribution of edge s to K^e and m is the number of element edges, which is 6 for tetrahedra or 28 for hexaedra.

Denoting by $\mathcal{E}$ the set of all elements sharing a given edge s, we may add their contributions, arriving to the edge matrix,

$$K_s = \sum_{s \in \mathcal{E}} K_s^e \tag{6}$$

The resulting matrix is symmetric, and we need to store only the upper off diagonal 3×3 block per edge. The edge-by-edge matrix-vector product may be written as,

$$Kp = \sum_{s=1}^{nedges} K_s p_s \tag{7}$$

where *nedges* is the total number of edges in the mesh and p_s is the restriction of p to the edge degrees-of-freedom. In Table 1 we compare the storage requirements to hold the coefficients of the element stiffness matrices and the edge stiffness matrices as well as the *flop* count and indirect addressing (*i/a*) operations for computing matrix-vector products using element and edge-based data structures for tetrahedral meshes. All data in these tables are referred to *nnodes*, the number of nodes in the finite element mesh. According to Lohner (1994), the following estimates are valid for unstructured 3D grids, $nel \approx 5.5 \times nnodes$, $nedges \approx 7 \times nnodes$.

Table 1. Memory to hold the stiffness matrix coefficients and computational costs for element and edge-based matrix-vector products for tetrahedral finite element meshes

Data Structure	*Memory*	*flop*	*i/a*
EBE	429 × *nnodes*	1,386 × *nnodes*	198 × *nnodes*
Edges	63 × *nnodes*	252 × *nnodes*	126 × *nnodes*

For meshes composed by 8-noded hexaedra we performed a study to access the asymptotic ratio between the number of edges and the number of elements. Figure 1

shows, for an increasing number of divisions along each direction of a cube, the computed ratio between the number of resulting edges and the number of hexahedral finite elements in the meshes. We may note that the curve tends to an asymptotic ratio of 13.

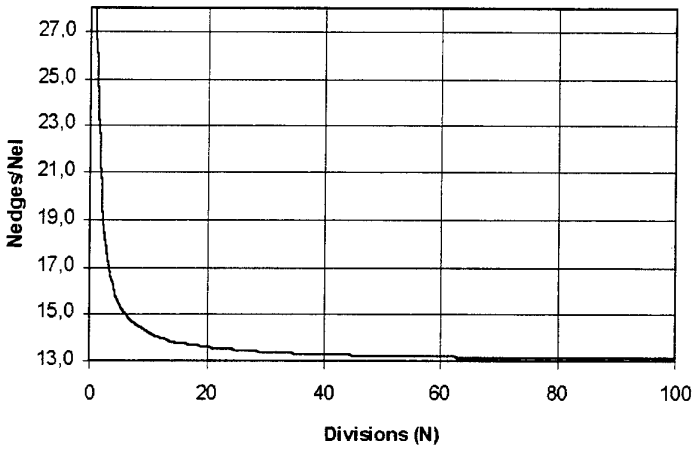


Fig 1. Edges/elements ratio on a cube.

Considering the computed asymptotic ratio, we built Table 2, which compares memory estimates to hold the stiffness matrix coefficients and operation counts to compute the matrix-vector products for hexahedral meshes, considering the element-by-element (EBE) and edge-based strategies. In this Table we considered $nel \approx nnodes$, $nedges \approx 13 \times nel$.

Table 2. Memory to hold the stiffness matrix coefficients and computational costs for element and edge-based matrix-vector products for hexahedral finite element meshes.

Data Structure	*Memory*	*flop*	*i/a*
EBE	$300 \times nnodes$	$1{,}152 \times nnodes$	$72 \times nnodes$
Edges	$117 \times nnodes$	$336 \times nnodes$	$234 \times nnodes$

Clearly data in Tables 1 and 2 show the superiority of the edge-based scheme over element-by-element strategies. However, compared to EBE data structure, the edge scheme does not present a good balance between *flop* and *i/a* operations. Indirect addressing represents a major CPU overhead in vector, RISC and cache-based parallel machines. To improve this ratio, Lohner (1994) have proposed several alternatives to the single edge scheme. The underlying concept of such alternatives is that once data has been gathered, reuse them as much as possible. This idea, combined with node renumbering strategies, Lohner (1998), introduces further enhancements in the finite

element edge-based scheme. We have found (Martins et al, 1997) that, for tetrahedral meshes, structures formed by gathering edges in spatial triangular and tetrahedral arrangements, the superedges, present a high data reutilization ratio and are simple to implement. The superedges are formed reordering the edge list, gathering edges with common nodes to form tetrahedra and triangles. To make a distinction between elements and superedges, we call a triangular superedge a *superedge3* and a tetrahedral superedge a *superedge6*. The matrix-vector product for a *superedge3* may be expressed as,

$$Kp = \sum_{s=1,4,7...}^{ned3} (K_s p_s + K_{s+1} p_{s+1} + K_{s+2} p_{s+2}) \tag{8}$$

and for a *superedge6*,

$$Kp = \sum_{s=1,7,13...}^{ned6} (K_s p_s + K_{s+1} p_{s+1} + K_{s+2} p_{s+2} + K_{s+3} p_{s+3} + K_{s+4} p_{s+4} + K_{s+5} p_{s+5}) \tag{9}$$

where *ned3* and *ned6* are respectively the number of edges grouped as *superedge3*'s and *superedge6*'s. Table 3 gives the estimates for *i/a* reduction and *flop* increase for both types of superedges. We may see that we achieved a good reduction of *i/a* operations per edge, with a negligible increase of float point operations.

Table 3. Indirect addressing reduction and *flop* increase for the superedges.

Type	*Edges*	*Nodes*	*ia/edge*	*i/a reduction*	*flop/edge*	*flop increase*
Edge	1	2	18:1	1.00	46:1	1.00
Superedge3	3	3	27:3	0.50	134:3	0.97
Superedge6	6	4	36:6	0.33	302:6	1.09

For hexahedral meshes we may gather the edges forming the superedges shown in Figure 2.

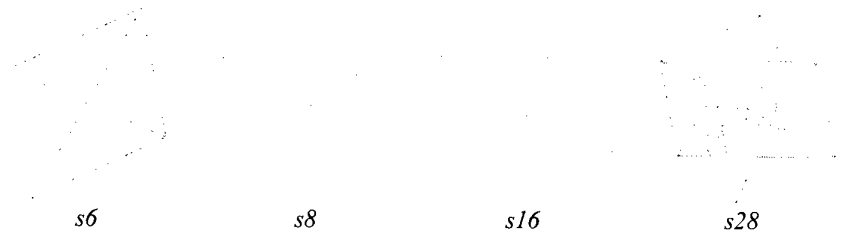


Fig 2. Superedge arrangements for hexaedra.

The resulting matrix-vector products for each superedge type can be expressed as,

$$Kp = \sum_{s=1,7,13...}^{ned6} (K_s p_s + K_{s+1} p_{s+1} + K_{s+2} p_{s+2} + K_{s+3} p_{s+3} + K_{s+4} p_{s+4} + K_{s+5} p_{s+5}) \quad (10)$$

$$Kp = \sum_{s=1,9,17...}^{ned8} (K_s p_s + K_{s+1} p_{s+1} + K_{s+2} p_{s+2} + ... + K_{s+6} p_{s+6} + K_{s+7} p_{s+7}) \quad (11)$$

$$Kp = \sum_{s=1,17,33...}^{ned16} (K_s p_s + K_{s+1} p_{s+1} + K_{s+2} p_{s+2} + ... + K_{s+14} p_{s+14} + K_{s+15} p_{s+15}) \quad (12)$$

$$Kp = \sum_{s=1,29,57...}^{ned28} (K_s p_s + K_{s+1} p_{s+1} + K_{s+2} p_{s+2} + ... + K_{s+26} p_{s+26} + K_{s+27} p_{s+27}) \quad (13)$$

where *ned6*, *ned8*, *ned16* and *ned28* are respectively the number of edges grouped as *s6*, *s8*, *s16* and *s28* types. Table 4 gives the estimates for i/a reduction and flop increase for these superedge types. We may observe that we also achieved a good i/a reduction with a negligible increase of float point operations. However coding complexity is increased, particularly for the *s16* and *s28* superedges. For a given finite element mesh we first reorder the nodes by Reverse Cuthill-Mckee algorithm to improve data locality. Then we extract the edges forming as much as possible *superedges*. After that we color each set of edges by a greedy algorithm to allow parallelization on shared vector multiprocessors and scalable shared memory machines. We have observed that for general unstructured grids more than 50% of all edges can be grouped into *superedges*.

Table 4. Indirect addressing reduction and *flop* increase for the superedges.

Type	*Edges*	*Nodes*	*ia/edege*	*i/a reduction*	*flop/edge*	*flop increase*
Edge	1	2	18: 1	1.00	46: 1	1.00
s6	6	4	36: 6	0.33	302: 6	1.09
s8	8	8	72: 8	0.50	410: 8	1.11
s16	16	8	72:16	0.25	861:16	1.17
s28	28	8	72:28	0.14	1361:28	1.06

4. NUMERICAL EXAMPLES

4.1 Performance assessment of edge-based matrix-vector product for tetrahedral meshes

The performances of the single edge-based matrix-vector product algorithm and the algorithms resulting from the decomposition of an unstructured grid composed by tetrahedra into *superedges* are shown in Table 5 and 6, respectively for a Cray J90 superworkstation and for a SGI Origin 2000 with r10000 processors. In these experiments we employed randomly generated indirect addressing to map global to local, that is, edge quantities. Table 5 lists the CPU times for the matrix-vector products on the Cray J90 for an increasing number of edges, supposing that all edges in the mesh may be grouped as *superedge3*'s or *superedge6*'s.

Table 5. CPU times in seconds for edge-based matrix-vector products on the Cray J90.

Number of Nedges	*Edges*	*Superedge3*	*Superedge6*
3,840	1.92	1.89	1.87
38,400	2.81	2.42	2.23
384,000	11.96	8.12	5.82
3,840,000	102.4	59.64	42.02
38,400,000	1,005.19	579.01	399.06

We may observe that gathering the edges in superedges reduces considerably the CPU times, particularly for in the *superedge6* case. Another set of experiments were conducted on the SGI Origin 2000, a scalable shared memory multiprocessor. The average results of 5 runs, in non-dedicated mode, considering a total number of edges of 2,000,000 are shown in Table 6. We may observe that all data structures present good scalability, but the superedges are faster. For 32 CPU's the *superedge6* matrix-vector product is almost 4 times faster than the product with single edges.

Table 6. CPU times in seconds for edge-based matrix-vector products on the SGI Origin 2000.

Processors	*Edges*	*Superedge3*	*Superedge6*
4	48.0	22.8	15.6
8	31.0	15.8	11.1
16	19.0	9.6	6.8
32	10.9	5.8	3.9

4.2 Performance assessment of edge-based matrix-vector product for hexaedrical meshes

The performances of the edge matrix-vector product algorithm and the algorithms for the *s6* to *s28* decomposition of a hexaedrical finite element mesh are shown in Tables 7 and 8, respectively for a Cray J90 superworkstation and for a SGI Origin 2000. These experiments were also conducted under the same conditions of the previous experiment, that is, we employed randomly generated indirect addressing to map global to edge quantities. Table 7 lists for each *superedge* arrangement the CPU times for the matrix-vector products on the Cray J90, supposing that all edges in the mesh can be grouped as *s6*, *s8*, *s16* or *s28 superedges*.

Table 7. CPU times in seconds for edge-based matrix-vector products on the Cray J90se.

Data Structure	*nedges* = 21,504	*nedge s*= 215,400	*nedges* = 2,150,400
Edge	0.011	0.109	1.084
s6	0.011	0.108	1.080
s8	0.014	0.134	1.346
s16	0.016	0.148	1.483
s28	0.013	0.123	1.233

We may observe that only the *s6* arrangement reduces the CPU time when compared to the performance of the single edge algorithm. This behavior may be attributed to the code complexity of *s8*, *s16* and *s28* algorithms. We also made similar experiments on the SGI Origin 2000, a scalable shared memory multiprocessor. The results, for the same amount of edges are listed in Table 8. We may observe that although the *s6* algorithm is still the faster, all other *superedge* arrangements are faster than the single edge matrix-vector algorithm. This may be credited to the memory hierarchy of the SGI Origin 2000, where data locality plays a fundamental role in cache optimization. Parallel performance in this case is similar to the results obtained

in the previous set of experiments. For *nedges*=2,150,000 all superedge arrangements achieved speed-up's around 4 on 32 processors with respect to a 4-processor run.

Table 8. CPU times in seconds for edge-based matrix-vector products on the SGI Origin 2000.

Data Structure	*nedges* = 21,504	*nedges* = 215,400	*nedges* = 2,150,400
Edge	0.011	0.14	1.33
s6	0.006	0.12	0.75
s8	0.007	0.15	1.22
s16	0.011	0.14	1.01
s28	0.009	0.13	1.23

4.3 Extensional Behavior of a Sedimentary Basin

We study the extensional behavior of a sedimentary basin presenting a sedimentary cover (4 km) over a basement (2 km) with length of 15 km and thickness of 6 km. The model has an ancient inclined fault with 500 m length and 60° of slope. The relevant material properties are compatible with the sediment pre-rift sequence and basement. We have densities of 2450 kg/m^3 and 2800 kg/m^3 respectively for the sediment layer and basement; Young's modulus of 20 GPa for the sedimentary cover and 60 GPa for the basement; Poisson's ratio, 0.3 for both rocks. The ratio between initial horizontal and vertical (gravitational) stresses is 0.429. We assume that both materials are under undrained conditions and modeled by Mohr-Coulomb failure criterion. Thus, we have sedimentary cover cohesion of 30 MPa, basement cohesion of 60 MPa, and internal friction angle of 30° for both materials. The finite element mesh (see Figure 3) comprises 2,611,036 tetrahedra, 445,752 nodal points and 3,916,554 edges. The number of *superedge6*'s is 57% of the total number of edges, while the number of superedge3's is just 6% of total. We consider the model simply supported at its left and bottom faces, and we apply tension stresses at the right face and shear stresses at the basement, opposing the basin extension. The loads are applied in 12 increments, and the analysis is performed until the complete failure of the model. Memory requirements to solve this problem employing element and edge-based data structures are respectively 203.9 and 35.3 Mwords respectively. We solve this problem on a 16 CPU's Cray J90se using the ITS method and the edge-based strategy. Displacement and residual tolerances were set to 10^{-3}. We selected PCG tolerances in the interval $[10^{-6}, 10^{-1}]$. The parallel solution took only 15 minutes of elapsed time, corresponding to 36 nonlinear ITS iterations and 9,429 PCG iterations. Figure 4 shows the yield ratio contours in the last load increment. The effects of the fault and the two material layers may be clearly seen.

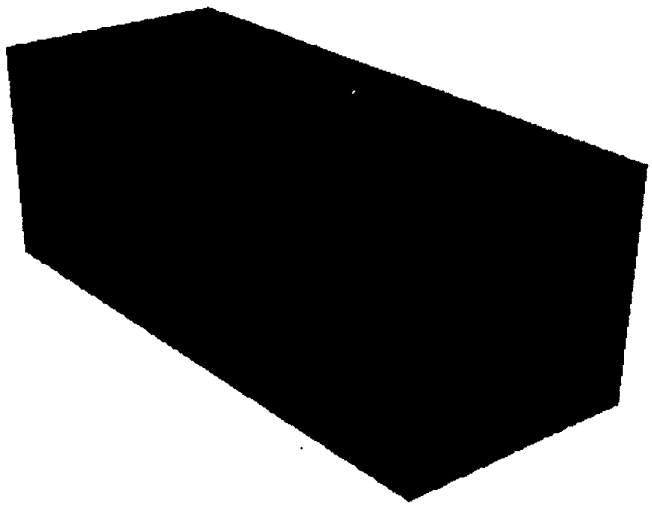


Fig. 3. Finite element mesh for the sedimentary basin.

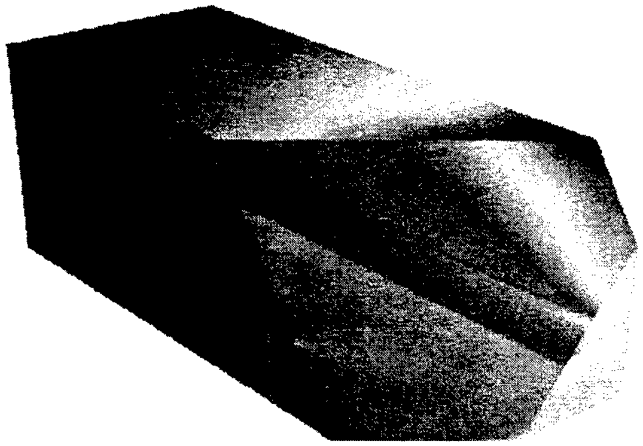


Fig. 4. Yield ratio contours for the sedimentary basin at 12th load increment.

5. CONCLUSIONS

We presented a fast, parallel, memory inexpensive, finite element solution scheme for analyzing large-scale 3d nonlinear solid mechanics. Our scheme employ novel nonlinear solution strategies and suitable data-structures, allowing us to tackle challenging problems in their full complexity. The novel data structures, based on edges, rather than elements, are applicable to meshes composed by tetrahedra and hexaedra. Grouping edges into *superedges* we may improve further the computational efficiency of the matrix-vector product, reducing the overhead associated to indirect addressing, particularly on scalable shared memory multiprocessors.

ACKNOWLEDGEMENTS

This work is partially supported by CNPq grant 522692/95-8. Computer time on the Cray J90 is provided by the Center of Parallel Computing at COPPE/UFRJ. The authors are indebted to SGI Brazil for providing computer time on a Cray J90se and an Origin 2000 at Eagan, MN, USA.

REFERENCES

Arguello, J.G., Stone, C.M., Fossum. A.F., Progress on the development of a three dimensional capability for simulating large-scale complex geologic process, 3rd North-American Rock Mechanics Symposium, Int. S. Rock Mechanics, paper USA 327-3, 1998.

Biffle, J.H., JAC3D - A three-dimensional finite element computer program for the nonlinear quasi-static response of solids with the conjugate gradient method, Sandia Report SAND87-1305, 1993.

Blaheta, R., Axelsson, O., Convergence of inexact newton-like iterations in incremental finite element analysis of elasto-plastic problems, Comp. Meth. Appl. Mech. Engrg, 141, pp. 281-295, 1997.

Crisfield, M.A., Nonlinear finite element analysis of solids and structures, John Wiley and Sons, 1991.

Ferencz, R.M., Hughes, T.J.R., Iterative finite element solutions in nonlinear solid mechanics, in Handbook for Numerical Analysis, Vol. VI, Editors, P.G. Ciarlet and J.L. Lions, Elsevier Science BV, 1998.

Kelley, C.T.H. Iterative Methods for Linear and Nonlinear Equations, SIAM, Philadelphia, 1995.

Lohner, R., Edges, stars, superedges and chains, Comp. Meth. Appl. Mech. and Engrg., Vol. 111, pp. 255-263, 1994.

Lohner, R., Renumbering strategies for unstructured-grid solvers operating on shared-memory, cache-based parallel machines, Comp. Meth. In Appl. Mech. Engrg., Vol. 163, pp. 95-109, 1998.

Luo, H, Baum, J.D., Lohner, R., Edge-based finite element scheme for the euler equations, AIAA Journal, 32 (6), pp. 1183-1190, 1994.

Martins, M.A.D., Coutinho, A.L.G.A., Alves, J.L.D., Parallel iterative solution of finite element systems of equations employing edge-based data structures, 8th SIAM Conference on Parallel Processing for Scientific Computing, Editors, M. Heath et al, 1997.

Papadrakakis, M., Solving large-scale problems in mechanics: the development and application of computational solution procedures, John Wiley and Sons, 1993.

Peraire, J., Peiro, J., Morgan, K., A 3d finite element multigrid solver for the euler equations, AIAA Paper 92-0449, 1992.

A Multiplatform Distributed FEM Analysis System using PVM and MPI

Célio Oda Moretti[1], Túlio Nogueira Bittencourt[1], and Luiz Fernando Martha[2]

[1] Computational Mechanics Laboratory, Department of Structural and Foundation Engineering, Polytechnic School, University of São Paulo,
Av. Prof. Almeida Prado, trav. 2, no. 83,
CEP 05508-900 - São Paulo - Brazil
Phone: +55 11 818-5367 / Fax: +55 11 818-5181
{moretti, tbitten}@usp.br
http://www.lmc.ep.usp.br

[2] Department of Civil Engineering and Technology Group on Computer Graphics - TeCGraf, Pontifical Catholic University of Rio de Janeiro - PUC-Rio,
Rua Marquês de São Vicente, no. 225,
CEP 22453-900 - Rio de Janeiro - Brazil
Phone: +55 21 512-5984 / Fax: +55 21 259-2232
lfm@tecgraf.puc-rio.br
http://www.tecgraf.puc-rio.br

Abstract. A multiplatform computational system for parallel finite element structural analysis using a distributed memory environment is described in this paper. The complete system is comprised by integrated programs, each of one responsible for a different task: pre-processing, mesh partitioning (necessary to perform the parallel analysis), structural analysis and post-processing. The main focus here is the structural analysis program, showing details of features and added capabilities, necessary to work in a High Performance Computing environment. An existing finite element method program (FEMOOP) has been adapted to implement the parallel features. A important feature of this program is its portability, which allows FEMOOP to work in different computational platforms, taking advantage of specific capabilities of each platform. FEMOOP works with PVM and MPI libraries. The main objective of this work is to show the parallel analysis program performance running in different computational platforms, and using the two communication libraries PVM and MPI. The two communication libraries have been used. The processing time and speed-up of model analyses are shown, and the results are compared and discussed.

1 Introduction

Nowadays a great variety of parallel computer machines are disposable to be used, each of one with specific capabilities and advantages. However, these machines usually present different architectures and operational systems, which represent a barrier to adapt a same parallel analysis program code efficiently to different machines.

In this paper, a multiplatform computational system for parallel finite element structural analysis will be presented. This system is capable to work in different computational platforms and architectures. To attain this capability, the main feature of the programming code must be the portability, which allows an easy adaptation to different platforms. The complete system is comprised by integrated programs, each of one responsible for a specific task: pre-processing, mesh partitioning, structural analysis and post-processing.

The main focus in this work is the structural analysis program. This program is called FEMOOP (Finite Element Method - Object Oriented Programming) [1] and it has been adapted to work in a parallel environment. FEMOOP is organized using object-oriented concepts of the C++ programming language [2][3], and has been developed at the Department of Civil Engineering (PUC-Rio) and at Computational Mechanics Laboratory (Polytechnic School / USP). New features and capabilities have been added to this originally sequential program to adapt it to work in a distributed memory environment. This environment uses the message passing to perform the communication among the various processes. There are some communication libraries that manage this message passing process. FEMOOP can work with PVM (Parallel Virtual Machine) [4] and MPI (Message Passing Interface) [5]. Both are well know communication libraries that can be used in different platforms.

In the following sections, the structural analysis program and the computational environments used to run it will be described. The communication libraries PVM and MPI will also be described, showing advantages and disadvantages of each one. Finally, some examples of a same model running in different platforms and using the two communication libraries will be presented. The processing time and speed-up of these examples will be compared and discussed.

2 The Analysis System

In the system presented here, the structural analysis is performed by a finite element program called FEMOOP (Finite Element Method - Object Oriented Programming) [1], which is organized using object-oriented concepts [2][3]. One of the most important advantages of the object-oriented programming is the code extensibility. This feature allows new implementations with minimum impact over the existent code. Another important feature of the program code is its portability, which allows an easy code adaptation to different platforms. These two features have allowed the adaptation of the original sequential code to a parallel environment and to different computational platforms. The parallel environment considered in this work is a distributed memory environment, which can be comprised by a local area network, a parallel computer or a multiprocessor machine. In a previous work [6], parallel analyses running in a local area network was presented. In this case, the local area network can be viewed as a virtual parallel machine with multiple processors and distributed memory. In the present work, the analyses will be performed in a parallel computer and

a multiprocessor machine. These computational environments will be described later in this paper.

To adapt FEMOOP to the parallel computational environment a new class has been created, which is responsible for data manipulation. Also, a series of new functions has been implemented into existent classes. The first step necessary to adapt FEMOOP to the parallel environment has been the implementation of a library responsible for the message passing management. The main objective of this library is to limit the direct access message passing functions. This access limitation facilitates an eventual change of the message passing manager or the addition of a new one. A change or addition of a message passing manager has impact only over the library code. This parallel procedure library contains all functions necessary to perform the message passing in a distributed memory environment. The main functions implemented here are responsible for sending and receiving messages among processors, for parallel process initialization, and for the identification of program type (if it is either a master or a task program). Using these features and facilities, the present version of FEMOOP can work with two communication libraries: PVM (Parallel Virtual Machine) [4] and MPI (Message Passing Interface) [5]. These two well know communication libraries have a portable code, which is an important feature for the work developed here. The implementation used and characteristics of these two libraries will be described later in this paper. The user can choose between PVM and MPI at compilation time.

The parallel programming paradigm adopted here has been the master-slave model. In this model, the master is a separate program responsible for process spawning, initialization, reception and display of results, and timing of functions. The task (or slave) programs are executed concurrently and interact through message passing. The actual structural analysis is done by the task programs, each of one responsible for the work corresponding to one substructure. Through interaction between these task programs, the global solution is obtained and then it is sent to the master program.

To obtain the linear system of equilibrium equations a substructuring technique [7] has been employed. When this technique is used, the original structure is partitioned into a number of substructures, which are distributed among the processors. The substructure degrees of freedom are classified as internal DOF and boundary DOF, which are shared between neighbor substructures. The substructures interact through this boundary DOF only. Then, the stiffness matrices of each substructure are mounted. In this work, the internal unknowns are eliminated using Crout method. After this step, a condensed system with terms corresponding only to boundary unknowns is obtained. All these procedures can be performed concurrently. To solve the partitioned global system, a parallel iterative solver has been used. A parallel implementation of the pre-conditioned conjugate gradient (PCG) method [8][7] has been chosen as the solution method adopted in this work. Basically, this parallel implementation of the PCG method consists of parallel operations between matrices and vectors. The sequence of op-

erations is the same both in the parallel and in the sequential versions of the PCG method.

To employ this substructuring technique, the stiffness matrix $K^{(i)}$, the force vector $f^{(i)}$, and the nodal unknowns $u^{(i)}$, corresponding to each substructure i, have been mounted with terms partitioned into internal and boundary terms. These partitions are presented in Equation 1, where indices I and S correspond respectively to internal and boundary (or shared) terms.

$$K^{(i)} = \begin{bmatrix} K_I^{(i)} & K_{IS}^{(i)} \\ K_{IS}^{(i)^T} & K_S^{(i)} \end{bmatrix}, f^{(i)} = \begin{bmatrix} f_I^{(i)} \\ f_S^{(i)} \end{bmatrix}, u^{(i)} = \begin{bmatrix} u_I^{(i)} \\ u_S^{(i)} \end{bmatrix}. \tag{1}$$

For each substructure, the linear equation system is written in the form

$$\begin{bmatrix} K_I^{(i)} & K_{IS}^{(i)} \\ K_{IS}^{(i)^T} & K_S^{(i)} \end{bmatrix} \begin{bmatrix} u_I^{(i)} \\ u_S^{(i)} \end{bmatrix} = \begin{bmatrix} f_I^{(i)} \\ f_S^{(i)} \end{bmatrix}. \tag{2}$$

Eliminating the internal unknowns, a condensed equation system is obtained

$$\overline{K}_S^{(i)} u_S^{(i)} = \overline{f}_S^{(i)}, \tag{3}$$

where $\overline{K}_S^{(i)}$ and $\overline{f}_S^{(i)}$ are respectively the condensed stiffness matrix and the condensed force vector, and

$$\overline{K}_S^{(i)} = K_S^{(i)} - K_{IS}^{(i)^T} K_I^{(i)^{-1}} K_{IS}^{(i)}, \tag{4}$$

$$\overline{f}_S^{(i)} = f_S^{(i)} - K_{IS}^{(i)^T} K_I^{(i)^{-1}} f_I^{(i)}. \tag{5}$$

The global linear equation system can be written in the form

$$\left[\sum_{i=1}^{p} L^{(i)^T} \overline{K}_S^{(i)} L^{(i)} \right] = \sum_{i=1}^{p} L^{(i)^T} \overline{f}^{(i)}, \tag{6}$$

where p is the number of substructures and $L^{(i)}$ is a boolean matrix that describes the substructure connectivity in the original structure.

The adaptation to different platforms has been done without need of FEMOOP code changing. The unique requirement is that the desirable communication library is present in the compilation platform. As already mentioned, the user chooses the communication library at compilation time. If necessary, the desirable communication library must be compiled and installed in the current platform. PVM and MPI implementations have a very portable code and both can be compiled in many different platforms.

The complete analysis system is comprised by integrated programs, each of one responsible for a specific task. The package MTOOL (Bidimensional Mesh Tool) [9] has been used as the pre-processor in this work. MTOOL is an interactive graphics program for bidimensional finite element mesh generation. With this program, the geometry, material properties, and boundary conditions

of the model are defined. After the model generation, the structure has to be partitioned into a number of substructures to take advantage of the parallel environment. This partitioning is made through the use of PARTDOM program [10]. The parallel analysis is performed by FEMOOP program. A program named MVIEW (Bidimensional Mesh View) [11] has been utilized as the post-processor tool. MVIEW is an interactive graphical program for visualization of structural analysis results. For a complete description of the entire parallel analysis system, see [6].

3 Computational Environments

In this work, two computational environments have been used to perform the parallel analysis. These environments have different architectures and operating systems, and the analysis program FEMOOP could be used at both environments without code change. In the following sections, the environments will be described.

3.1 IBM SP-2

One of the machines used in this work was an IBM SP-2, available from Advanced Scientific Computation Laboratory (LCCA) of University of São Paulo. The SP2 machine has 12 processors and 4.5 Gb of memory and a great capacity of processing, data storage and memory, which allows large scale model analyses.

3.2 SUN Ultra Enterprise 3000

Another machine used in this work was multiprocessor workstation SUN Ultra Enterprise 3000 (E3000). The E3000 machine has 6 processors and 1.5 Gb of memory. This workstation is a fully dedicated machine to parallel processing, which allows the total use of the machine capacity. Table 1 presents a summary of the specifications of the two machines used in this work.

4 Communication Libraries

As already mentioned, FEMOOP can work with two communication libraries: PVM and MPI, which are briefly described in the following sessions.

4.1 Parallel Virtual Machine (PVM)

PVM is an integrated set of software tools and libraries that emulates a concurrent computing framework on interconnected computers of different architectures. The main objective of the PVM system is to enable a collection of computers to be used for concurrent or parallel computation. A heterogeneous computer network can be viewed as a single parallel computer. In this work, the 3.3.10 PVM version has been used, which has been developed by Oak Ridge National Laboratory.

Table 1. IBM SP-2 and E3000 specifications

Specification	IBM SP-2	E3000
Number of processors	12	6
Memory (GB)	4.5	1.5
Disk space (GB)	109	54
Architecture	POWER2 IBM Risc/6000	Superscalar SPARC version 9 UltraSparc
Operating system	AIX 4.1.4	Solaris 2.5.1
Fully dedicated to parallel processing	No	Yes

4.2 Message Passing Interface (MPI)

MPI is a message-passing application programmer interface, with specifications proposed by a broadly committee of vendors, implementers, and users. Any MPI implementation must respect these specifications, which define protocols and semantics (such as a message buffering and message delivery progress) used to manage the message passing process.

In SP2 machine was used the IBM AIX Parallel Environment with a native implementation of MPI standards [12]. In E3000 machine was used the MPICH implementation [13], developed by Argone National Laboratory and Mississippi State University. In the two machines was used the first version of MPI standards (MPI-1).

5 Examples

In this section, the performance and speed-up of analyses of a same model running on the two used computational environments are presented. These analyses have been performed using the two communication libraries PVM and MPI. The numerical example used to measure system performance is a beam with geometry, boundary conditions and mesh presented in Fig. 1. The model attributes are: E = 7000 kN/cm^2, $\nu = 0.25$ and thickness = 1 cm. The model has created and discretized using a regular mesh of 13x130 plane stress Q8 elements.

5.1 Example 1: IBM SP-2

In this section, the performance and speed-up of the parallel analysis system running in IBM SP-2 machine are presented. The analyses have been performed using 1 to 6 processors and using PVM and MPI communication libraries. Fig. 2 shows the total analysis time and Fig. 3 shows the corresponding speed-up.

A Multiplatform Distributed FEM Analysis System Using PVM and MPI

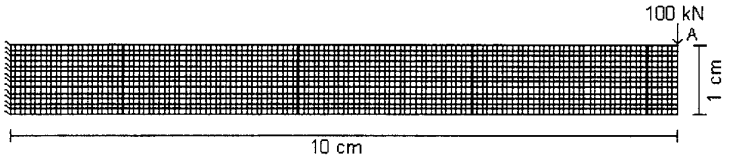


Fig. 1. Model geometry, boundary conditions and mesh

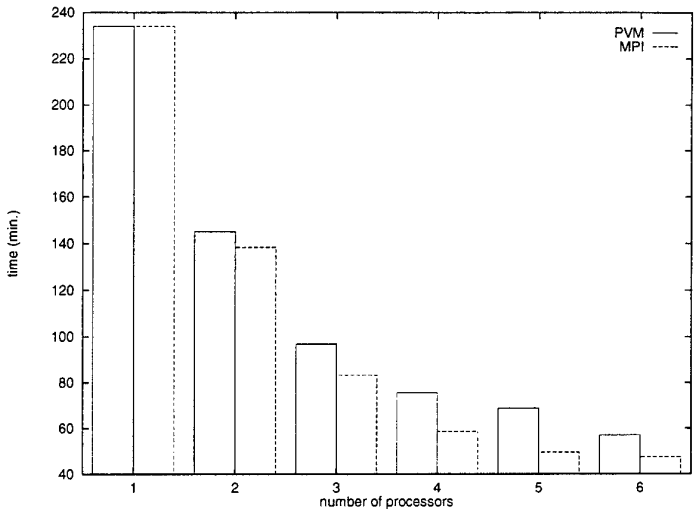


Fig. 2. Total time consumed to complete the model analysis using PVM and MPI communication libraries running on a IBM SP2

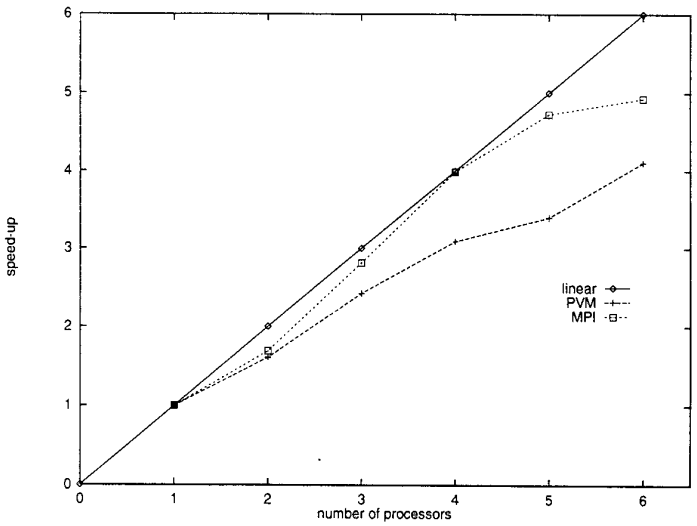


Fig. 3. Speed-up of model analysis using PVM and MPI communication libraries running on a IBM SP2. The linear speed-up represents a system with ideal scalability

5.2 Example 2: SUN Ultra Enterprise 3000

In this section, the performance and speed-up of the parallel analysis system running in a SUN Ultra Enterprise 3000 machine are presented. The analyses have been performed using 1 to 6 processors and using PVM and MPI communication libraries. Fig. 4 shows the total analysis time and Fig. 5 shows the corresponding speed-up.

The results presented in Figures 2 to 5 show that MPI library performance is often better than PVM library, mainly in the SP-2 machine. The use of a native implementation of MPI library in the SP-2 machine increases its performance and machine scalability, which can be seen in Fig, 2 and 3. This native implementation, developed by IBM, explores the SP-2 capabilities more efficiently than the portable implementation of PVM. In the E3000 machine, portable implementations of PVM and MPI have been used and the performances of both are very close.

6 Conclusions

The parallel analysis system presented can work in different parallel platforms, such as a parallel supercomputer, a multiprocessor machine or a local area network. This feature allows the use of the same parallel analysis program in the currently available platform. With this flexibility, the user can performs a low cost parallel analysis using an available local area network, or a high performance parallel analysis using a parallel computer.

A Multiplatform Distributed FEM Analysis System Using PVM and MPI

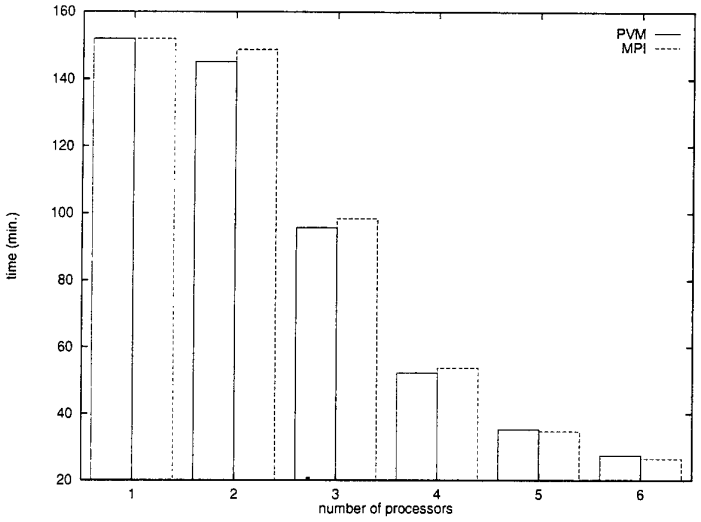


Fig. 4. Total time consumed to complete the model analysis using PVM and MPI communication libraries running on a Sun Ultra Enterprise 3000

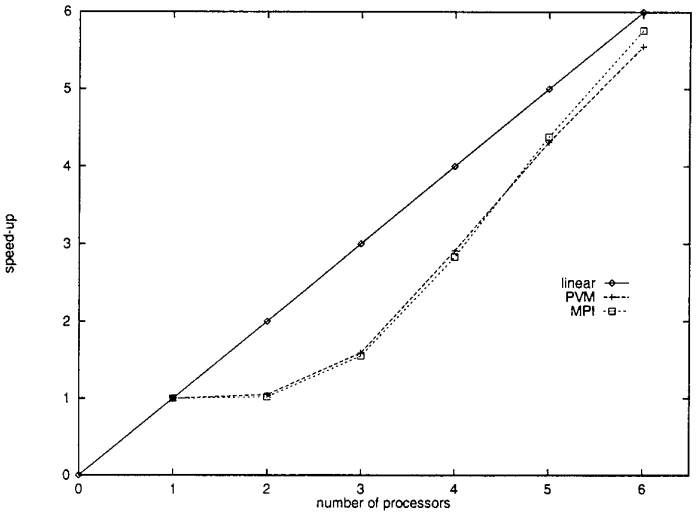


Fig. 5. Speed-up of model analysis using PVM and MPI communication libraries running on a Sun Ultra Enterprise 3000. The linear speed-up represents a system with ideal scalability

The adaptation to a new computational environment is done without code change, which allows an easy compilation work. The performance results presented in this work show that this parallel analysis system can explore the capabilities of different machines very efficiently. The use of PVM and MPI communication libraries gives more flexibility to this system and increases the number of platforms that can be used.

References

1. Martha, L.F., Menezes, I.F.M., Lages, E.N., Parente Jr., E., Pitangueira, R.L.S.: An OOP Class Organization for Materially Nonlinear Finite Element Analysis, Joint Conference of Italian Group of Computational Mechanics and Ibero-Latin American Association of Computational Methods in Engineering, Padova, Italy, Sep. 1996, pp. 229-232, 1996.
2. Fujii, G.: Análise de Estruturas Tridimensionais: Desenvolvimento de uma Ferramenta Computacional Orientada para Objetos, Dissertação de Mestrado, Dep. de Engenharia de Estruturas e Fundações (PEF), Escola Politécnica, USP, 1997.
3. Guimarães, L.G.S., Menezes, I.F.M., Martha, L.F.: Object Oriented Programming Discipline for Finite Element Analysis Systems (in Portuguese), Proceedings of XIII CILAMCE, Porto Alegre, RS, Brasil, Vol. 1, pp. 342-351, 1992.
4. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderman, V.: PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, 1994.
5. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference, The MIT Press, Cambridge, Massachusetts, 1996.
6. Moretti, C.O., Bittencourt, T.N., Martha, L.F.: A Low Cost Distributed System for FEM Parallel Structural Analysis, VECPAR'98 - 3rd International Meeting on Vector and Parallel Processing, Porto, Portugal, June 21-23, pgs 1063-1075, 1998.
7. Nour-Omid, B., Raefsky, A., e Lyzenga, G., Solving Finite Element Equations on Concurrent Computers, in A.K. Noor, Ed., Parallel Computations and Their Impact on Mechanics, pp. 209-227, ASME, New York, 1987.
8. Hestenes, M. e Stiefel, E., Methods of Conjugate Gradients for Solving Linear Systems, Journal of Research of the National Bureau of Standards, Vol. 49, No. 6, pp. 409-436, Research Paper 2379, December 1952.
9. MTOOL - Bidimensional Mesh Tool (Versão 1.0) - Manual do Usuário, Grupo de Tecnologia em Computação Gráfica - TeCGraf / PUC-Rio, 1992.
10. Moretti, C.O., Bittencourt, T.N., André, J.C., and Martha, L.F., Algoritmos Automáticos de Partição de Domínio, Boletim Técnico, Departamento de Engenharia de Estruturas e Fundações, Escola Politécnica - USP, 1998.
11. MVIEW - Bidimensional Mesh View (Versão 1.1) - Manual do Usuário, Grupo de Tecnologia em Computação Gráfica - TeCGraf / PUC-Rio, 1993.
12. IBM AIX Parallel Environment: MPI Programming and Subroutine Reference, No. GC23- 3894
13. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, v. 22, no. 6, pp. 789-828, 1996.

Synchronous NonLocal Image Processing on Orthogonal Multiprocessor Systems

Leonel Sousa and Oliver Sinnen

Department of Electrical Engineering IST/INESC
R. Alves Redol 9, 1000 Lisboa, Portugal
las@inesc.pt

Abstract. A method for mapping nonlocal image processing algorithms onto Orthogonal Multiprocessor Systems (OMP) is presented in this paper. Information is moved between memory modules by alternating the processors mode of accessing the memory array. The introduction of synchronisation barriers when the processors attempt to change the memory access mode of the OMP architecture synchronises the processing. Two parallel and nonlocal algorithms of the low and intermediate image processing levels are proposed and their efficiency is analysed. The performance of the OMP system for these type of algorithms was evaluated by simulation.

1 Introduction

The OMP architecture can be classified as a parallel shared memory architecture [5, 8]. The most important characteristics of a n-processor OMP architecture are (see Fig. 1a): *i*) the memory is divided in n^2 memory modules organised as a two-dimensional array; *ii*) the processors and the memory modules are interconnected by non-shared buses; *iii*) processors are allowed to concurrently access distinct rows or columns of the memory array.

The OMP architecture has two different mutually exclusive modes of operation, which correspond to different ways of accessing the memory: *row access mode* and *column access mode*. With the architecture in *row access mode*, any processor P_i has direct access to the row i of the array of memory modules ($M_{i,j}$ $0 \leq j < n$); with the architecture in *column access mode*, any processor P_i has direct access to the column i of the array of memory modules ($M_{j,i}$ $0 \leq j < n$). Therefore, the system buses are not shared and the memory access is free of conflicts in each one of the access modes.

The communication between any pair of processors of an OMP architecture can take place on two different memory modules: *e.g.* the pair of processors (P_i, P_j) can communicate through the memory modules $(M_{i,j}, M_{j,i})$. On the other hand, modules located on the main diagonal of the memory matrix are only accessed by a single processor: *e.g.* module $M_{i,i}$ is only accessed by processor P_i.

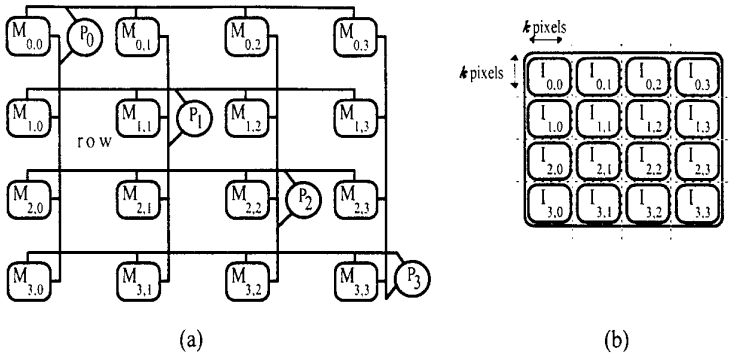


Fig. 1. OMP architecture with 4 processors: a) logical diagram; b) distribution of the image by the memory modules.

Spatial (image) parallelism is often found in low and intermediate level image processing operators [11]. Unlike task parallelism, in image parallelism the algorithm is applied in parallel to separated parts of the original image. Generally, OMP systems have a rather low number of processors relatively to the number of pixels of an image. The image is distributed by the processors of an OMP architecture by mapping the array of pixels into the array of memory modules (see Fig. 1b): a $N \times N$ image is divided into n^2 sub-images, with $\kappa = N/n$ neighbour pixels, and sub-images with neighbour pixels are placed in adjacent memory modules.

This paper presents the design and analysis of nonlocal image processing parallel algorithms for OMP systems. The parallel algorithms considered, presuppose the above mentioned distribution of the image among the memory modules. This paper begins by presenting a general method for mapping nonlocal image processing on OMP systems (Section 2). Afterwards some parallel algorithms for typical image processing tasks are proposed (Section 3). In Section 4 the performance of an OMP image processing system is evaluated using the simulation results. Finally Section 5 is devoted to the conclusions.

2 Nonlocal Image Processing on OMP Systems

The fundamental difference between nonlocal and local image processing is due to the restriction-less location of the pixels needed to process the image, in the first case. Thus there is a much more demanding interaction between processors in nonlocal image processing tasks. The rules that govern the information transfers can be unknown *a priori*, because they depend on the processing parameters and/or on the image content.

Communication oriented models of parallel processing explicitly include communication aspects of the processing. The parallel processing results from an alternate sequence of stages: a *computation stage* followed by a *communication*

stage [14]. In a computation stage, the processing is carried out independently by the multiple processors. In the communication stages, instead, nonlocal information, required for the next processing stage, is exchanged among the processors.

With the image parallelism approach, parallel processing stages consist on the application of similar image processing operators to different parts of the image. The processors of a OMP system have direct access to the image pixels stored on the row and column modules of a two-dimensional memory array.

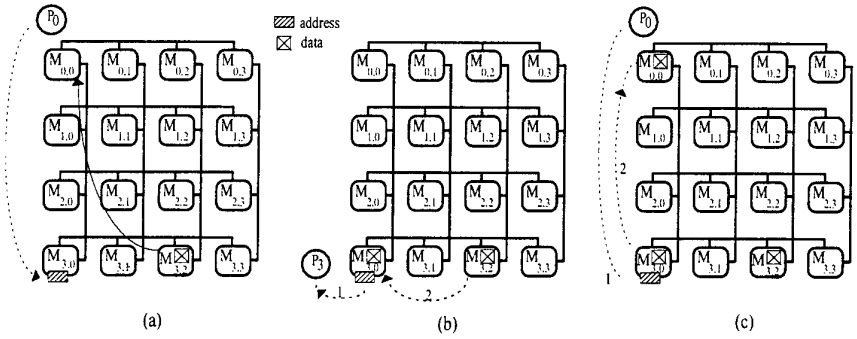


Fig. 2. Data transfer procedure for nonlocal image processing on OMP systems.

Nonlocal image processing often requires that random transfers of data be carried out at the communication stages. As referred in Section 2, processors of an OMP system may exchange data through pairs of exclusive memory modules. Therefore, if the processor P_i needs to access data stored on the memory module $M_{j,k}$ it has to establish communication with processor P_j or P_k. With the OMP architecture in the *column access* mode, data can be moved using the following procedure (the procedure is illustrated in Fig 2 for P_0 and $M_{3,2}$): step a) P_i places the address of the data, namely the index of array memory row and the relative position inside the module, on the memory $M_{j,i}$; step b) memory access mode is changed and P_j accesses the data address on $M_{j,i}$ and moves the data from $M_{j,k}$ to $M_{j,i}$; step c) the architecture is put back in the initial memory access mode and processor P_i directly accesses the claimed data on the $M_{j,i}$ memory.

This procedure can be extended in order to transfer data and partial results concurrently among any processors of a OMP system. At the end of a processing stage, the addresses of the data to be transfered are placed in the memory modules according to the rules presented above. Processors request a change of the memory access mode to signal the transition from a processing stage to a communication stage. These requests are used to implement a synchronisation barrier: each processor reaching the barrier waits until all other processors have also reached the barrier. A communication stage begins by changing the memory access mode of the architecture. Then, each processor looks for the addresses of

the data that have to be moved on all memory modules of a row (see Fig. 2b) and undertakes the task. In spite of the data being concurrently moved by the processors, there is no guarantee about a fair work distribution between the processors, since the source and destination addresses are unknown *a priori*. The exit of a communication stage is made by implementing another synchronisation barrier with the requests of the processors for changing the memory access mode of the architecture.

The time spent in the communication stages can lead to a considerable decrease of the system efficiency. The procedure proposed for global transfer of data is simple and general but can, itself, present the following drawbacks: a) the time spent for the data transfer can be relevant relatively to the processing time; b) the task of data transfer can be unbalanced among the multiple processors. Statement a) is connected with the granularity of the parallelism and with the time spent in the control of the architecture. The other statement also influences the time spent in the communication stages. An unbalanced transfer load can also contribute to the lowering of the processing efficiency.

These problems are referred in the following sections with the analysis of typical nonlocal image processing tasks. Efficient parallel algorithms for geometric image transformations and for the Hough transform are subsequently proposed.

3 Nonlocal Image Processing Parallel Algorithms

Two parallel algorithms used in nonlocal image processing tasks are proposed: an algorithm for image rotation, that requires a nonlocal exchange of information as a function of the rotation parameters; and an algorithm for computing the Hough transform of an image, which is a transformation commonly applied in intermediate level image processing.

The Image Rotation

Geometric transformations, namely rotation, are usually used for image normalisation [7]. For rotating an image by θ around a centre point (i_0, j_0), pixel values have to be moved to new locations, which have to be calculated through the parametric equations. Due to the discrete nature of the image representation, a well defined one-to-one transformation is applied to each pixel of the rotated image (i', j') to calculate its location on the original image (i, j):

$$\begin{aligned} i &= R_i^{-1}(i', j', \theta) = \lfloor (i' - i_0)\cos\theta + (j' - j_0)\sin\theta + i_0 \rfloor \\ j &= R_j^{-1}(i', j', \theta) = \lfloor (j' - j_0)\cos\theta - (i' - i_0)\sin\theta + j_0 \rfloor \quad . \end{aligned} \tag{1}$$

Pixel (i', j') of the processed image receives the value of pixel (i, j) of the original image, or a new value resulting from the interpolation of the values of the pixels that surround (i, j) [7].

The method for exchanging information on the OMP architecture, presented in the previous section, can be applied directly in parallel to the rotating pixels

of different sub-images. The coordinates of the pixels of different sub-images are computed in parallel and their values are placed in the proper memory modules. Two main decisions have to be taken in order to have a full specification of the parallel algorithm: the number of pixels that should be processed in parallel and their locations in the sub-images.

For taking the first decision two opposite conditions have to be considered: a) for each communication stage the memory access mode has to be changed twice and all memory modules have to be checked—more pixels per processing stage allows less relative time spent in communication; b) the extra memory required grows with the number of pixels processed per stage—the worse situation occurs when the n coordinates calculated by a processor stay in the same memory module ($\mathcal{O}(n^3)$ extra memory). The premise assumed in this paper points to the processing of n pixels by each processor in a certain processing stage.

The location of the pixels influences the balancing of the work load of the processors during the communication stages. In order to analyse the requirements of data movement, in the perspective of an orthogonal transfer, lets suppose that the transfer take place through the rows of the memory modules and consider the two simplified situations: a) the rotation of an image around its central point, using an algorithm that processes in parallel n pixels (1 pixel per processor) of an image row; b) a situation similar to the described above, except that the pixels to be processed belong to a single column of the image. This last situation is only used for analysing data transfer, because, in column access mode, the pixels can not be processed in parallel.

For the a) situation pixels processed in parallel have the same i' coordinate and a j' coordinates that differs by multiples of κ (see Fig. 1b), while for the b) situation an inverse relation between coordinates is observed. The pixels processed in the a) situation give rise to coordinates of the original image (Eq. 1) in the following rows of the memory array:

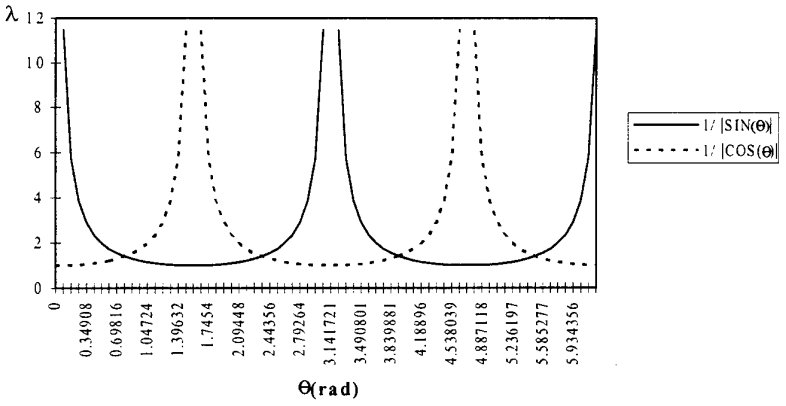


Fig. 3. Number of pixels to be transfered per row of memory modules (1 pixel per processor) for image rotation.

$$\lfloor i/\kappa \rfloor = \lfloor \frac{(i'-i_0)\cos\theta}{\kappa} + \frac{(j'-j_0)\sin\theta}{\kappa} + i_0/\kappa + \xi \times \sin\theta \rfloor \; ; \; 0 \leq \xi < n \quad . \quad (2)$$

In this case, the coordinates of two pixels go to the same row of memory modules for $\xi \times \sin\theta < 1$. So, the number of coordinates that go to a single row of memory modules (λ_a) can be approximately expressed by the equation $\lambda_a = 1/|\sin\theta|$.

In the b) situation pixels go to the memory modules of the following rows:

$$\lfloor i/\kappa \rfloor = \lfloor \frac{(i'-i_0)\cos\theta}{\kappa} + \frac{(j'-j_0)\sin\theta}{\kappa} + i_0/\kappa + \xi \times \cos\theta \rfloor \; ; \; 0 \leq \xi < n \quad . \quad (3)$$

The number of coordinates that go to a single row of memory modules (λ_b) can be approximately expressed by the equation $\lambda_a = 1/|\cos\theta|$.

A diagram of both functions (λ_a and λ_b) is depicted in fig. 3. The maximum number of coordinates in fig. 3 should not be infinite, but limited by the number of pixels processed. However, the diagram of fig. 3 shows that the maximum values of the functions λ_a and λ_b are out of phase ($\pi/2$). Moreover the maximum value of one function corresponds to the minimum value of the other, thus, if the coordinates of n rows of pixels (b situation) with n pixels each (a situation) are calculated in each processing stage then the maximum number of pixels that have to be transfered per memory row results from the combination of the numbers found for both situations.

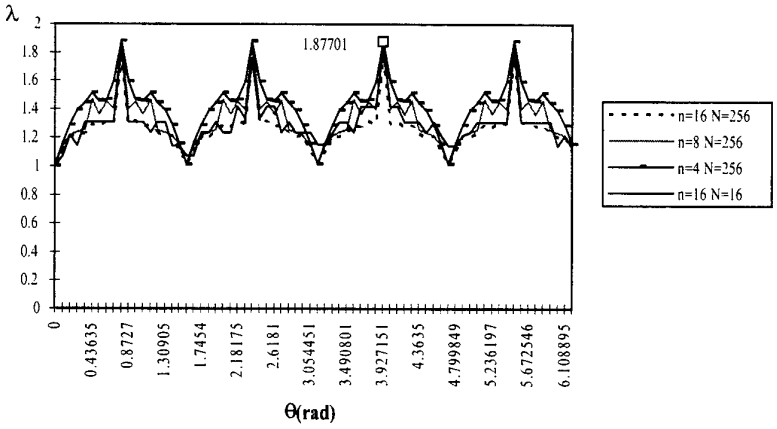


Fig. 4. Mean values of the maximum number of pixels that have to be transfered for a row of memory modules (n pixels per processor) for image rotation.

The diagram of fig. 4 displays the mean values of the maximum number of pixels that have to be transfered in a row of the memory modules, for rotating an image around its central pixel. The diagram, obtained by computer simulation,

shows that no more than $2 \times n$ pixels have to be transfered in any row of the memory modules. The maximum number of pixels occurs when the rotation angle is multiple of $\pi/4$—the intersection points of the functions λ in the diagram of the fig. 3.

```
   do_all processors (φ = 0, ..., n − 1)
      for l =0 to κ² − 1 do begin
      R_{φ,h} := 0 { number of pixels to transfer }
       <⊥ mode >
       for m =0 to n − 1 do begin
1.      { computes-(i, j) Eq. 1 for i' = m × κ + ⌊l/κ⌋; j' = φ × κ + l mod κ }
        if (i and j inside the image limits) then begin
2.        P_{m,φ} := i; Q := R_{⌊i/κ⌋,φ}++; S_{⌊i/κ⌋,φ}[Q].x := i; S_{⌊i/κ⌋,φ}[Q].y := j
        end
         else P_{m,φ} := −1
       end
       <⊢ mode >
       for h =0 to n − 1 do begin
        for m =0 to R_{φ,h} do begin
3.        i := S_{φ,h}[m].x; j := S_{φ,h}[m].y; T_{φ,h}[m] := I_O_{φ,⌊j/κ⌋}[i mod κ][j mod κ]
        end
        R_{φ,h} := 0
       end
       <⊥ mode >
       for m =0 to n − 1 do begin
        if (P_{m,φ} ≥ 0) then begin
4.        W := P_{m,φ}; Q := R_{m,φ}++; I_P_{m,φ}[⌊l/κ⌋][l mod κ] := T_{⌊W/κ⌋,φ}[Q]
       end
    end_all
```

Algorithm 1 *: Image rotation on the OMP architecture.*

The parallel algorithm for the image rotation on an OMP architecture is formally specified in Algorithm 1. The construct **do_all processors** $(\phi = 0, \ldots, n-1) \ldots$ **end_all** means that the processing inside the block is done in parallel by the n processors. The construct $< \ldots >$ indicates a synchronisation point for changing the memory access mode of the architecture ($\vdash$ for *row access* and $\perp$ for *column access*). Data structures are named using capital letters. Indexes are associated to them, for indicating the modules used, whenever the variables are stored in the shared memory. The original and processed image arrays are identified by the symbols I_O and I_P, respectively.

Lines 1 and 2 of the Algorithm 1 are devoted to the parallel computation of the coordinates of n pixels and to its placement on the adequate modules for orthogonal data transfer. Pixels of the original image are moved across the modules of each row of memory modules in line 3, with the architecture in the *row access* mode. In line 4, the values of the pixels are placed on the new coordinates, with the architecture in the *column access* mode. The procedure is repeated κ^2 times for processing all the $N \times N$ pixels of an image.

The complexity of the algorithm is $\mathcal{O}(\frac{N^2}{n})$ for lines 1, 2 and 4. The maximum number of pixels which have to be transfered in a row of memory modules (line 5 of the algorithm) is proportional to $\frac{N^2}{n}$ (see fig. 4). Therefore, the Algorithm 1 is $\mathcal{O}(\frac{N^2}{n})$ and has an efficiency of 100%.

The Hough Transform

The Hough transform, which is often applied for shape analysis due to its robustness to noise [1], is calculated using information about the binary edges of the images [6]. For detecting collinear points, the edge point (i, j) is transformed in a set of (ρ_l, θ_l) ordered pairs—ρ is the normal distance from the origin of the image to the line and θ is the angle of this normal with the horizontal axis. The following parametric equation is applied for any (i, j) using a stepwise increment for θ_l $(\delta\theta)$:

$$\rho_l = j \times \cos\theta_l + i \times \sin\theta_l \quad , \tag{4}$$

with collinear points voting to a common value (ρ_x, θ_x).

The processing load involved in the calculation of the Hough transform depends on the image features. Therefore, the distribution of the sub-images of equal size by the processors is not enough for guaranteeing the balancing of the processing load. One way of balancing the work of the processors is by segmenting the Hough space and by forcing each processor to compute Eq. 4 for all edge pixels of the image but only for a range of θ_l [2]. With the image distributed by the memory modules, an algorithm of this type demands the transmission of the coordinates of the edge points found by each processor to all other processors.

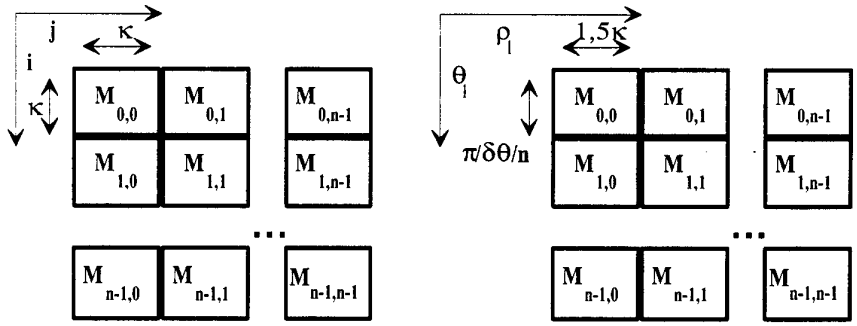


Fig. 5. Distribution of the image and the transformed space by the shared memory.

Let's distribute the image and the Hough space by the memory modules as depicted in Fig. 5. Each processor must transmit the coordinates of the edge points found in a set of pixels of its sub-image to all other processors, during a communication stage. If the pixels of the different sub-images that are processed in parallel have the same relative positions, the quantity of information which has to be transmitted can be reduced: only binary data about the presence of edges among the pixels is required.

A parallel algorithm for the Hough transform calculation on an OMP architecture is formally specified in Algorithm 2. This algorithm guarantees the balancing of the processing by the multiple processors, but requires an angular resolution $(\delta\theta)$ such as: $n \leq \pi/\delta\theta$.

```
do_all processors (φ = 0, ..., n − 1)
  for l =0 to κ² − 1 do begin
   R := 0
   for m =0 to n − 1 do begin
    <⊥ mode >
1.  if (I_O_{m,φ}[⌊l/κ⌋][l mod κ] ≠ 0) then R := R + 2^m
    T_{m,φ} := R
   end
   <⊢ mode >
   for h =0 to n − 1 do begin
    R := T_{φ,h}
    for m=0 to n − 1 do begin
     if((R ∧ 01) ≠ 0) then begin
      for t = φ × π/(δθ×n) to (φ + 1) × π/(δθ×n) − 1 do begin { ρ_max ≈ 1,5 × N }
2.     W := ((l mod κ + h × κ) × cos[t] + (l/κ + m × κ) × sin[t])/1.5
3.     I_P_{φ,⌊W/κ⌋}[t − φ × π/(δθ×n)][⌊W⌋ mod κ]++
      end
     end
     R := R/2
    end
   end
  end
 end_all
```

Algorithm 2 *: Hough Transform of images on the OMP architecture.*

Processors start a communication stage checking for the presence of edges among n pixels of a sub-image, located on the n different modules of a column—information about the location of these pixels is not relevant, since it is common to all processors. Information about the found edges is coded on a single memory word and transmitted through all memory modules of a column (line 1 of the Algorithm 2). The architecture operation mode is changed for the row memory access and each processor collects and decodes the information transmitted by all other processors, consulting all the memory modules in a row. Every time an edge is found each processor computes Eq. 4 for a range of values of θ_l—$\frac{\pi}{\delta\theta \times n}$ different values—(line 2 of the Algorithm 2). The Hough space is distributed in such a way that the resulting ρ values calculated by a processor are locally stored in the row of memory modules accessible to it (see Fig. 5). The accumulators in the Hough space are actualised with no further need of data moving (line 3 of the Algorithm 2).

Supposing an image with $\mathcal{C}$ edge points, the complexity of line 2 of the Algorithm 2 is $\mathcal{O}(\mathcal{C} \times \frac{\pi}{\delta\theta \times n})$ (computation of Eq. 4). The coding and the transfer of information about the edges (line 1 of the Algorithm 2) takes $\mathcal{O}(\frac{N^2}{n})$ more steps. The overall efficiency of the processing for Algorithm 2 is $\mathcal{O}(N^2 \times \frac{\pi}{\delta\theta \times n})$, since $\mathcal{C} < N^2$.

The parallel algorithms proposed for image rotation and Hough transform on an OMP architecture have a processing efficiency of 100%. However, in practice, the time spent in communication limits the performance of the processing systems. The operation of an OMP system was simulated by computer, in order to predict the performance of a real image processing system.

4 Performance Analysis

In this section, the performance of an image processing OMP system is evaluated based on results obtained by computer simulation. The simulations have been done by adopting an algorithm driven approach and by using programming tools developed by the authors [10]. These tools allow the inclusion of detailed information about the operation and characteristics of the target OMP systems in the simulation process. It is therefore possible to achieve simulation results with a great degree of fidelity.

Several parallel image processing systems with different characteristics can be designed around an OMP architecture [4, 9]. The systems can use different types of components and can adopt specific techniques for accelerating the access to the shared memory (*e.g.* interleaved memory access mechanisms). The characteristics of the processing discussed in this paper point to the use of processors able to perform fast floating point operations and with a very short memory access cycle. The control of the image systems should be simple, allowing the implementation of synchronisation barriers and the change of the memory access mode with small waste of time.

An image processing system with the following principal characteristics was simulated: a) the specifications of the individual processors are related to the ones of a signal processor [13]—16 MIPS and 33 MFLOPS approximately, and a cycle time of about 60 ns; b) the number of instruction cycles needed for changing the system memory access mode is considered to be 10; c) read and write memory operations take only 1 instruction cycle; d) simple memory access schemes are considered, without memory interleaving.

The simulations of the OMP system were carried out with the parallel algorithms proposed for the image rotation and the Hough transform. The expected processing times for the OMP system were collected from the simulations and are presented in Tables 1 and 2. Simulations are made with 512×512 images under different conditions that regard: a) the parameters of the processing; b) the number of processors used; c) the features of the images used for the Hough transform. For the simulations with the Hough transform a synthetic image (see fig. 7a) and a real image were used (see fig. 7b), by applying the Marr algorithm for edge detection [3]—application of the Laplacian operator to the image previously filtered with a Gaussian function ($\sigma = 1.5$). The edge detection algorithm uses local processing which is easily mapped onto an OMP architecture [12]. Table 2 also present the expected processing time for the edge detection parallel algorithm.

For each of the different situations, the tables show the processing efficiency ($\mathcal{EF}$) mean and standard deviation for different numbers of processors, within a range between 4 and 32. $\mathcal{EF}$ is defined as the quotient between the time of the sequential processing and the time of the parallel processing multiplied by the number of processors.

Table 1 presents the times spent with Algorithm 1, for the image rotation according to various rotation angles (θ). The efficiency is generally not high, with communication and processing stages taking comparable times. The processing

512 × 512 images; ($i_0 = 256, j_0 = 256$); bilinear interpolation							
$\theta(^\circ)$	execution time (ms)					$\overline{\mathcal{EF}}$ (%)	$\sigma(\mathcal{EF})$ (%)
	1 *Proc.*	4 *Proc.*	8 *Proc.*	16 *Proc.*	32 *Proc.*		
15	918	465	249	150	92	41.2	7.1
45	877	517	299	178	106	34	6.2
60	886	475	264	164	100	37.5	7.3
90	975	451	219	110	57	54.6	0.9
180	975	451	218	109	55	55.3	0.8

Table 1. Performance of the OMP system for image rotation.

times approach the video frame rate when the number of processors increases to 32. The mean value of the efficiency varies with the rotation angles in agreement with the analysis presented in the previous section. Other simulations are made by duplicating the number of pixels considered in each processing step. For the lowest mean value of the processing efficiency ($\theta = 45^o$) an improvement of approximately 5% is achieved, while for the highest mean value ($\theta = 180^o$) an improvement of approximately 7% is achieved.

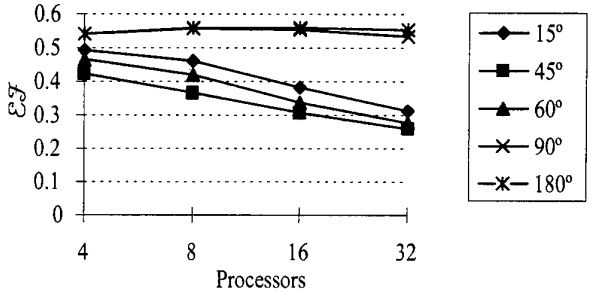


Fig. 6. Processing efficiency for image rotation.

Rotation angles that initially lead to greater processing times give rise to images with regions of pixels that do not have a correspondence with those of the original image. This problem results in a non uniform distribution of the processing load and reinforces the weight of communication times in the total time spent.

Table 2 presents the times spent in the edge detection and the Hough Transform tasks for two different types of images: a synthesised binary image (Fig. 7a) and a real image with 256 grey levels (Fig. 7b shows the edges detected in the real image with the Marr Algorithm for $\sigma = 1.5$). For each image, Algorithm 2 was applied for two different values of $\delta\theta$: $\delta\theta = \pi/512$ and $\delta\theta = \pi/64$.

For the real image, the mean value of the efficiency is relatively high for a low value of $\delta\theta$—a processor has to calculate Eq. 4 a great number of times in each processing stage. Table 2 and Fig. 8 show the decline of the efficiency with the increase of the $\delta\theta$ value. The image of Fig. 7b was synthesised in order to allow the observation of the behaviour of the processing efficiency in a very

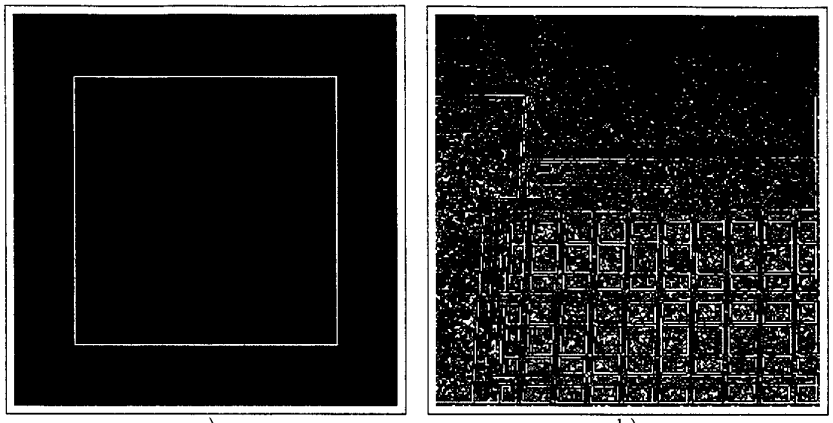


Fig. 7. 512 × 512 binary images used for the Hough transform: a) a synthesised image (binary); b) the edge detected for a real image (Marr algorithm).

Image segmentation–Edge detection Marr Algorithm (11 × 11) (MA); Hough Transform (HT)															
512 × 512 Images	$\delta\theta$	execution time (ms)										$\overline{\mathcal{EF}}$ (%)		$\sigma(\mathcal{EF})$ (%)	
		$1Pr.$		$4Pr.$		$8Pr.$		$16Pr.$		$32Pr.$					
		MA	HT	MA	HT	MA	HT	MA	HT	MA	HT	MA	HT	MA	HT
real	$\pi/512$	13990	7841	3501	2483	1754	1311	883	720	445	402	99.2	70.7	1.7	6.8
fig. 7b)	$\pi/64$	13990	1063	3501	406	1754	250	883	178	445	142	99.2	44.8	1.7	15.9
synthetic	$\pi/512$	–	546	–	235	–	164	–	135	–	122	–	34.7	–	16.7
fig. 7a)	$\pi/64$	–	151	–	124	–	107	–	106	–	107	–	15.3	–	9.9

Table 2. Performance of the OMP system for the image segmentation task using Hough transform.

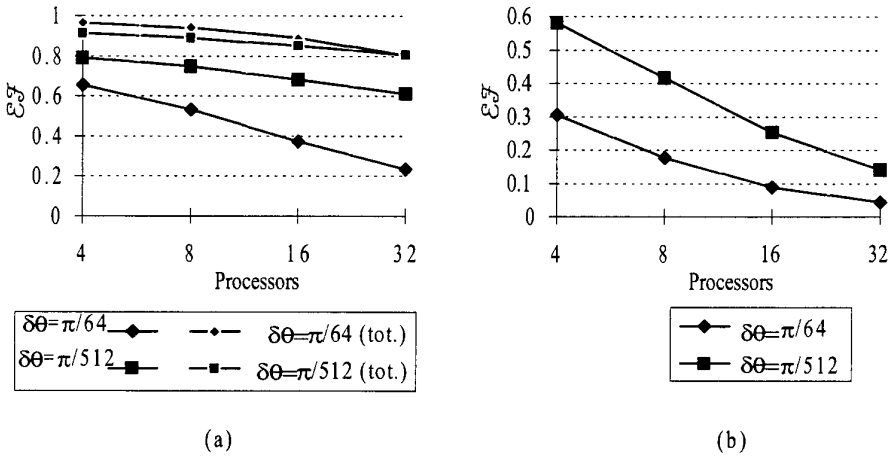


Fig. 8. Processing efficiency for image segmentation (edge detection plus Hough transform: a) for the real image; b) for the synthetic image.

adverse situation. The number of supposed edges is quite small which implies a small processing load when compared to the communication requirements. The mean value of the processing efficiency decreases to approximately half the value obtained for the real image. The decrease of the processing efficiency with the number of processors is also greater than the observed for real images.

5 Conclusions

A method for mapping nonlocal image processing tasks onto OMP systems, using image parallelism, is presented in this paper. No *a priori* knowledge about information exchange requirements is needed. The processing is modelled by a sequence of processing and communication stages. The information is orthogonally transfered or broadcasted during the communication stages in parallel by the multiple processors. The processing is synchronised at the begin and at the end of the stages by implementing synchronisation barriers.

The effectiveness of the method is shown by the development and analysis of parallel algorithms for typical nonlocal image processing, namely for the image rotation and the Hough transform tasks. The analysis of the complexity of the algorithms demonstrates that the processing efficiency achieved is 100%.

The performance of an OMP image processing system is evaluated based on simulation results. The times for the proposed nonlocal image processing parallel algorithms, show that the system can have a good performance if the following aspects are considered: *i*) a fair division of the processing in a stage is almost achieved by distributing equally sized sub-images among the processors—which also means among the memory modules; *ii*) the number of pixels processed in each step must be big enough, in order to reduce the communication time weight in the total processing time; *iii*) the responsibility of transferring the information in a step should be divided by the multiple processors uniformly.

References

1. Dana H. Ballard and Cristopher M. Brown. *Computer Vision.* Prentice-Hall, Londres, 1982.
2. D. Ben-Tzvi, A. Naqvi, and M. Sandler. Synchronous multiprocessor implementation of the Hough transform. *Computer Vision, Graphics, and Image Processing*, 52:437–446, 1990.
3. Robert M. Haralick and Linda G. Shapiro. *Computer and Robot Vision*, volume I. Addison-Wesley, 1992.
4. K. Hwang and D. Kumar Panda. *Parallel Architectures and Algorithms for Image Understanding*, chapter The USC Orthogonal Multiprocessor for Image Understanding, pages 59–94. Academic Press, 1991.
5. Kai Hwang, Ping-Sheng Tseng, and Dongseung Kim. An orthogonal multiprocessor for parallel scientific computations. *IEEE Transactions on Computers*, 38(1):47–61, January 1989.
6. J. Illingworth and J. Kittler. A survey of the Hough transform. *Computer Vision, Graphics, and Image Processing*, 44:87–116, 1988.

7. Azriel Rosenfeld and Avinash C. Kak. *Digital Picture Processing*, volume 2 of *Computer Science and Applied Mathematics*. Academic Press, Londres, segunda edition, 1982.
8. Isaac D. Scherson and Yiming Ma. Analysis and applications of the orthogonal access multiprocessor. *Journal of Parallel and Distributed Computing*, 7(2):232–255, October 1989.
9. Leonel Sousa, José Caeiro, and Moisés Piedade. An advanced architecture for image processing and analysis. In *IEEE International Symposium on Circuits and Systems*, volume 1, pages 77–80. IEEE Singapore Section, The Institute of Electrical and Electronics Engineers, May 1991.
10. Leonel Sousa and Moisés Piedade. Simulation of SIMD and MIMD shared memory architectures on UNIX based systems. In *IEEE International Symposium on Circuits and Systems*, pages 637–640, California, May 1992. IEEE Circuits and Systems Society.
11. Leonel A. Sousa and Moisés Piedade. *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, chapter Low Level Parallel Image Processing. WILEY Series in Parallel Computing. John Wiley & Sons, 1993.
12. Leonel Augusto Sousa. *Parallel Image Processors with Orthogonal Access to Shared Memory*. PhD thesis, Instituto Superior Técnico, Lisboa, 1996. (only available in the portuguese language).
13. Texas Instruments. *TMS320C40x User's Guide*, 1994.
14. S. Yalamanchili and J. K. Aggarwal. Analysis of a model for parallel image processing. *Pattern Recognition*, 18(1):1–16, 1985.

Reconfigurable Mesh Algorithm for Enhanced Median Filter

Byeong-Moon Jeon, Kyu-Yeol Chae, and Chang-Sung Jeong*

Department of Electronics Engineering, Korea University,
Anamdong 5-ka, Sungbuk-ku, Seoul 136-701, Korea
jbm@snoopy.korea.ac.kr, csjeong@chalie.korea.ac.kr

Abstract. In this paper, we derive an enhanced 2-D median filter with adaptive size which can offer a more desirable combination of impulse noise suppression and detail preservation properties. The median filter is developed by applying an enhanced one-dimensional length-decision rule for horizontal, vertical, and two diagonal directions. Then, we present an optimal parallel algorithm for the enhanced median filter on the reconfigurable mesh and prove that the RMESH algorithm is done in $O(w)$ time on $N \times N$ RMESH and this complexity is optimal when the size of an image is $N \times N$ and the filter size is $w \times w$.

1 Introduction

Application of median filter to an image requires some cautions because median filtering tends to remove image details such as thin lines and sharp corners while suppressing noise. In response to these difficulties, several variations of median filters have been proposed[1]-[4]. Generally speaking, the preservation of signal features and the elimination of noise are two contradictory aspects in signal or image processing. The conceivable solution is to vary the size of median filter according to the distribution of impulse noise. This idea can settle the problem which often exhibits blurring for large window size or insufficient noise suppression for small window size. According to this theory, Lin[5] proposed the algorithm based on impulse noise detection employing adaptive-length median filters when images are corrupted by impulse noise. Unfortunately, Lin's filter contains the inherent drawbacks. In this paper, we derive an enhanced one-dimensional length-decision rule as well as an enhanced 2-D median filter with adaptive size, and shall show that they can solve the problems inherent in Lin's filter and provide a more desirable combination of impulse noise suppression and detail preservation properties. From the experimental results, we show that our median filter can achieve the best image quality among all filters considered.

So far, a lot of multiprocessor architectures have been proposed for parallel processing. Of these, a very attractive interconnection model is the two-dimensional mesh connected computer. However, since mesh is a natural and realistic parallel architecture for efficient solution of many problems but solution

* This research is supported by the Brain Korea 21 Project.

times are constrained by long data movements, researchers have studied special architectures whose bus system can be dynamically changed to improve the communication efficiency. If the bus system can be dynamically changed under program control, it is referred to as *reconfigurable*. Several different reconfigurable mesh (RMESH) architectures have been proposed in the literature. These include the polymorphic–torus[6][7], the mesh with reconfigurable buses (MRB)[8], the processor array with a reconfigurable bus system (PARBUS)[9][10], the reconfigurable network (RN)[11][12], and the mesh restriction of reconfigurable network (MRN) [11][13]. Conceptually, these reconfigurable architectures are functionally equivalent. We present a parallel algorithm for the enhanced median filter on the reconfigurable mesh. Besides, we prove that the RMESH algorithm is done in $O(w)$ time on $N \times N$ RMESH and this complexity is optimal when the size of an image is $N \times N$ and the filter size is $w \times w$.

The organization of this paper is as follows. In Section 2, we briefly review the problems of Lin's filter and derive the enhanced one–dimensional length–decision rule and the enhanced 2–D median filter with adaptive size. The performance of our median filter is evaluated and compared with other filters' by applying the proposed method to test images. In Section 3, we provide some preliminaries related to our parallel algorithm, and present the RMESH algorithm for the enhanced median filter. Finally, in Section 4, we give concluding remarks.

2 Enhanced Median Filter

Let us consider 1–D adaptive–length median filter. If the length of 1–D window varies adaptively according to the width of impulse noise, the median filter will eliminate noise efficiently while preserving more signal features.

Property 1. When the maximum length of 1–D window is w, the median filter with window length $2K+1$ should be applied to remove impulse noise of width K, $1 \leq K \leq \lfloor \frac{w}{2} \rfloor$.

For removing impulse noise in two–dimensional images, 2–D adaptive median filter can be obtained from expanding 1–D adaptive median filter. Since edge preservation is essential in image processing, it is important to take into account structural information of the image. In this paper, we assume that horizontal, vertical, and two diagonal (45–degree and 135–degree) lines in the window are to be preserved. Thus, such a modification of 2–D median filter leads to 1-D median filters used in four directions.

Property 2. 2–D median filter with adaptive size is derived by expanding 1–D median filter with adaptive length for horizontal, vertical, and two diagonal (45–degree and 135–degree) directions.

2.1 Lin's Filter

Based on Property 1 and 2, Lin proposed the adaptive window-size median filter to suppress noise effectively and at the same time preserve image details[5]. However, his median filtering algorithm has several problems awaiting solution, where the maximum window size is 5×5.

The first problem of Lin's filter is that the window length determined from the width of mixed impulse noise (positive and negative impulse noise) does not correspond to the desirable length conduced from Property 1. If the pixels inside 1-D window are denoted as $x(n-2), x(n-1), x(n), x(n+1)$, and $x(n+2)$, Lin defined the difference between neighboring pixels as follows,

$$p_i = x(n+i-1) - x(n+i) \qquad for \;\; i = 1, 2$$
$$p_i = x(n+i+1) - x(n+i) \qquad for \;\; i = -1, -2$$

And Lin discriminated strictly between positive and negative impulse noise by signs of p_i's. When the pixels within a window are corrupted by both positive and negative impulse noise, one must first apply the algorithm for positive impulse noise and then apply the algorithm for negative impulse noise. This process has a weakness requiring a lot of time. Besides, though the one-dimensional length-decision algorithm can remove mixed impulse noise, it violates Property 1. For instance, Fig. 1 shows the mixed impulse noise of width 2. Since $p_{-2} > 0$, $p_{-1} < 0$, and $p_1 < 0$, Lin's algorithm allows the window length 3. However, in fact, the desirable window length is 5 because the mixed impulse noise has width 2.

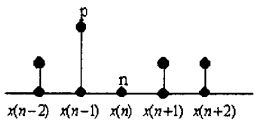


Fig. 1. Mixed impulse noise of width 2 (p: positive impulse noise, n : negative impulse noise)

The second problem is that the condition related to the impulse noise of width 1 is not proper. That is, Lin proposed that the following condition could detect impulse noise of width 1 and the window length would be three.

$$p_1 > T \textbf{ and } p_{-1} > T$$

where T is a threshold value. However, this condition includes the case that the current pixel $x(n)$ is misjudged as positive impulse noise, though $x(n)$ is not corrupted noise. As shown in Fig. 2, when $x(n)$ is original and $x(n-1)$ and $x(n+1)$ are negative impulse noise, the above condition is satisfied. Thus, the condition has a problem which can distort the attribute of $x(n)$.

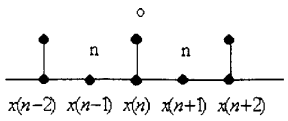


Fig. 2. Case that the center pixel is not corrupted by noise (n : negative impulse noise, o : original pixel)

Finally, in the 2–D adaptive median filtering algorithm, the maximum length in two diagonal directions is limited to three. This causes that the abilities of noise suppression and preserving image details are decreased in two diagonal directions.

2.2 Enhanced 2–D Median Filter with Adaptive Size

In order to easily derive the enhanced one–dimensional length–decision rule, we limit the maximum length of 1–D adaptive median filter to 5 in the rest of the paper. Naturally, the maximum size of 2–D adaptive median filter is 5×5. According to Property 1, we can suppress impulse noise whose maximum width is 2. Positive or negative impulse noise is defined as a pixel which is larger or smaller than both neighbors by T, where T is a threshold value and we determine it as 45. And the difference between neighboring pixels is defined as follows.

$$diff(i) = |x(n+i-1) - x(n+i)| \qquad for \;\; i = 1, 2$$
$$diff(i) = |x(n+i+1) - x(n+i)| \qquad for \;\; i = -1, -2$$

At first, to overcome the first problem of Lin's filter, we use new threshold values, T_{min} and T_{max}, that detect the width of mixed impulse noise exactly, where T_{min} is used to decide if the center pixel and its neighbor are impulse noise of the same types and T_{max} is used to decide if they are impulse noise of different types.

Theorem 1. *Impulse noise of width 2 can be suppressed by the median filter with window length 5 if either of the following conditions is satisfied.*

$$diff(1) > T \textbf{ and } (diff(-1) < T_{min} \textbf{ or } diff(-1) > T_{max}) \textbf{ and } diff(-2) > T \tag{1}$$

or

$$diff(-1) > T \textbf{ and } (diff(1) < T_{min} \textbf{ or } diff(1) > T_{max}) \textbf{ and } diff(2) > T \tag{2}$$

Fig. 3 (a)–(h) show the cases that the width of impulse noise is 2. Among them, (a)–(d) show that $x(n-1)$ and $x(n)$ are impulse noise, and condition (1) can be

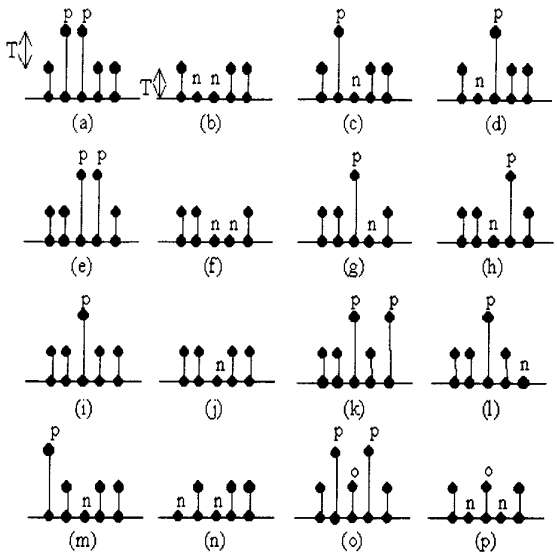


Fig. 3. The mixed impulse noise in 1-Dimensional window length 5 (p : positive impulse noise, n : negative impulse noise, o : original pixel)

applied in these cases. If there is little difference between $x(n-1)$ and $x(n)$, that is, $x(n-1)$ and $x(n)$ are impulse noise of the same types, $diff(-1) < T_{min}$ is satisfied. If there is a wide difference between $x(n-1)$ and $x(n)$, $diff(-1) > T_{max}$ is satisfied, and this means that $x(n-1)$ and $x(n)$ are impulse noise of different types (see Fig. 3 (c) and (d)). And $diff(1) > T$ or $diff(-2) > T$ is used to define the boundary between $x(n)$ and $x(n+1)$ or between $x(n-2)$ and $x(n-1)$. As for the threshold values T_{min} and T_{max}, we tried experimentally various values and obtained 5 and 100, respectively. In a manner similar to condition (1), it is easy to derive condition (2) from Fig. 3(e)–(h).

The second problem of Lin's filter can cause to destroy the original pixel. To avoid the problem, we consider a new condition.

Theorem 2. *Impulse noise of width 1 can be suppressed by the median filter with window length 3 if either of the following conditions is satisfied.*

$$diff(1) > T \textbf{ and } diff(-1) > T \textbf{ and } (diff(-2) < T \textbf{ or } diff(2) < T) \quad (3)$$

or

$$((diff(-1) < T \textbf{ and } diff(1) > T) \textbf{ or } (diff(-1) > T \textbf{ and } diff(1) < T)) \textbf{ and } \frac{1}{2} < \frac{diff(1)}{diff(-1)} < 2 \quad (4)$$

$diff(1) > T$ **and** $diff(-1) > T$ of condition (3) guarantees that the attribute of center pixel $x(n)$ is different from those of its neighbors $x(n+1)$ and $x(n-1)$. Similarly, $diff(-2) < T$ **or** $diff(2) < T$ guarantees that the attributes of $x(n-2)$ and $x(n-1)$ are the same or those of $x(n+1)$ and $x(n+2)$ are the same. At this point, we need to pay attention to the fact that logical operation **or** is used. This means that if condition (3) is satisfied, the group composed of $x(n-2)$ and $x(n-1)$ or the group composed of $x(n+1)$ and $x(n+2)$ must have the same attribute and $x(n)$ must be different from one of two groups. Thus, Fig. 3 (i)–(n) satisfy condition (3) but (o) and (p) do not satisfy. For the condition (4), ($diff(-1) < T$ **and** $diff(1) > T$) **or** ($diff(-1) > T$ **and** $diff(1) < T$) shows that the current pixel is located at the boundary between two regions. And $\frac{diff(1)}{diff(-1)}$ is used for detecting impulse noise with low amplitude.

If the current pixel is original, it is natural that the pixel is not filtered. So we need to examine whether or not the current pixel is impulse noise.

Theorem 3. *The center pixel of window is not filtered if the following condition is satisfied.*

$$diff(-2) < T \textbf{ and } diff(-1) < T \textbf{ and } diff(1) < T \textbf{ and } diff(2) < T \quad (5)$$

Since an image consists of many features which are very valuable for human vision, we need to preserve important image features. According to Property 2, our 2–D median filter adapts the window size by applying the enhanced one–dimensional length–decision rule (i.e., conditions (1)–(5)) for four directions. As mentioned above, Lin's filter has the problem that the maximum length in two diagonal directions is limited to 3 when the maximum size of 2–D window is 5×5. In our 2–D adaptive median filtering algorithm, the same maximum length 5 is used for each direction. This can produce the more improved performance than Lin's filter. The current pixel satisfying condition (5) in one direction must not be filtered for the other directions in our algorithm, because it is not noise and hence must not be changed. However, since condition (5) is the necessary condition for the fact that the current pixel becomes original, there is some cases that the current pixel does not satisfy condition (5) even if it is original. Such the pixel also satisfies neither of condition (3) and (4). In these cases, we do not decide in current direction whether or not the center pixel is noise and examine it in another directions. Our 2–D adaptive median filter with 5×5 window is described as follows.

Algorithm Enhanced_Median_Filter
Input : An image with mixed impulse noise
Output: A noise–removed image
Step 1. Do the following for each direction : horizontal, vertical, 45–degree, and 135–degree directions
- **(1.1)** If either condition (1) or (2) is satisfied, decide window length 5 and go to Step 1.
- **(1.2)** If either condition (3) or (4) is satisfied, decide window length 3 and go to Step 1.

(1.3) If condition (5) is satisfied, the center pixel remains unchanged and go to Step 3.

(1.4) If none of above conditions is satisfied, decide window length 1 and go to Step 1.

Step 2. Sort data samples according to the adaptive lengths of four directions and replace current pixel with median value.

Step 3. Move 5×5 window to next pixel for successive filtering.

2.3 Experimental Results

In this subsection, the algorithm *Enhanced_Median_Filter* is evaluated by applying it to noisy images and compared with other existing noise removal techniques. In order to compare the performance of our 2–D adaptive median filter with those of other filters[1][2][3][4][5], we apply these filters to the images corrupted by impulse noise. And we compare their efficiency in noise suppression and detail–preserving characteristics by SNR value. Two 512×512 test images, 'Lena' and 'San Francisco', are degraded by only impulse noise with probability of an impulse occurring, $p = 0.1$ and $p = 0.05$.

Table 1. Comparative results in SNR for various probabilities of impulse noise

Filter Type (5×5)	Lena		San Francisco	
	$p = 0.1$	$p = 0.05$	$p = 0.1$	$p = 0.05$
Median filter	32.32 dB	32.55 dB	25.81 dB	25.86 dB
Recursive median filter[1]	34.06 dB	34.59 dB	26.53 dB	26.63 dB
CWM filter[2]	34.84 dB	36.91 dB	29.58 dB	30.24 dB
ACWM filter[3]	36.05 dB	37.29 dB	28.44 dB	31.58 dB
Optimal stack filter[4]	33.22 dB	33.60 dB	26.07 dB	26.15 dB
Lin's filter[5]	38.32 dB	41.05 dB	30.88 dB	32.48 dB
Our median filter	39.40 dB	42.39 dB	31.83 dB	34.24 dB

Table 1 lists the SNR values of the filtered images. In this table, the optimal stack filter is originally designed under the certain structural constraints[4]. That is, we adopted the same structural constraints as our 2–D adaptive median filter. The performance of Lin's filter shows higher comparing to the conventional filtering methods. However, due to the problems indicated in this section, the performance of it is worse than our median filter with adaptive size. As expected, our median filter provides better performance than the other filters in removing impulse noise.

3 Parallel Algorithm on Reconfigurable Mesh

The reconfiguration mesh architecture used in this paper is based on the architecture MRN defined in [11][13]. $N_1 \times N_2$ RMESH consists of an $N_1 \times N_2$ array of

processing elements (PEs) which are connected to a grid–shaped reconfigurable bus system as shown in Fig. 4 (a). As usual, it is assumed that every processor knows its own coordinates. And PE(i, j) is connected to its four neighbors PE$(i-1, j)$, PE$(i+1, j)$, PE$(i, j-1)$, and PE$(i, j+1)$, provided they exist. Every processor has four ports denoted by N (North), S (South), E (East), and W (West). Internal connection among four ports of a processor can be configured during the execution of algorithms. We use the notation $\{g\}$ to represent the local connections within a processor, where g denotes a group of ports that are connected together within the processor. For example, {SW, NE} represents the configuration in which S port is connected to W port while N port is connected to E port. Our computation model allows at most two connections to be set in each processor at any one time. Furthermore, these two connections must involve disjoint pairs of ports as illustrated in Fig. 4 (b). Therefore, the number of possible connection patterns in each processor is 10. In a single unit of time, each processor can perform basic arithmetic and logic operations on its own data and can connect or disconnect its local connections among four ports.

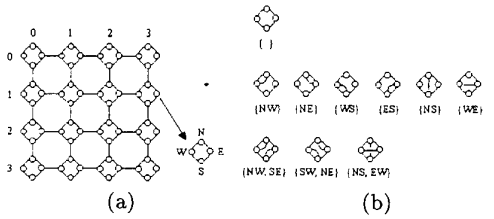


Fig. 4. 4×4 RMESH and its connection patterns

Note that by setting the local connections properly, processors that are attached to the same subbus can communicate with one another by broadcasting values on the common subbus. The RMESH model of this paper allows several processors to read the same bus component; however, it does not allow more than one processor to write on the same bus component at the same time, i.e., a concurrent–read exclusive–write model. If no value is being transmitted on the subbus, the read operation has no result. It is assumed that communications along buses take $O(1)$ time. Although inexact, recent experiments with the YUPPIE[6][7] and the PPA[14][15] reconfigurable multiprocessor systems seem to indicate that this approximation is a reasonable working hypothesis.

3.1 Basic Operations

The purpose of this subsection is to provide several procedures for the reconfigurable mesh that will be used in the design of our RMESH algorithm.

We assume that PE(i,j) initially holds a value $I(i,j)$, $0 \leq i,j \leq N-1$. Each PE is required to accumulate s items of I value in its array A as specified

$$A[q](i,j) \;=\; I(i,(j-\lfloor\frac{s}{2}\rfloor+q)\ mod\ N)$$

where $\lfloor\frac{s}{2}\rfloor \leq j < N-\lfloor\frac{s}{2}\rfloor$ and $0 \leq q < s$. In this operation, we do not apply accumulate operation in the border of RMESH within the range of $\lfloor\frac{s}{2}\rfloor$ length. The procedure for this is as follows.

Procedure Accumulate1(A,s,I)
for $k := 0$ **to** $\lfloor\frac{s}{2}\rfloor$ **do**
for all i,j $(0 \leq i,j \leq N-1)$ **in parallel**
1) PE(i,j) connects ports E and W;
2) PE(i,j) disconnects the horizontal bus if $j\ mod\ (\lfloor\frac{s}{2}\rfloor+1) = k$;
3) PE(i,j) broadcasts $I(i,j)$ through E port and $A[(k+(\lfloor\frac{s}{2}\rfloor+1)-(j\ mod\ (\lfloor\frac{s}{2}\rfloor+1)))\ mod\ (\lfloor\frac{s}{2}\rfloor+1)](i,j) := I(i,j)$ if $j\ mod\ (\lfloor\frac{s}{2}\rfloor+1) = (k+1)\ mod\ (\lfloor\frac{s}{2}\rfloor+1)$;
4) PE(i,j) receives the value from W port and stores it into $A[(k+(\lfloor\frac{s}{2}\rfloor+1)-(j\ mod\ (\lfloor\frac{s}{2}\rfloor+1)))\ mod\ (\lfloor\frac{s}{2}\rfloor+1)](i,j)$;
5) PE(i,j) broadcasts $I(i,j)$ through W port if $j\ mod\ (\lfloor\frac{s}{2}\rfloor+1) = (k-1)\ mod\ (\lfloor\frac{s}{2}\rfloor+1)$;
6) PE(i,j) receives the value from E port and stores it into $A[((k-1)\ mod\ (\lfloor\frac{s}{2}\rfloor+1)+(\lfloor\frac{s}{2}\rfloor+1)-(j\ mod\ (\lfloor\frac{s}{2}\rfloor+1)))\ mod\ (\lfloor\frac{s}{2}\rfloor+1)+\lfloor\frac{s}{2}\rfloor](i,j)$;
endfor;
endfor;

An illustration of how this procedure works is provided in Fig. 5, where the size of RMESH is 8×8 and $s = 5$. Based on this figure, we can easily see that the procedure *Accumulate*1(A,s,I) correctly accumulates s items of I value in $O(s)$ time on any row of $N \times N$ RMESH. On the other hand, the following operation can be done in a similar way of the procedure *Accumulate*1(A,s,I).

$$A[q](i,j) \;=\; I((i-\lfloor\frac{s}{2}\rfloor+q)\ mod\ N,j)$$

where $\lfloor\frac{s}{2}\rfloor \leq i < N-\lfloor\frac{s}{2}\rfloor$ and $0 \leq q < s$. As we can see, there is no difference between two operations except the direction along which the values are accumulated. Therefore, an $O(s)$ procedure for this operation can be derived by just adjusting the internal connection among four ports of each PE.

Procedure Accumulate2(A,s,I)
for $k := 0$ **to** $\lfloor\frac{s}{2}\rfloor$ **do**
for all i,j $(0 \leq i,j \leq N-1)$ **in parallel**
1) PE(i,j) connects ports N and S;
2) PE(i,j) disconnects the vertical bus if $i\ mod\ (\lfloor\frac{s}{2}\rfloor+1) = k$;
3) PE(i,j) broadcasts $I(i,j)$ through S port and $A[(k+(\lfloor\frac{s}{2}\rfloor+1)-(i\ mod\ (\lfloor\frac{s}{2}\rfloor+1)))\ mod\ (\lfloor\frac{s}{2}\rfloor+1)](i,j) := I(i,j)$ if $i\ mod\ (\lfloor\frac{s}{2}\rfloor+1) = (k+1)\ mod\ (\lfloor\frac{s}{2}\rfloor+1)$;

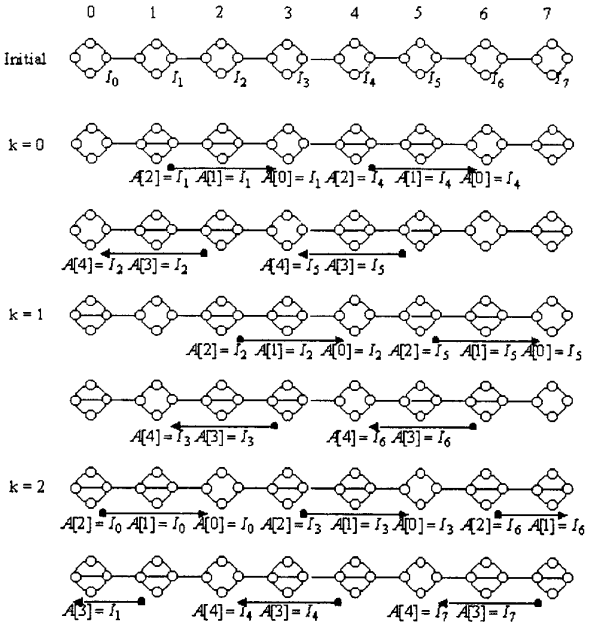


Fig. 5. Example of $Accumulate1(A, s, I)$

4) PE(i, j) receives the value from N port and stores it into $A[(k+(\lfloor \frac{s}{2} \rfloor + 1) - (i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1))) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1)](i, j)$;

5) PE(i, j) broadcasts $I(i, j)$ through N port if $i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) = (k - 1) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1)$;

6) PE(i, j) receives the value from S port and stores it into $A[((k - 1) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) + (\lfloor \frac{s}{2} \rfloor + 1) - (i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1))) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) + \lfloor \frac{s}{2} \rfloor](i, j)$;

endfor;

endfor;

Next, we show how to accumulate s items of I value along two diagonal directions of RMESH. These operations are defined as follows.

$$A[q](i,j) = I((i - \lfloor \tfrac{s}{2} \rfloor + q) \ mod \ N, (j - \lfloor \tfrac{s}{2} \rfloor + q) \ mod \ N)$$
$$A[q](i,j) = I((i - \lfloor \tfrac{s}{2} \rfloor + q) \ mod \ N, (j + \lfloor \tfrac{s}{2} \rfloor + q) \ mod \ N)$$

where $\lfloor \frac{s}{2} \rfloor \leq i, j < N - \lfloor \frac{s}{2} \rfloor$ and $0 \leq q < s$. They involve establishing a number of subbuses and broadcasting values along them. The details of these operations are spelled out in the following procedures, $Accumulate3(A, s, I)$ and $Accumulate4(A, s, I)$.

Procedure Accumulate3(A, s, I)
for $k := 0$ **to** $\lfloor \frac{s}{2} \rfloor$ **do**
for all i, j $(0 \leq i, j \leq N-1)$ **in parallel**
1) PE(i,j) establishes the local connection {WS, NE};
2) PE(i,j) disconnects ports W and S if $i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) = k$;
3) PE(i,j) broadcasts $I(i,j)$ through S port and $A[(k + (\lfloor \frac{s}{2} \rfloor + 1) - (i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1))) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1)](i,j) := I(i,j)$ if $i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) = (k+1) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1)$;
4) PE(i,j) receives the value from W port and stores it into $A[(k + (\lfloor \frac{s}{2} \rfloor + 1) - (i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1))) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1)](i,j)$;
5) PE(i,j) broadcasts $I(i,j)$ through N port if $i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) = k$;
6) PE(i,j) receives the value from E port and stores it into $A[((k-1) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) + (\lfloor \frac{s}{2} \rfloor + 1) - (i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1))) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) + \lfloor \frac{s}{2} \rfloor](i,j)$;
endfor;
endfor;

Procedure Accumulate4(A, s, I)
for $k := 0$ **to** $\lfloor \frac{s}{2} \rfloor$ **do**
for all i, j $(0 \leq i, j \leq N-1)$ **in parallel**
1) PE(i,j) establishes the local connection {NW, ES};
2) PE(i,j) disconnects ports E and S if $i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) = k$;
3) PE(i,j) broadcasts $I(i,j)$ through S port and $A[(k + (\lfloor \frac{s}{2} \rfloor + 1) - (i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1))) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1)](i,j) := I(i,j)$ if $i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) = (k+1) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1)$;
4) PE(i,j) receives the value from E port and stores it into $A[(k+(\lfloor \frac{s}{2} \rfloor + 1) - (i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1))) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1)](i,j)$;
5) PE(i,j) broadcasts $I(i,j)$ through N port if $i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) = k$;
6) PE(i,j) receives the value from W port and stores it into $A[((k-1) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) + (\lfloor \frac{s}{2} \rfloor + 1) - (i \ mod \ (\lfloor \frac{s}{2} \rfloor + 1))) \ mod \ (\lfloor \frac{s}{2} \rfloor + 1) + \lfloor \frac{s}{2} \rfloor](i,j)$;
endfor;
endfor;

Similarly, *Accumulate*3(A, s, I) and *Accumulate*4(A, s, I) correctly accumulate s items of I value in $O(s)$ times on $N \times N$ RMESH.

Given a sequence A of n elements and an integer k, where $A = \{a_1, a_2, \cdots, a_n\}$ and $1 \leq k \leq n$, it is required to determine the kth smallest element in A. This is known as the selection problem. *Selection*(A, k) can be performed in $O(n)$ time on a single processor.

3.2 Parallel Enhanced Median Filter

In this subsection, we present the RMESH algorithm for the enhanced 2–D median filter with adaptive size. Let $I(0...N-1, 0...N-1)$ be an $N \times N$ image with $I(i,j)$ being the gray value of the pixel (i,j). We assume that initially,

$N \times N$ image is mapped on the RMESH with $N \times N$ size such that PE(i,j) holds $I(i,j)$, and the window size is $w \times w$.

Since our median filter adapts the window size by applying the enhanced one-dimensional length-decision rule for horizontal, vertical, and two diagonal directions, every processor must know all the image pixels of four directions within the range of the window centered at it. Hence, every processor needs to communicate with other processors along four directions which will be observed in the window. Such communication can be achieved by using the procedures $Accumulate1, Accumulate2, Accumulate3$, and $Accumulate4$. At this time, every processor PE(i,j) accumulates the data received from horizontal direction into $hor[0,1,\cdots,w-1](i,j)$, the data received from vertical direction into $ver[0,1,\cdots,w-1](i,j)$, and the data received from two diagonal directions into $diag1[0,1,\cdots,w-1](i,j)$ and $diag2[0,1,\cdots,w-1](i,j)$, respectively. In addition, while accumulating the image pixels, each processor simultaneously calculates the difference between neighboring pixels. Our RMESH algorithm consists of the following sequence of steps.

Parallel Algorithm Parallel_Enhanced_Median_Filter

{horizontal direction}

Step 1. Every processor PE(i,j) accumulates the image pixels in horizontal direction by calling the procedure $Accumulate1(hor, w, I)$.

Step 2. Every processor PE(i,j) applies the enhanced one-dimensional length-decision rule and decides the appropriate window length, l_{hor}. And PE(i,j) extracts the elements corresponding to l_{hor} from the set hor which are candidates for being median value in Step 10 and includes them into the set *sample* which is initially the null set.

{vertical direction}

Step 3. Every processor PE(i,j) accumulates the image pixels in vertical direction by calling the procedure $Accumulate2(ver, w, I)$.

Step 4. Every processor PE(i,j) applies the enhanced one-dimensional length-decision rule and decides the appropriate window length, l_{ver}. And PE(i,j) extracts the elements corresponding to l_{ver} from the set ver and includes them into the set *sample*.

{45-degree diagonal direction}

Step 5. Every processor PE(i,j) accumulates the image pixels in 45-degree diagonal direction by calling the procedure $Accumulate3(diag1, w, I)$.

Step 6. Every processor PE(i,j) applies the enhanced one-dimensional length-decision rule and decides the appropriate window length, l_{diag1}. And PE(i,j) extracts the elements corresponding to l_{diag1} from the set $diag1$ and includes them into the set *sample*.

{135-degree diagonal direction}

Step 7. Every processor PE(i,j) accumulates the image pixels in 135-degree diagonal direction by calling the procedure $Accumulate4(diag2, w, I)$.

Step 8. Every processor PE(i,j) applies the enhanced one-dimensional length-decision rule and decides the appropriate window length, l_{diag2}. And PE(i,j) extracts the elements corresponding to l_{diag2} from the set $diag2$ and includes them into the set *sample*.

{median filtering with adaptive size}
Step 9. Every processor PE(i,j) calculates $l_{tot} = l_{hor} + l_{ver} + l_{diag1} + l_{diag2}$.
Step 10. Every processor PE(i,j) replaces $I(i,j)$ with median value by calling the procedure $Selection(sample, \lceil \frac{l_{tot}}{2} \rceil)$.

In the RMESH algorithm, we get the window length for each direction adaptively by applying our one–dimensional length–decision rule. According to each window length, we obtain the elements at each direction that practically take part in median operation and store them into the set *sample* that will be input of the procedure *Selection*. Therefore, after Step 8 is done, every processor becomes to know the window lengths of four directions and the set *sample*. Now, each processor must determine the median element in *sample* whose size is $l_{hor}+l_{ver}+l_{diag1}+l_{diag2}$. This operation can be achieved by calling the procedure *Selection*, where the procedure's inputs are *sample* and $\lceil \frac{l_{hor}+l_{ver}+l_{diag1}+l_{diag2}}{2} \rceil$. Because each of the above steps can be implemented in $O(w)$ time, we have the following theorem.

Theorem 4. *When the size of an image is $N \times N$ and the window size is $w \times w$, the parallel algorithm for the enhanced 2–D median filter with adaptive size is done in $O(w)$ time on $N \times N$ RMESH.*

Let $\Omega(T(n))$ be a lower bound on the number of sequential steps required to solve a problem of size n. Then $\Omega(T(n)/N)$ is a lower bound on the running time of any parallel algorithm that uses N processors to solve that problem. According to Theorem 4, the running time of our algorithm is $O(w)$ and the number of processors used is $N \times N$. Therefore, the cost of the algorithm *Parallel_Enhanced_Median_Filter* is optimal since the lower bound on the number of sequential steps is $O(N^2 w)$ for the median filter.

4 Conclusion

We have derived the enhanced 2–D median filter with adaptive size that can solve the problems of Lin's filter and so remove impulse noise effectively in images. The enhanced 2–D median filter is developed by applying our one–dimensional length–decision rule for horizontal, vertical, and two diagonal directions. Our experiments have shown that the proposed methods improve the performance over a number of well–known techniques.

The parallel model with a fixed topology leads to an inevitable tradeoff between the need for low network diameter and the need to limit the number of interprocessor communication links. One method for providing efficient and flexible communication among the processors is based on the concept of network reconfiguration. The most significant advantage of reconfigurable architecture is the flexibility of forming special topologies dynamically as required by the problem. As a result, this can decrease the communication diameter and reduce the bottleneck for designing efficient algorithm. In this paper, the RMESH parallel algorithm has been presented for the enhanced 2–D median filter with adaptive

size. When the size of an image is $N \times N$ and the window size is $w \times w$, our RMESH algorithm is done in $O(w)$ time on $N \times N$ RMESH. Specifically, we have proven that our RMESH algorithm is optimal by comparing its cost with the lower bound on the number of sequential operations.

References

1. G. Qiu, "An Improved Recursive Median Filtering Scheme for Image Processing," *IEEE Trans. Image Processing*, vol. 5, no. 4, pp. 646–648, 1996.
2. S. J. Ko and Y. H. Lee, "Center Weighted Median Filters and Their Applications to Image Enhancement," *IEEE Trans. Circuits and Systems*, vol. 38, no. 9, pp. 984–993, 1991.
3. B. M. Jeon, K. Y. Chai, and C. S. Joeng, "ACWM(Adaptive Center Weighted Median) Filters to Reduce Impulse Noise," *Proc. 24th KISS(Korea Information Science Society) Conf.*, pp. 142–145, 1997.
4. Lin Yin, "Stack Filter Design : A Structural Approach," *IEEE Trans. Signal Processing*, vol. 43, no. 4, pp. 831–840, 1995.
5. Ho-Ming Lin, "Median Filters with Adaptive Length," *IEEE Trans. Circuits and Systems*, vol. 35, no. 6, pp. 675–690, 1988.
6. H. Li and M. Maresca, "Polymorphic-Torus Network," *IEEE Trans. Computers*, vol. 38, no. 9, pp. 1345–1351, 1989.
7. M. Maresca and H. Li, "Connection Autonomy in SIMD Computers : A VLSI Implementation," *J. Parallel Distrib. Computing*, vol. 7, pp. 302–320, 1989.
8. R. Miller, V. K. P. Kumar, D. I. Resis and Q. F. Stout, "Parallel Computations on Reconfigurable Meshes," *IEEE Trans. Computers*, vol. 42, no. 6, pp. 678–692, 1993.
9. B. F. Wang and G. H. Chen, "Constant Time Algorithms for the Transitive Closure and Some Related Graph Problems on Processor Arrays with Reconfigurable Bus Systems," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 1, no. 4, pp. 500–507, 1990.
10. S. S. Lin, "Constant-Time Algorithms for the Channel Assignment Problem on Processor Arrays with Reconfigurable Bus Systems," *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 13, no. 7, pp. 884–890, 1994.
11. Y. Ben-Asher, D. Peleg, and A. Schuster, "The Power of Reconfiguration," *J. Parallel Distrib. Computing*, vol. 13, pp. 139–153, 1991.
12. J. W. Jang and V. K. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *Proc. 6th Int. Parallel Processing Symposium*, pp. 130–137, 1992.
13. J. W. Jang, C. H. Park, and V. K. Prasanna, "A Fast Algorithm for Computing a Histogram on Reconfigurable Mesh," *IEEE Trans. PAMI*, vol. 17, no. 2, pp. 97–106, 1995.
14. M. Maresca, "Polymorphic Processor Arrays," *IEEE Trans. Parallel and Distrib. Syst.*, vol. 4, no. 5, pp. 490–506, 1993.
15. M. Maresca, H. Li, and P. Baglietto, "Hardware Support for Fast Reconfigurability in Processor Arrays," *Proc. Int. Conf. Parallel Processing*, pp. 282–289, 1993.

Parallel Implementation of a Track Recognition System Using Hough Transform

Augusto Cesar Heluy Dantas, José Manoel de Seixas, and
Felipe Maia Galvão França

COPPE/EE/UFRJ
C.P. 68504, Rio de Janeiro 21945-970, Brazil
augusto@lps.ufrj.br, seixas@lps.ufrj.br, felipe@cos.ufrj.br

Abstract. The reconstruction of tracks left by particles in a scintillating fibre detector from a high energy collider experiment is discussed. The track reconstruction algorithm is based on using the Hough transform, and achieves an efficiency above 86%. The algorithm is implemented in a 16-node parallel machine using two parallelism approaches in order to speed up the application of the Hough transform, which is known from its large computational cost.

1 Introduction

In modern high-energy particle collider experiments the *trackers* play an important role. Their task is to reconstruct the tracks left in the detectors by reactions resulting from particle beam collisions. This is a very important task, as it allows the computation of the momentum of particles.

At CERN ([1]), the European Laboratory for Particle Physics, LEP (*Large Electron-Positron Collider*) collides electrons and positrons at four detection points, and the resulting reactionss are observed by a set of sub-detectors that are placed around such collision points. In our case, we focus on the SFT (*Scintillating Fibre Tracker*), a tracker that has been developed to operate at L3, one of the four detectors present at LEP.

The structure of the SFT can be seen in Fig. 1. The SFT has two shells, placed 0.187 m and 0.314 m away from the collision axis. These shells are composed of very thin scintillating fibres with 60 μm of diameter, arranged in groups of 1000 fibres. Each shell has four sublayers of 2 mm: two of them (ϕ_1 and ϕ_2) provide the coordinates in the $r\phi$ plane, and the other two (u and v) furnishes, through stereo vision, the coordinates in the rz plane. One may note that the tracks are only observed in such two small regions, which have a large gap between them.

As the resulting subparticles reach the tracker shells, fibres get excited, producing light and transmiting it towards the light detectors, which convert light into an electrical signal. The light detectors are image tubes made of silicon chips, which have the task of recording the position at which the tracker was reached by a particle. Fig. 2 illustrates this process. These pixelchips function similar to CCDs (Charge Capacitor Devices).

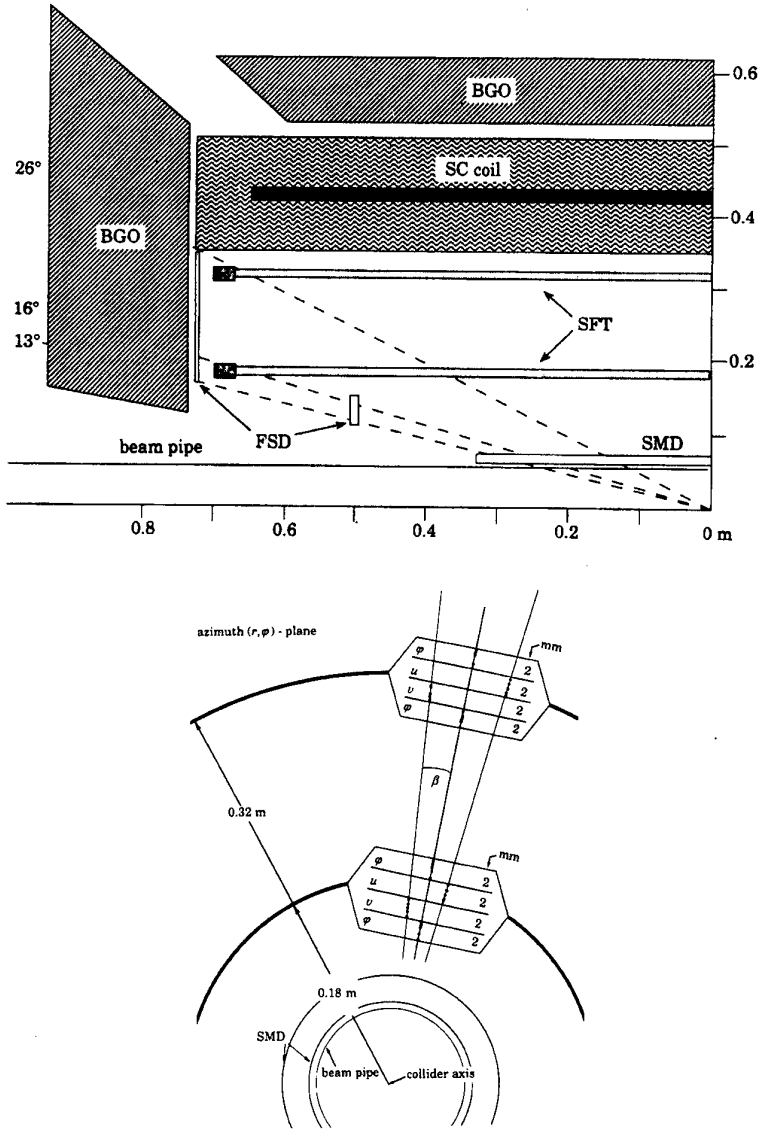


Fig. 1. rz-plane of SFT (top) and the arrangement of the scintilatting fibre shells (bottom).

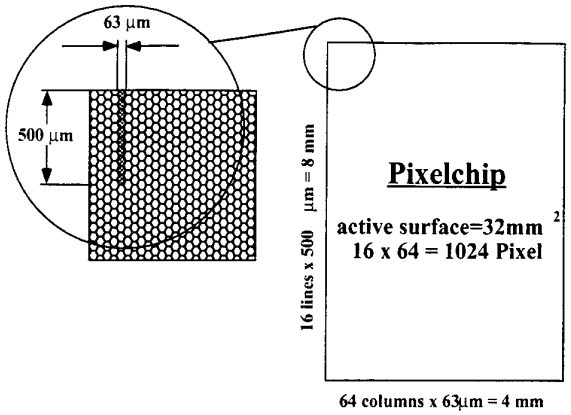


Fig. 2. Recording the positions of interactions between particles and the tracker.

Data used in this work were generated by Monte Carlo simulation ([5]). The software package PYTHIA 5.6/JETSET 7.3 ([2]) was used for generating electron-positron collisions at LEP conditions, and the simulation of the SFT structure was made by using the software GEANT 3.159 ([4]). The generated events correspond to 2D images[1] formed by pixels that are produced on the tracker due to particle interactions with the detector. A typical event to be reconstructed is shown in Fig. 3. Tracks consist of helices, totally described by their two parameters: curvature κ and angle θ. Low energy tracks are removed from the original image[2], as they do not represent tracks of interest of the envisaged physics.

In this paper we propose to use the Hough transform ([3]) to reconstruct the tracks of the SFT. The Hough transform is known to be a powerful technique for track reconstruction, although its application is limited by its highly intensive computational requirements. Therefore, envisaging to speed up the reconstruction procedure, we exploit the parallelism in the Hough transform and implement the track reconstruction algorithm on a 16-node parallel machine.

In the following two sections, the fundaments of the Hough Transform and the main features of the parallel machine (TN-310 system) are presented. Section 4 details the parallelization techniques developed and the achieved results in terms of speed-up and track reconstruction efficiency. Finally, some conclusions are derived in Section 5.

[1] The third coordinate (z) is considered to be zero for the tracks we have.
[2] This filtering process is a kind of preprocessing and is not a task of the reconstruction algorithm.

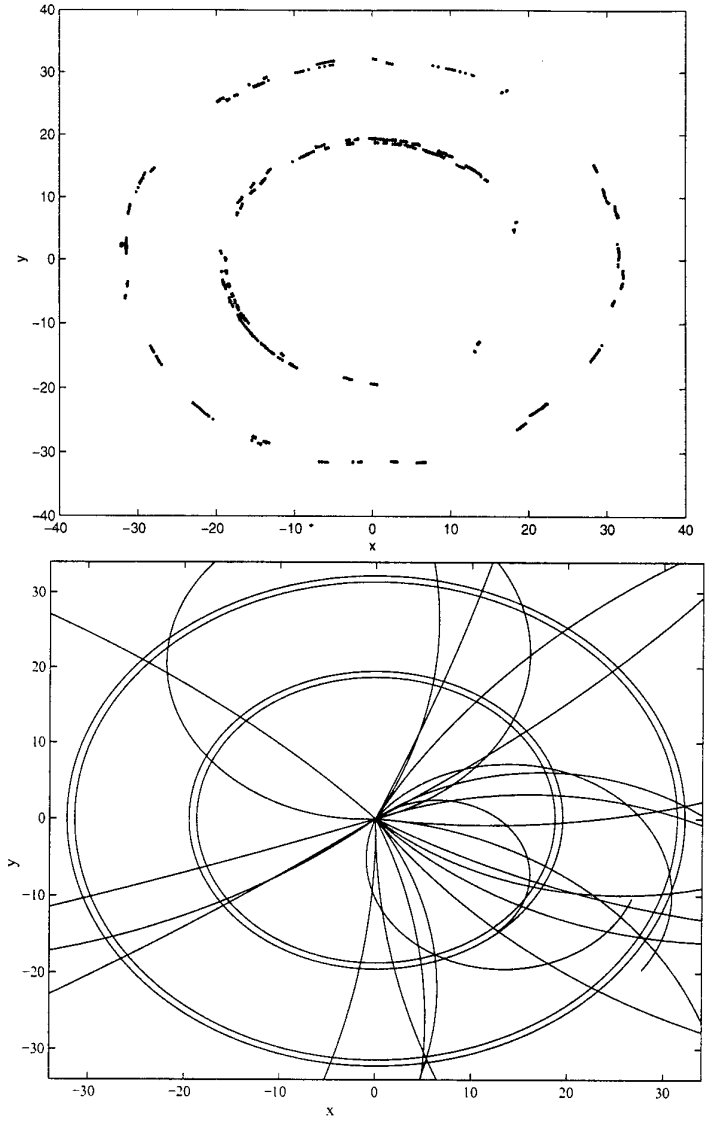


Fig. 3. A typical event: pixels (top) and the corresponding target tracks (bottom).

2 The Hough Transform (HT)

The Hough Transform was introduced by Paul V. C. Hough in 1962 ([3]). The initial idea was to use it in the detection of complex patterns in binary images. The method has encountered a significant resistance to its adoption due to its enormous computational cost. As substantial improvements with respect to its implementation have been made in the last years, the Hough transform finds applications nowadays in many image reconstruction problems.

2.1 The Standard Hough Transform (SHT)

The HT's main goal is to determine the values of the parameters that completely define the original image shape. In order to achieve this goal, the input space is mapped onto the parameter space and by histogramming the resulting mapped points the parameter values are determined. In other words, a global detection problem (in the input space) is converted into a local one (in the parameter space).

As an example, let's consider the problem of detecting straight lines in a noisy environment. Using the slope m and the offset c as the parameters for the straight lines, the model is obtained from these two parameters (where the hat indicates the estimate of a parameter):

$$y = \hat{m}x + \hat{c} \tag{1}$$

From this equation, a relation f can be derived as:

$$f((\hat{m}, \hat{c}), (x, y)) = y - \hat{m}x - \hat{c} = 0 \tag{2}$$

This relation maps each possible combination of parameter values $(\hat{m}, \hat{c})$ onto a set (x, y) of the input space, that is, the parameter space is mapped onto original input data space. From this, an inverse relation, say g, can be defined, so that it maps the input space onto the parameter space. This the so called *backprojection*:

$$g((\hat{x}, \hat{y}), (m, c)) = \hat{y} - \hat{x}m - c = 0 \tag{3}$$

For the straight line problem, relation g results in:

$$c = -\hat{x}m + \hat{y} \tag{4}$$

After having backprojected the input space onto the parameter space, we search for the regions in the parameter space for which the "density of lines" is high, that is, we search for values of (m, c) that have large probability in representing an actual straight line from the input space. This search is performed by building an accumulator over the parameter space, which performs data histogramming. As an example, consider an input space as shown in Fig. 4, where four lines have to be detected from their pixels, in spite of the noisy

pixels. Performing the corresponding backprojection, the histogram in the parameter space is obtained as shown, where the values for m and c corresponding to each straight line can be estimated from the four clear peaks observed in the parameter space.

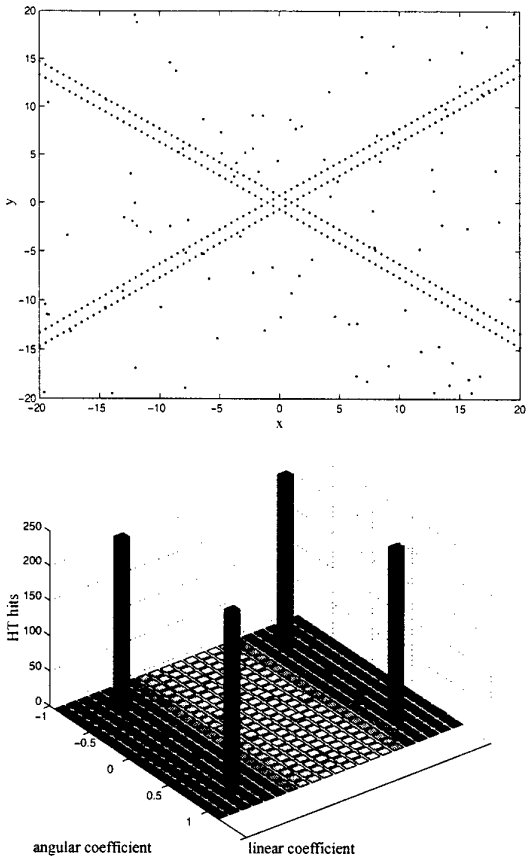


Fig. 4. Four straight lines to be detected in a noisy image (top) and the corresponding histogram in the parameter space (bottom).

The accuracy in the estimation of the curve parameters depends on the granularity of the accumulator. As higher granularity (higher number of channels in the histogram) in the accumulator produces better accuracy, the estimation of the actual parameters improves with finer granularity. Thus, the computational cost for the SHT is usually very high.

The SHT can be easily extended for arbitrary shapes. Relations f and g are simply generalized to:

$$f((\hat{a}_1, \hat{a}_2, ..., \hat{a}_n), (x, y)) = 0 \quad (5)$$

$$g((\hat{x}, \hat{y}), (a_1, a_2, ..., a_n)) = 0 \quad (6)$$

For the track reconstruction problem, we search for peaks in the $\kappa\theta$ parameter space in order to detect helices in the input space.

2.2 The Local Hough Transform (LHT)

The Local Hough Transform [6] aims at reducing the number of operations required to accumulate the HT. Instead of backprojecting each point from the input space onto an entire curve in the parameter space, the LHT maps each *pair* $\{(x_1, y_1), (x_2, y_2)\}$ of pixels from the input space onto its unique correspondent point (a_1, a_2) in the parameter space. For the straight line case, this requires to solve the pair of equations below:

$$y_1 = mx_1 + c \quad (7)$$

$$y_2 = mx_2 + c \quad (8)$$

Of course, if there are n parameters to be determined, a set of n equations shall be solved. The LHT reduces drastically the number of calculations necessary for histogramming the parameter space, as it does not require a scanning process. Although its restrictions for some problems, the LHT is well suited for track reconstruction problem because data are concentrated in two small (local) regions of the input space. Indeed, we do not perform every combination of pixels, but pairs are only formed from pixels belonging to different shells and whose ϕ (polar) coordinates do not differ too much, as very curly tracks are not expected from the physics of the experiment.

3 The TN-310 System

The TN-310 system ([7]) is a MIMD parallel computer with distributed memory. It houses 16 nodes that comply with the HTRAM (*High performance TRAnsputer Modules*) standard, equally split into two cards. Each node can communicate to any other node by sending messages through a fast interconnection network based on STC104 chips ([8]).

Each HTRAM node contains a transputer (InMOS T9000), 8 MB of RAM memory, a DSP (ADSP-21020) and a buffer of 256 KB for communication between the transputer and the DSP. The transputers are very good processors for communication tasks, due to its VCPs (Virtual Channel Processors), and the DSPs are optimized for signal processing applications. Fig. 5 shows the general architecture of the machine, including the interconnection network.

Note that there are different numbers of switches involved in the communication of different HTRAM nodes. Therefore, to achieve faster speeds, an optimum

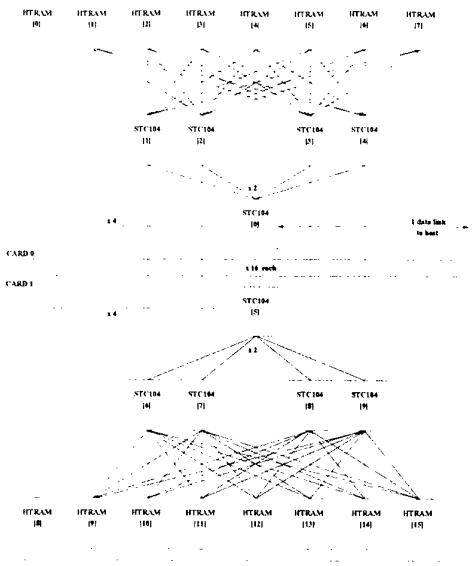


Fig. 5. Interconnection network for TN-310 system.

placement of processes into nodes must be realized. A PC running Windows hosts the system.

Programming for the TN-310 system involves both describing the system configuration and coding the tasks to be run on each processing node. In terms of system configuration, the number of processors to be used must be informed to the system, and the way they will communicate must also be clearly established and configured. These features are described in an extra configuration file.

The TN-310 system provides three layers of programming: PVM (Parallel Virtual Machine), RuBIS (microkernel), and C-Toolset, which was chosen for this work due to its faster execution time. This environment allows processes to be coded in ANSI C language. Some libraries were developped to include communication functions and procedures.

Fig. 6 illustrates the general process of generating a single executable file in C-Toolset from configuration and process codings. A makefile has to be written in order to hold compilation and linking correctly. The internal structure of the machine is kept in a file written in a specific language (NDL-Network Description Language), and can not be changed by users.

4 Track Reconstruction

The tracking reconstruction algorithm based on the Hough transform was implemented on the TN-310 system. Two different parallel approaches were developed and compared with a sequential implementation in the same environment.

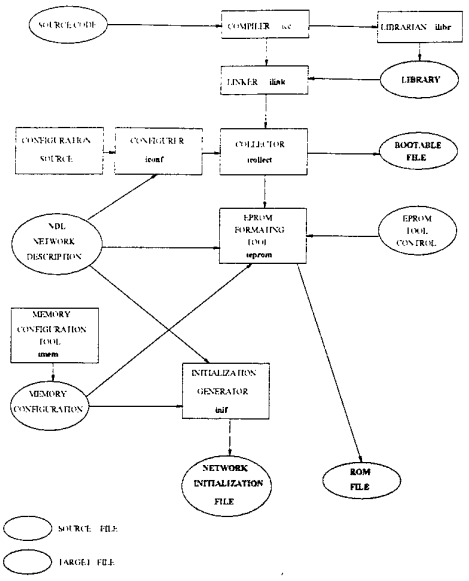


Fig. 6. Compilation and linking in C-Toolset environment.

4.1 The sequential algorithm

For the sequential implementation of the Hough transform, three main parts can be distinguished: accumulation of the HT, elimination of peaks in the histograms that may not represent real tracks and grouping of neighbor peaks that may represent the same track[3]. After these phases, we can verify the efficiency of the algorithm. Steps 2 and 3 can be viewed as a filtering process in the parameter space, and these steps are quite important to avoid detecting ghost tracks.

Some parameters used in steps 2 and 3 must be optimized in order to achieve the best efficiency. This was developed during a training phase. Three parameters are the most important ones: two for determining neighboring regions and the third for determining if a cell has a value high enough to represent a real track. The algorithm was trained for reaching the optimum values for the 79 events available (total of 1321 helices) by considering the set of parameters that resulted in a higher efficiency in reconstructing the tracks. This efficiency is computed by averaging three figures of merit often used in track reconstruction: precision (the rate between the number of real tracks reconstructed and total number of reconstructed tracks), recall (rate between the number of real tracks reconstructed and the actual number of real tracks in the input space) an goodness (rate between the difference of the total number of real tracks reconstructed and the number of ghost tracks detected and the actual number of real tracks

[3] When the accumulator reaches higher granularity, an actual peak may artificially be split into two, so that grouping neighboring cells may be considered.

present in the input space). The optimized algorithm achieved a correct recognition of 86.5% (average efficiency) of the tracks, and 99.5% of these tracks were correctly reconstructed. This corresponds to a resolution better than 10% in the momentum reconstruction of particles, according to an algorithm suggested in [5]). Fig. 7 shows the reconstruction for the event shown in Fig. 3.

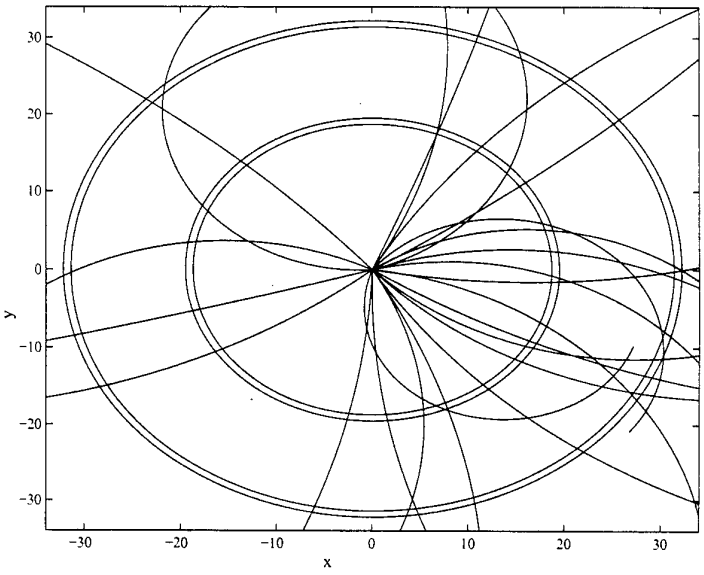


Fig. 7. Reconstructed event (compare with Fig. 3).

4.2 Using Data Parallelism

Having developped the sequential algorithm, the next task was to parallelize it. The first approach was to use a *master/slave* architecture to implement data parallelism (see Fig. 8). A master process continuously receives data from the host machine and distributes them sequentially to free slaves that perform the reconstruction algorithm. The minimum number of slaves required by the application depends on the ratio between computing time and communication time, as when the slave that has first received data to process becomes free, it is useless to add processing nodes in the chain.

For this parallelization, we obtained a speed-up (gain) of 14.7 for the SHT and 11.25 for the LHT. As the optimum numbers of slaves were respectively 15 and 12, the parallel efficiencies were equal to 98% and 93.8%. Note that as the LHT algorithm is faster, its computation/communication ratio is lower, less slaves are necessary to optimize parallelization, and so we get a lower speed-up.

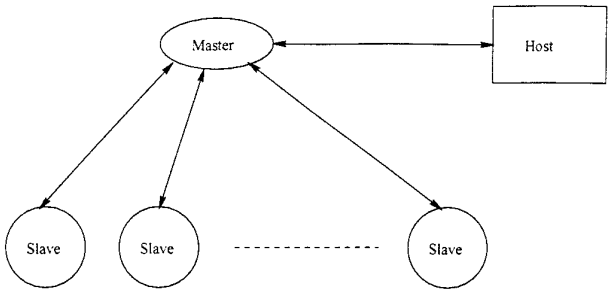


Fig. 8. Master/Slave architecture.

However, the absolute time of operation for the parallel LHT remains lower than that for the parallel SHT[4].

4.3 Using Instruction Parallelism

The second approach for parallelizing the algorithm was to implement a form of instruction parallelism. In this case, slaves operate over the same data (distributed by the master) but performing different parts of the whole reconstruction algorithm. Due to interdependencies, this approach tends to be more communication intensive than the previous one based on data parallelism.

The division of tasks among slaves was made as following: each slave was responsible to execute the whole Hough transform (either global or local) algorithm for the same data over *only one region* of the parameter space, according to Fig. 9. At the step 2 of the reconstruction algorithm (elimination), each cell of the accumulator must know the value stored in all the cells laying in a neighborhood region (for the fronteer cells, these regions are represented by the dashed lines in the figure). Therefore, before starting step 2, slaves must communicate in order to proceed in their tasks.

The way slaves communicate is illustrated in Fig. 10. In order to reduce the communication overheads, only neighbor nodes do communicate. After performing the elimination task and before the grouping phase, slaves must communicate again, sending the corrections their neighbors need. These corrections are a direct consequence of the scanning mechanism used to look for peaks in the histogram (from top to bottom, a line scanning mechanism).

The main difference of this communication scheme from the previous one is that now the slaves send data only to the neighbors that are at their right or below them, due to the scanning direction (see Fig. 11).

Fig. 12 illustrates the whole procedure for implementing this instruction parallelism.

[4] A time of 2.02 seconds is necessary to process an event in the parallel LHT. This is an average value, because events have different numbers of tracks.

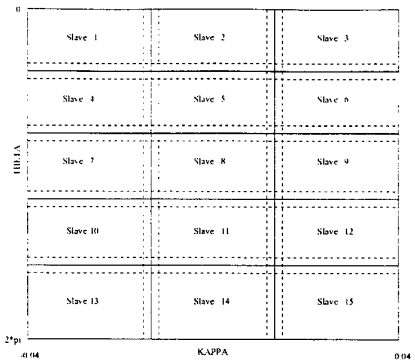


Fig. 9. The parameter space division and the neighborhood regions.

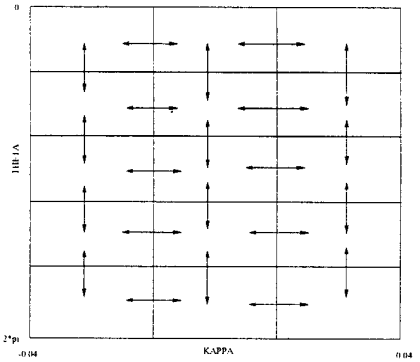


Fig. 10. Communication map among slaves.

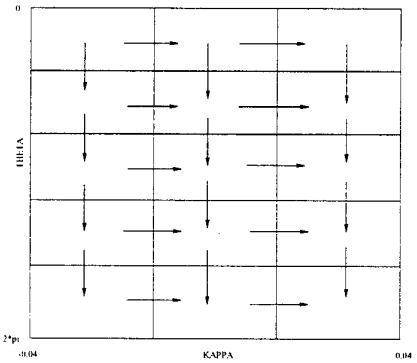


Fig. 11. Sending corrections after elimination.

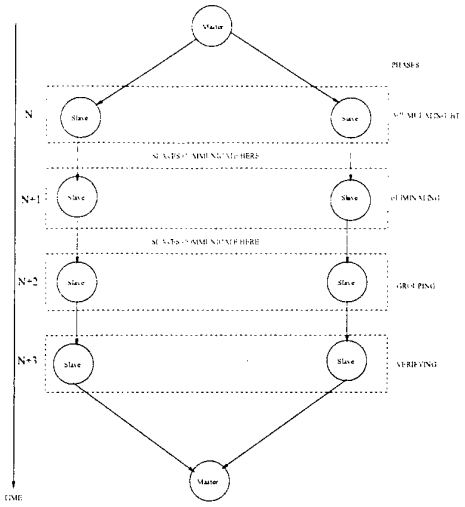


Fig. 12. Data/Instruction flow in time.

For this application, a speed-up equal to 11.8 and an efficiency of 78.8% (for 15 slaves) were achieved for the SHT. This algorithm was not suitable for implementing the LHT, because the speed-up remained very low in comparison to the one with data parallelism due to the fact that, in the LHT, few communication is desired (computation is faster), and in this second application a lot of communication is established.

5 Conclusions

A track reconstruction algorithm for a scintillating fibre tracker in experimental high-energy physics was developed using Hough transforms. The algorithm was successfuly implemented in a 16-node parallel machine. Two methods for partition of the sequential implementation were developed, using data and instruction parallelism techniques.

The reconstruction algorithm was able to identify correctly 86.5% of the tracks and allowed the computation of the momentum with a resolution better than 10% for 99.5% of the identified tracks.

The parallel approach proved to run this complex reconstruction algorithm in about 2 seconds per event, and using data parallelism a 98% of parallelism efficiency was achieved. The algorithm is now being transported to a similar MIMD environment based on DSPs, in order to achieve a further improvement in processing speed.

6 Acknowledgements

We would like to thank CAPES, CNPq, FUJB and FAPERJ (Brazil), CERN (Switzerland), and UE (Belgium) for the financial support provided to this project.

References

1. http://www.cern.ch.
2. T. Sjöstrand, Pythia 5.6 and Jetset 7.3-Physics and Manual CERN-TH/6488-92.
3. Illingworth, J. and Kittler, J.. "A Survey of the Hough Transform". Computer Vision, Graphics, and Image Processing 44, pp. 87-116, 1988.
4. R. Brun et al., GEANT 3. CERN-DD/EE/84-1.
5. F. Anselmo, F. Block, L. Cifarelli, C. D'Ambrosio, T. Gys, G. La Commare. H. Leutz, M. Marino and S. Qian. "Track recognition with a central two-shell scintillating fibre tracker (SFT)". CERN-ECP/94-7, 1994.
6. Ohlsson, Mattias and Peterson, Carsten. "Track finding with deformable templates - the elastic arms approach". Computer Physics Communications 71, pp. 77-98, 1992.
7. Telmat Multinode. "TN310 System Manual and Training Set". France, 1995.
8. "Networks, Routers and Transputers". Edited by M. Jane et al. SGS-Thomson, 1993.

Modeling of Explosions using a Parallel CFD Code

C. Troyer[1] *; H. Wilkening[2] **; R. Koppler[1] and T. Huld[2]

[1] GUP, Johannes Kepler University Linz, A–4040 Linz,Austria
[2] ISIS, Joint Research Centre, I–21020 Ispra (Va), Italy

Abstract. Computational fluid dynamics is known as one of the most challenging applications for high performance computing due to the complex physics often involved in such simulations. In this paper we address the problems that arise in the simulation of chemical explosions. In such a simulation physical scales in time and space must be resolved, which may vary by many orders of magnitude and therefore put strong requirements on the computational resources.
Parallel computation can offer the required computational needs at a reasonable price when distributed memory machines such as workstation clusters can be used.
In this paper we present first experiences made when parallelizing the CFD code REACFLOW, an unstructured-grid CFD code for solving transient chemically reactive flows such as vapour cloud explosions.

Keywords: Computational fluid dynamics, Distributed computing and operating systems

1 Introduction

Although gas cloud explosions in industrial environments have a very low probability, their effect on society can be severe. In 1974 for example, 28 people were killed by an explosion in Flixborough, England [9].

Fig. 1 shows an aerial view of a chemical factory plant in Ludwigsburg, Germany, after an explosion in 1948. The cause was a hydraulic rupture after exposure to solar radiation of an overfilled tank car, followed by a vapour cloud explosion of the released 30400 kg dimethyl ether. In the chemical vapour explosion 207 people were killed and 3818 injured, 500 seriously [6].

Due to the large scales and the complex geometry of industrial plants and to strong dependencies of explosions on scale and geometrical shape, explosions on large scale are impossible or prohibitively expensive to study by experiments only. A potential remedy is to study explosions by computational fluid dynamics simulation. However, explosion simulations make enormous demands on the computer code because an explosion is an unsteady phenomenon. Different flow

* Candidate to the best student paper award

** Corresponding author: Heinz.Wilkening@jrc.it, Tel.:+39 0332/78-5181, Fax:+39 0332/78-6198

Fig. 1. Aerial view after an explosion in a chemical plant in Ludwigsburg, Germnay in 1948

regimes such as subsonic, transonic and supersonic flow must be adequately handled by the numerical scheme. In addition, a large complex geometry must often be modeled, but due to the nonlinearities of e.g. the chemical source terms, high resolution is desired at certain positions in time and space. Parallel computing is an obvious way to improve the turn-around time for such simulations. In addition, the simulations are often limited by the amount of memory which can be accessed by a ordinary workstation. Such a situation is quite common with today's hardware, and again here parallel computing can help by accessing the memory of several workstations for a single calculation.

In this paper we describe the experiences made in parallelizing REACFLOW, a CFD-Code for the simulation of chemically reactive gas flows, in particular for gas cloud explosions. Due to the different scales in space and especially in time, special attention is made to study the effects arising from the different time scales involved.

The paper is organized as follows: section 2 presents the fundamental equations to be solved numerically. In section 3 we describe the numerical discretization and methodology. A sample calculation run is presented in section 4. Section 5 gives a short description of the parallelization procedure, and results using this procedure are presented in section 6. Finally, section 7 contains a summary of the results obtained.

2 Governing equations

The numerical simulation of combustion processes is based on conservation equations for mass, chemical species, momentum and energy. For turbulence modeling

additional transport equations for the turbulent kinetic energy k and the turbulent kinetic energy dissipation ϵ are solved. In weak form the system of governing equations can be written in the following compact notation [3]

$$\frac{\partial}{\partial t}\left(\int_{\Omega} U \mathrm{d}V\right) + \int_{\partial\Omega} n_i F_{i,\mathrm{conv}}(U)\mathrm{d}A + \int_{\partial\Omega} n_i F_{i,\mathrm{diff}}(U,\nabla U)\mathrm{d}A = \int_{\Omega} S(U)\mathrm{d}V \tag{1}$$

where $U = (\rho_\gamma, \rho u_i, \rho E)^{\mathrm{T}}$ is the vector of conserved quantities which are the unknowns of the system. Here, ρ_γ, $(\gamma = 1, \Gamma)$ are the partial densities, ρu_i is the momentum vector and ρE is the total energy. The other terms are given as follows:

Convective fluxes: $F_{i,\mathrm{conv}} = \left(\rho_\gamma u_i, \rho u_i u_j + p\delta_{ij}, \rho u_i \left(h + u_i^2/2\right)\right)^{\mathrm{T}}$

Diffusive fluxes: $F_{i,\mathrm{diff}} = \left(-D_\gamma \frac{\partial Y_\gamma}{\partial x_i}, -\tau_{ij}, -\sum_{\gamma=1}^{\Gamma} h_\gamma D_\gamma \frac{\partial \rho_\gamma}{\partial x_i} - \tau_{ij}u_j - \lambda\frac{\partial T}{\partial x_i}\right)^{\mathrm{T}}$

Source terms: $S = \left(-\dot{\rho}_\gamma, -\rho g_i, -\rho g_j u_j - \sum_{\gamma=1}^{\Gamma} \dot{\rho}_\gamma \Delta h_\gamma^f\right)^{\mathrm{T}}$, including chemical

reactions.

For detonation modeling the chemical mass production may be described adequately by a reduced chemical kinetics scheme

$$\dot{\rho}_\gamma = M_\gamma \sum_{r=1}^{R} \left(\nu_{\gamma r,b} - \nu_{\gamma r,f}\right)\dot{\omega}_r \tag{2}$$

where M_γ is the molar mass and $\nu_{\gamma r,f}$ is the stoichiometry of the γ'th component in the r'th forward reaction ($\nu_{\gamma r,b}$ backward reaction) of a total of R chemical reactions. $\dot{\omega}_r$ is the progress rate for the r'th reaction given by:

$$\dot{\omega}_r = k_{f,r}(T)\prod_{\gamma=1}^{\Gamma}\left(\frac{\rho_\gamma}{M_\gamma}\right)^{\nu_{\gamma r,f}} - k_{b,r}(T)\prod_{\gamma=1}^{\Gamma}\left(\frac{\rho_\gamma}{M_\gamma}\right)^{\nu_{\gamma r,b}} \tag{3}$$

where the forward and backward Arrhenius' rates for the r'th reaction, $k_{f,r}(T)$, $k_{b,r}(T)$, have the following form:

$$k_{f,r}(T) = A_{f,r}T^{b_{f,r}}e^{-E_{f,r}/RT} \tag{4}$$

(equivalent expression for $k_{b,r}(T)$), where $A_{f,r}$, $b_{f,r}$ and $E_{f,r}$ are constants for a given reaction, r.

3 Numerical methods

3.1 Geometrical discretization

The compressible solver in REACFLOW is based on a numerical scheme of the Finite Volume type. REACFLOW uses a computational grid which is unstructured in space and based on subdividing the computational domain into simple geometrical elements. In two dimensions these elements are triangles, and in

three dimensions they are tetrahedra. For the three-dimensional version, the geometrical treatment is very similar to the one proposed by Nkonga and Guillard [8].

Based on these elements, a set of *control volumes* (CV's) are constructed around each vertex of the elements. In 2D the control volumes are delimited by segments of the medians of the elements. Inside a triangle the medians meet at the center of mass. $x_g = 1/3(x_i + x_j + x_k)$. In three dimensions the medians of the tetrahedra are planes going through the medians of one of the faces of the tetrahedron and the opposite vertex. These medians again meet at the center of mass of the tetrahedron, whereby the quadrilateral segments delimiting the control volumes will remain plane. The segments of the medians delimiting the control volume at one of the vertices of a tetrahedral element are shown in Fig. 2.

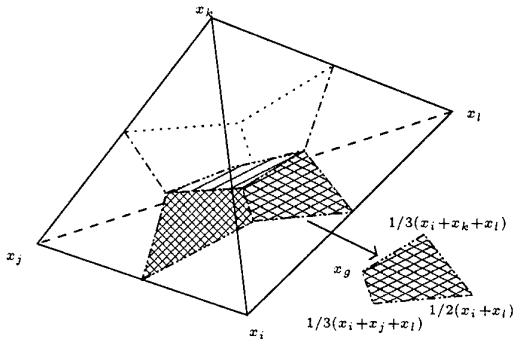


Fig. 2. One tetrahedral element and the quadrilateral segments of the medians delimiting the control volume around the node x_i inside this element.

The boundary between any two control volumes is then formed by the segments inside those elements that have the two control volumes in common. Inside a given control volume, $\mathcal{V}_j$, only the *average* of the state, U, is known. This is denoted $\overline{U_j}$. Inside the control volume a certain functional dependence may be assumed for U so long as the integral over $\mathcal{V}_j$ remains the same. Globally, U then has a piecewise continuous form.

3.2 Time discretization

For the time discretization, normally a simple first-order Euler step is used, whereby

$$\frac{\partial}{\partial t}\int_{\Omega} U dV \simeq \frac{|\Omega|}{\Delta t^n}\left(U^{n+1} - U^n\right) \tag{5}$$

where the superscript n denotes the n'th time step, and U^n is an approximation to $U(t^n)$. This calculation may be explicit or implicit depending on whether the flux and source terms are evaluated at t^n or t^{n+1}. For fast transient flow phenomena such as shocks and explosions the explicit timestepping is preferred and this has been used in the results described in this paper.

For the flux and source terms an *operator splitting* technique has been employed whereby each term in Eq. 1 is evaluated in turn and the resulting state updated before the next term is calculated.

3.3 Convective terms

For each control volume the rate of change of U is determined by the flux into that volume (Eq. 1). The type of discretization used means that U is generally discontinuous across boundaries. If we assume that the state is constant on either side of a boundary segment between two control volumes, we obtain a Riemann problem. The problem of calculating the flux can then be solved by a number of well-known methods. In REACFLOW we have implemented variants of Roe's approximate Riemann solver [10], and a solver of the flux-vector-splitting type.

3.4 Diffusive Terms

The diffusive flux terms all depend on the gradient of the state vector, which is not defined at the boundaries of the control volumes. A different approach must therefore be taken. For the gradients we adopt a continuous reconstruction where the state is assumed to vary linearly over each element with the values at the vertices given by the average state in the corresponding CVs.

3.5 Source term treatment

In the case where the source terms depend only on state variables, the source terms in each CV can be calculated separately, using the average state vector in that CV. This is the case for the chemical and gravitational source terms as well as for some of the turbulent source terms.

If the source terms contain the gradient of the state, as is the case for some of the turbulent terms, the gradient is calculated elementwise as for the diffusive terms. Then in each CV the source term is calculated separately for each element making up that CV.

Often the source terms will make the problem stiff, so an implicit timestepping method is needed. For instance, the finite-rate chemistry source terms of Eq. 2 may be very large for combustion processes. The finite-rate chemistry problem then becomes for the j'th CV:

$$\frac{|\mathcal{V}_j|}{\Delta t}\left(\begin{pmatrix}\rho_\gamma\\ \rho E\end{pmatrix}^{n+1}-\begin{pmatrix}\rho_\gamma\\ \rho E\end{pmatrix}^{n}\right)=\begin{pmatrix}\omega_\gamma(\rho_\gamma^{n+1},T^{n+1})\\ \sum_{\gamma=1}^{\Gamma}\omega_\gamma(\rho_\gamma^{n+1},T^{n+1})\Delta h_\gamma^f\end{pmatrix} \qquad (6)$$

This is a nonlinear algebraic problem, which is then solved using a modified version of the Newton-Raphson method[4]. Using such an iterative method has a great inpact on the parallelization, as will be shown later.

4 An example large scale detonation simulation

In this section we will show an example of a simulation made with REACFLOW, and show comparisons with experiments. For this simulation the non-parallel version of REACFLOW was used. With this version it is possible to perform adaptive simulations. For details of the adaptive algorithm see [11].

This large scale hydrogen experiment was performed in the Russian RUT Facility located near Moscow. The facility has a total volume of about $263m^3$ in the configuration used here. The volume has compartments of different size. There is a large volume, the so called canyon, followed by a long channel. The facility has a total length of 28m. An outline is shown in Fig. 3. Details of the geometry and the experiments performed can be found in [1].

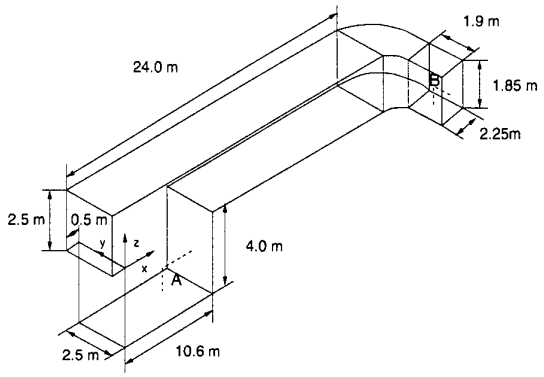


Fig. 3. Outline of the RUT facility.

For the simulation we chose test **hyd5** with a uniform hydrogen concentration of 20% hydrogen (by volume) in air at ambient conditions. The detonation was initiated by 200g of high explosive at a low position inside the canyon near the endwall (position A).

Fig. 4 shows a detail of the pressure distribution in the canyon at 2.5ms after ignition. The initial spherical detonation has interacted with the channel walls, which generates a complex sytems of pressure waves. Highest pressures occur right at the detonation front.

For the simulation with REACFLOW we used an initial grid with 3615 nodes and 15592 elements. The mesh distribution is nearly uniform. This corresponds to an initial resolution of $\Delta x \simeq 0.5$m. During the simulation nodes were added down to a minimum resolution of 0.04m depending on the pressure gradient. The maximum number of nodes reached during the simulation was about half a million node points.

Fig. 5 shows comparisons between experimental and simulated pressure time history plots at 2 different positions in the canyon. The detonation velocity

Fig. 4. Hydrogen detonation in the RUT facility, with 20 Vol% H_2. Pressure at $t = 2.5$ms after ignition, 25 isolines from 1 to 15bar.

is 1750m/s in the experiment as well as in the simulation. Results compare generally well with the experiments.

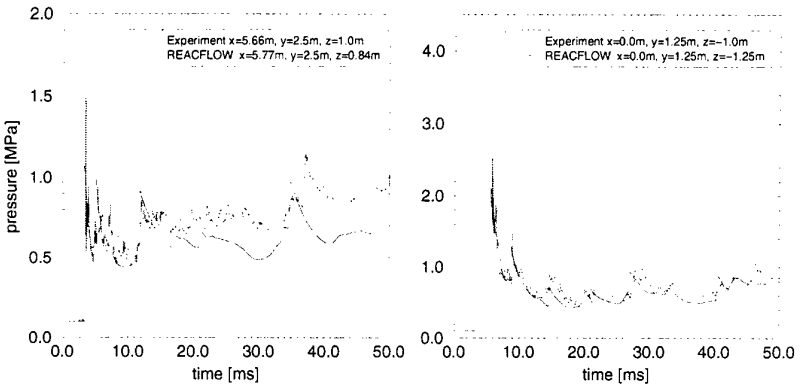


Fig. 5. Hydrogen detonation in the RUT facility. Pressure versus time for two positions inside the canyon (left is a sidewall, right is an endwall position).

5 Parallelization Procedure

Due to space limitations we describe here only the basic principles rather than the details. To ensure portability, parallelization is based on the message passing paradigm, and this is implemented using MPI libraries[7] which are available for almost every parallel machine.

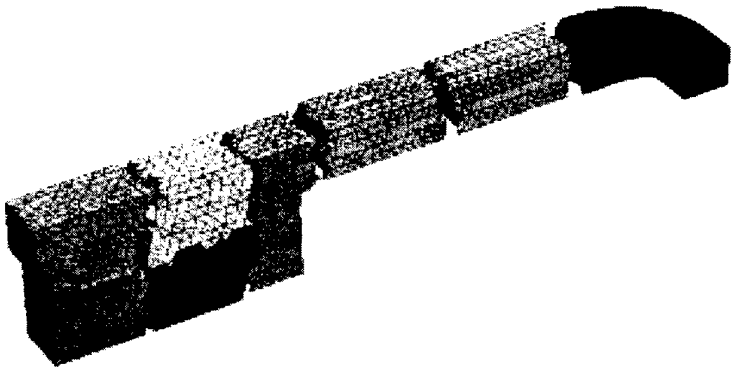


Fig. 6. Partition of the RUT geometry into eight subdomains.

To define the tasks to be calculated the domain is split into as many subdomains as there are processors available. Since at the moment REACFLOW is parallelized in a non–adaptive static grid version the partitioning of initial grid is done outside REACFLOW using a program called PART. PART is based on the public domain libary METIS [5]. Fig. 6 shows how PART–METIS splits the calculation domain of the RUT facility shown in section 4 into eight subdomains, at the same time including also partitioning of the grid. In a parallel calculation each subdomain is calculated by one processor.

Overlapping subdomains are used, since the calculation of the fluxes for a given CV require the states in the neighbouring CV's to be known. Using this technique, communication must happen only before, not during the calculation step. This reduces the amount of communication significantly for the price of additional memory use, as the overlapping CV's are stored twice. In addition, larger messages can be exchanged between processors, which again reduces communication time due to saved latency time of a communication process.

To organize the communication every processor P keeps the following lists:

- **neigbor**: a list of direct neighbour–processors P'
- **recvMap**: a list of pointers to CV's of P that must be updated by values of CV's from P', NULL if P' is not a direct neighbour of P
- **sendMap**: a list of pointers to CV's of P that must be sent to P' to update these CV's on P', NULL if P' is not a direct neighbour of P
- **recvCount**: integer list with the number of CV's that must be updated from P', recvCount[P'] = 0 if P' is not a direct neighbour of P
- **sendCount**: integer list with the number of CV's that must be sent to P', sendCount[P'] = 0 if P' is not a direct neighbour of P

Fig 7 illustrates these lists for processor P1, by means of a small example in two dimensions.s

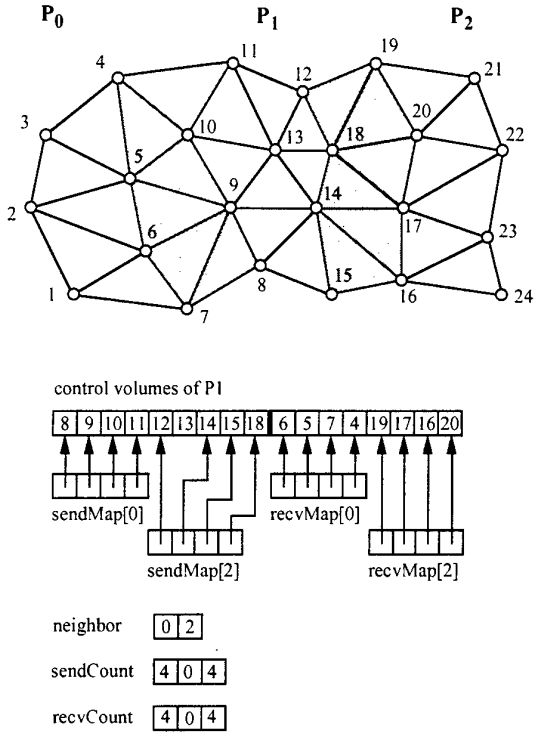


Fig. 7. Data structure for the communication table for processor P1.

6 Examples of simulations using Parallel REACFLOW

Two example cases are presented in this section: a non-reactive shock-tube simulation, and a reactive detonation simulation, shown in non-parallel form in section 4 (RUT). All simulations are performed on a shared-memory SGI-Origin 2000 parallel computer using from 1 up to 16 processors and a distributed memory workstation cluster. The workstation cluster is built from eight DEC-Alpha 21164A based machines connected by a Gigabit Myrinet network in a ParaStation 2 system configuration[1].

For all calculations which have been made it was found that there was no essential difference in the physical results between the single processor version of the REACFLOW and the parallel version. There was also no influence by the number of processors used. The small differences that have been found are due

[1] Details of the ParaStation 2 concept can be found under http://parastation.ira.uka.de/.

to roundoff errors, as in the parallel version numerical operations are performed in a different order.

The memory use of the parallel version of REACFLOW scales well against the single processor version of REACFLOW, nearly perfectly up to 10 processors but also depending on the problem size.

6.1 Inert shock–tube test case

As an example to test the fluid solver, a chemically inert test case was chosen. This allows to test the communication of the parallel version of REACFLOW including convection, diffusion and turbulence. Results were also compared with experiments.

The 12m FZK–shock–tube (0.35m diameter) [2] was separated into a 3m long helium filled high pressure section and a 9m long low pressure section. The last 6m of the low pressure section contained circular orifices as turbulence generators which blocked 30% of the tube.

Table 1. Speedup measurements for shock-tube simulation on SGI–ORIGIN 2000

Processors	1	2	3	4	5	6	7	8	9	10	11	12	14	16
Time [min]	177	89	56	41	31	27	25	23	20	19	19	18	17	17
Speedup	-	1.99	3.16	4.41	5.71	6.56	7.08	7.70	8.85	9.32	9.32	9.83	10.41	10.41

Table 2. Speedup measurements for shock-tube simulation on ParaStation 2 workstation cluster

Processors	1	2	3	4	5	6	7	8
Time [min]	258	169	165	110	94	50	36	35
Speedup	-	1.53	1.56	2.35	2.74	5.16	7.17	7.37

The results shown in tab. 1 and 2 show measured speedups for 1 to 16 processors for the SGI–Origin and 1 to 8 processors for the ParaStation 2 workstation cluster. Speedup is nearly linear up to 10 processors even though with a grid of 30517 nodes and 115239 elements the subtasks have become very small for each processor when using ten processors. For some values the speedup is even superlinear, probably due to cache effects. This is more visible on the ParaStation 2 workstation cluster as the cache of the DEC–Alpha is much smaller than that of

the SGI–Origin 20000. The results for the ParaStation 2 workstation cluster are also influenced by the fact that during the calculation the cluster may be used by other users both in interactive mode and in addition as FTP server which can generate unforeseen loads. These loads can distort the results.

6.2 Chemical reactive detonation simulation

For testing the chemical reactive part of the parallel REACFLOW version the simulation described in section 4 was repeated with one up to 16 processors in parallel. As this simulation had to be done in a non–adaptive way, a finer static grid with 160794 nodes and 841566 elements was used. As this size was beyond the memory capabilities of a single workstation of the ParaStation 2 cluster, in these tests only the SGI–Origin 2000 was used.

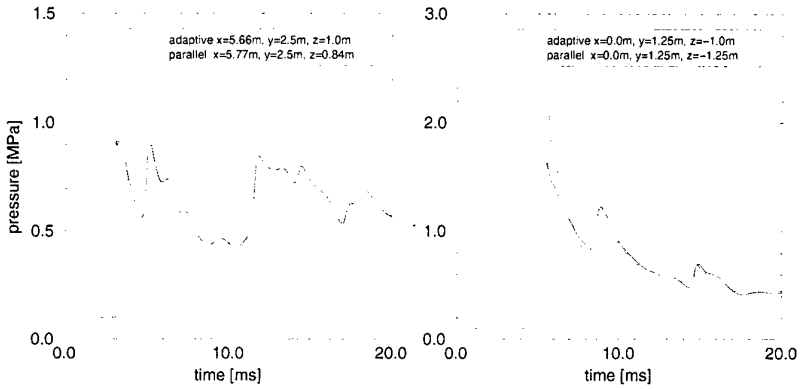


Fig. 8. Hydrogen detonation in the RUT facility. Pressure versus time for two positions inside the canyon (left is a sidewall, right is an endwall position).

Fig. 8 shows a comparison between the adaptive calculation presented in section 4 and the result obtained by a parallel simulation, described in this section. Since in the parallel case the spatial resolution of 0.13m is much coarser than 0.04m for the adaptive one, results are not as good as before. The maximum overpressure in the detonation front is more underpredicted than before.

Tab. 3 shows measured speed-ups for 1 to 16 processors. The speed-up is very poor for more than 4 processors even though the problem (with 160794 nodes and 841566 elements) is much bigger than the inert shock tube calculation. This can be explained by the problems arising due to chemical reactions in a detonation. Chemical reaction takes place on much shorter time scales than the fluid flow; therefore the chemical source term Eq. 2 is solved implicitly by an iterative algortithm (section 3.5). This implicit algorithm shows different convergence behaviour depending on whether or not a detonation wave is passing through the

Table 3. Speedup measurements for detonation simulation on SGI–ORIGIN 2000

Processors	1	2	4	6	8	10	12	16
Time [min]	2070	1212	698	515	443	370	343	279
Speedup	-	1.71	2.97	4.02	4.67	5.59	6.03	7.42

given point in space. The fluid solver itself is explicit which means that the calculation time for a timestep stays almost constant for each subdomain at any time.

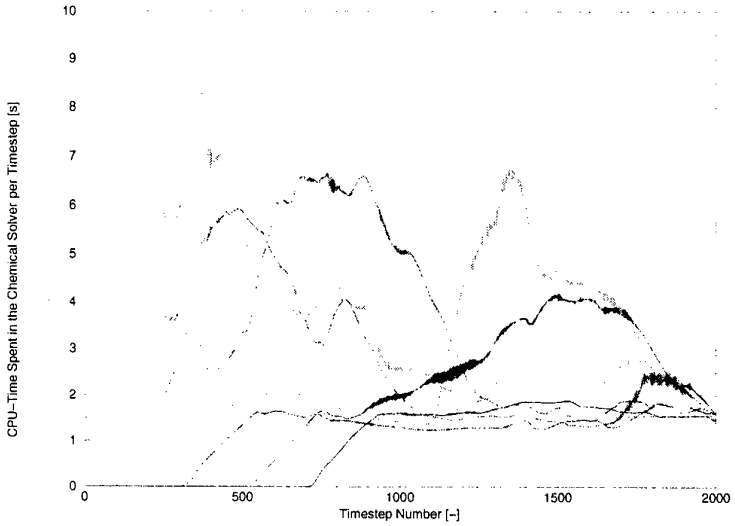


Fig. 9. Loadbalancing for the chemical solver for 10 processors.

Fig. 9 shows the CPU time spent in the chemical solver per timestep for ten processors. The CPU time varies between 0 and 9s with an average of about 2 to 3s. Nevertheless, as all processor are synchronized before and after the chemical solver, the fastest processors have to wait for the slowest ones. The total time spent in the chemical solver for all processors is about 196min, which is more than 50% of the whole calculation time. If optimal load balancing within the chemical timestep could be achieved, the calculation time for the chemical timestep would be reduced to about 85min per processor or 259min total calculation time including also the fluid solver. Consequently the speedup would also increase from 5.59 to 7.99, which then would correspond to a resonable efficiency

of 79.9%.

In addition, fig. 9 shows nicely how the computational effort for the chemical solvers increases when the detonation front moves into the corresponding subdomain of a processor.

Fig. 10 illustrates the load balancing of the chemical solver against the fluid solver. The total calculation time measured is mainly the sum of the maximum values of both graphs which is in total about 528min. The time spent in the chemical solver is nearly equal to that of the fluid solver. If there were optimal load balancing in the chemical solver as well as in the fluid solver, the total calculation time would reduce to 413min, with the chemical solver using about 40% and the fluid solver 60% of the total calculation time.

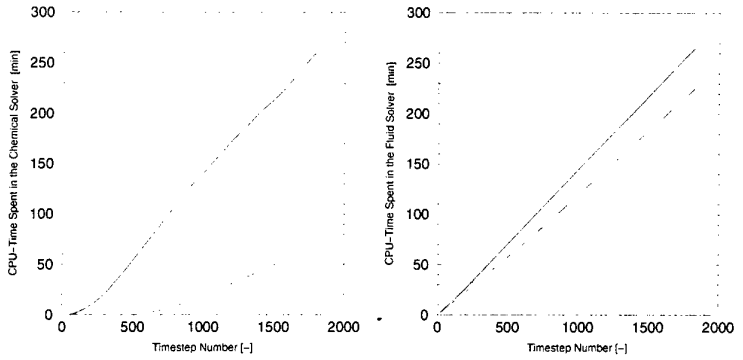


Fig. 10. Load balancing for the chemical solver (left) compared to the fluid solver (right) for 4 processors. Graphs show total time for maximum, minimum and average values of all 4 processors, integrated over all timesteps.

7 Summary

We have presented a parallelized version of REACFLOW, an unstructured grid CFD code for studying combusting gas flows. Preliminary results show that for a test case with rather complicated physics (shock waves and turbulence) the parallelization speed-up results are excellent both on shared-memory and distributed-memory architectures. For a case with combustion the iterative chemical source term solver causes rather severe load imbalance. To solve this problem, it is likely that dynamic node repartitioning will have to be introduced in the code. Such a solution would also be an advantage when dynamically adaptive grids are used.

For calculations where load imbalance is not a serious issue the code can be used both on shared-memory and distributed-memory computers, showing that communication is not a bottleneck. Being able to run on distributed memory

machines (in the shape of workstation clusters) is important, since these enjoy a significant advantage in price over shared-memory machines. This idea is fully supported by the ParaStation concept.

Acknowledgements

This paper is part of a Diploma–Thesis prepared by Christoph Troyer at the University of Linz, Austria, in collobration with the Joint Research Centre of the European Commission in Ispra, Italy.

We would like to thank Thomas Warschko from the University of Karlsruhe for his support by providing CPU time on their ParaStion 2 workstation cluster and also for his useful suggestions made during the project.

We would also like to thank Wolfgang Breitung and his co-workers of the Forschungszentrum Karlsruhe, Germany, for their support in providing us with the experimental data from the FZK shock-tube tests and the data from the RUT facility.

References

1. Breitung, W., Dorofeev, S. B., Efimenko, A. A., Kochurko, A. S., Redlinger, R. and Sidorov, V. P. (1994) Large Scale Experiments on Hydrogen-Air Detonation Loads and their Numerical Simulation, Proc. Int. Topical Meeting on Advanced Reactor Safety (ARS '94), Pittsburgh, USA, Vol. 2, pp. 733-745.
2. Breitung W., Royl P. and Veser A., Results on Hydrogen Behavior and Mitigation in Severe PWR Accidents, FZKA 5914 report, Karlsruhe, Germany, 1997
3. Grasso F. and Meola C., Euler and Navier–Stokes equations for compressible flows: finite–volume methods. In: Handbook of Computational Fluid Mechanics, Peyret, R. (Ed.), Academic Press, London, 1996
4. Huld T., A Finite-Rate Chemistry Solver for REACFLOW, JRC Ispra Report
5. Karypis G. and Kumar V., METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Tech. rep., Department of Computer Science, University of Minnesota, 1995 (http://www.cs.umn.edu/ karypis/metis/metis.html)
6. Marshall V. C., Major Chemical Hazards, Ellis Horwood Limited, Chichester (1987)
7. M.P.I. Form, MPI - A Message Passing Interface Standard, Computer Science Technical Report CS-94-230, University of Tennessee (1994)
8. Nkonga B. and Guillard H., Godunov type method on non-structured meshes for three-dimensional moving boundary problems, Comput. Methods Appl. Mech. Engrg.**113**, 183-204 (1994)
9. Parker R. J., The Flixborough Disaster, Report of the Court of Inquiry, HMSO (1975)
10. Roe, P. L., 1980, Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes, J. Comp. Phys., **43**, pp. 357–372.
11. Wilkening H. and Huld T., An adaptive 3–D CFD solver for modeling explosions on large industrial environmental scales, Combustion Science and Technology, **149**, pp. 361-388, 1999

Fluvial Flow of the Guaíba River - A Parallel Solution for the Shallow Water Equations Model

Rogério Luis Rizzi[1,3], Ricardo Vargas Dorneles[2,3], Cesar Albenes Zeferino[3], Tiaraju A. Diverio[3], Philippe O. A. Navaux[3], Altamiro A. Susin[3], Sergio Bampi[3]

[1] Departamento de Matemática e Estatítica, Universidade Estadual do Oeste do Paraná, Campus de Cascavel, Rua Universitária, 2069, 85801-110, Cascavel, PR, Brasil.

[2] Departamento de Informática, Universidade de Caxias do Sul, Rua Francisco Getúlio Vargas, 1130, 95001-970, Caxias do Sul, RS, Brasil.

[3] Programa de Pós-Graduação em Computação, Instituto de Informática, Universidade Federal do Rio Grande do Sul, C.P. 15.064, 91501-970, Porto Alegre, RS, Brasil.

{rizzi, cadinho, zeferino, diverio, navaux, susin, bampi}
@inf.ufrgs.br

Abstract. This work presents a parallel solution for the simulation of the fluvial flow of the Guaíba River implemented in a PC cluster with message passing. The governing equations are discretized by centered finite difference, which are defined into a staggered grid. Applying the ADI technique, it was developed a semi-implicit numerical scheme, which is solved in a parallel way using domain splitting in the *Y* direction and the pipelined Thomas in the *X* direction.

1 Introduction

The Guaíba River bathes all the metropolitan region of the city of Porto Alegre (at the south of Brazil). With 470 km^2 of surface, it receives the outflow of the Jacuí delta, which is formed by the confluence of Jacuí, Caí, Sinos and Gravataí Rivers, and flows into the Patos Lake [1]. It is about 50 km long and 15 km wide in some sections. The Guaíba River is sited between the 50° and 55° West parallels and 28° and 35° South latitudes. It is quite important for water supplying, fluvial transport and soil irrigation of its region. However, it receives a lot of industrial and domestic pollution.

The development and implementation of a high-resolution computation model allows the detailed simulation of the hydrodynamics behavior and mass transport in the river. This could help in the choice of the outflow points of the sewerage system and in the planning and evaluation of the impact of works for hydric usage, and so on.

However, to perform the simulation with the required refinement in an useful time, it is necessary high computational power and a lot of memory. PC clusters (PCCs) are an effective alternative to get high-performance in research institutes that can not pay for expensive supercomputers. With the development of high speed switched LANs, such as Myrinet, it is possible to build a PCC with an equivalent performance of a vector computer, but with a fraction of its price.

The Group of Computational Mathematics and the Group of Parallel and Distributed Processing of the Institute of Computer Science of the UFRGS has been working with the study and parallel implementation of numerical methods for partial differential equations (PDEs) solution on distributed memory machines. These studies have been performed on a PC cluster, where a parallel computational model for the Guaíba River was implemented.

2 Mathematical Model and Boundary Conditions

The mathematical model used in these parallel solutions is based on the Shallow Water Equations (SWEs), which are the governing equations for the two-dimensional flowing. They are obtained by means of the vertical integration for the horizontal water velocity to obtain the averaged horizontal velocity, resulting in a 2-D model. The SWEs are a quasi-linear system of hyperbolic PDEs for an incompressible and inviscid fluid with a free surface. Such equations formed by horizontal momentum and continuity equations can be described as [1]:

$$\frac{\partial U}{\partial t}+U\frac{\partial U}{\partial x}+V\frac{\partial U}{\partial y}-\Omega V+g\frac{\partial \eta}{\partial x}+g\frac{U\sqrt{U^2+V^2}}{C_h{}^2 Hu}-\frac{\tau_{sx}}{\rho_{water}Hu}-\varepsilon\nabla^2 U=0\ . \qquad (1)$$

$$\frac{\partial U}{\partial t}+U\frac{\partial V}{\partial x}+V\frac{\partial V}{\partial y}+\Omega U+g\frac{\partial \eta}{\partial y}+g\frac{V\sqrt{U^2+V^2}}{C_h{}^2 Hv}-\frac{\tau_{sy}}{\rho_{water}Hv}-\varepsilon\nabla^2 V=0\ . \qquad (2)$$

$$\frac{\partial \eta}{\partial t}+\frac{\partial(HuU)}{\partial x}+\frac{\partial(HvV)}{\partial y}=0\ . \qquad (3)$$

where:

- $\eta=\eta(x,y,t)$ is the elevation of the water above the mean level;
- H_u and H_v represent the depth below the mean level;
- $Hu=H_u+\eta$ and $Hv=H_v+\eta$ are the distances from bottom to water surface in the X and Y directions, respectively;
- $U=U(x,y,t)$ and $V=V(x,y,t)$ are the depth-averaged velocity components in the X and Y directions, respectively, in the Eulerian coordinate system (X,Y);
- g is the gravity acceleration;
- $\Omega=2\omega\sin\phi$ is the Coriolis force, where ω is the angular velocity of the earth and ϕ the latitude;
- $C_h=7{,}83\ln(0.3H/Z_o)$ is bottom stress Chezy coefficient, where Z_o is the bottom roughness;
- ρ_{water} is the water density;

- $\tau_{sx} = C_D \rho_{air} W_x^2$ is the wind stress component at water surface in the X direction, where W_x is the wind velocity component in this direction, measured 10 meters above water level, ρ_{air} is air density, and C_D is water surface stress coefficient;
- $\tau_{sy} = C_D \rho_{air} W_y^2$ is the wind stress component at water surface in the Y direction, where W_y is the wind velocity component is this direction;
- ε is the coefficient for the turbulent viscosity.

Since this is an initial-boundary value problem, it is necessary to specify these conditions. The physical boundary is divided in two sets: one is closed for the riverside and the other is opened, which is a virtual boundary delimiting the domain. In the closed boundary, it was defined null velocities and a zero level. In the opened boundary, it was specified two flow types: inflow and outflow. Therefore, it this mathematical model it was used two boundary conditions type:

- water level type;
- flow type.

3 Discretization: Semi-Implicit Scheme and Space-Staggered Grid

The numerical scheme was defined over a space-staggered grid Arakawa class C (Fig. 1). The decoupling of the variables that do not need to be defined on the same point of the grid results in a reduction of the processing time. Furthermore, this scheme maintains the same truncation error and has good conservative properties [2].

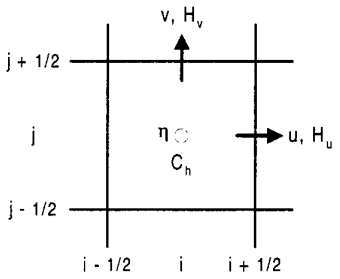


Fig. 1. Regular space-staggered grid Arakawa class C.

In the numerical scheme it was used a semi-implicit approach. Such decision was based on the fact that in a totally explicit approach the Courant-Friedrichs-Levy (CFL) conditions impose serious constraints in the time step. Furthermore, if sharp gradients occur, the numerical solution can develop oscillations or can be affected by the numerical viscosity [3]. On the other hand, in a fully implicit scheme, the constraints to the time step are removed because it is possible to generate unconditionally stable schemes. However, it is necessary to solve, simultaneously, a

large number of coupled nonlinear equations, and this solution depends on the available hardware. Even in this case, for accuracy, the time step can not be arbitrarily large.

An alternative strategy is used in this work. Some terms of the governing PDEs are discretized in an implicit way and the others in an explicit one. Using such approach in the SWEs discretization, the resulting algebraic linear equations systems (ALES) have structured pentadiagonal coefficient matrices. For instance, these matrices can be solved by iterative methods on the Krylov subspace or by direct methods. However, to improve the computational performance, these matrices can be partitioned using techniques such as ADI (Alternating Direction Implicit) to get structured tridiagonal matrices, which can be efficiently solved by the Thomas method.

A popular scheme that applies the ADI technique was developed by Leendertse ([4] and [5]) and used by others such as [1] and [6]. However, in Leendertse's approach, the continuity and momentum equations are alternately coupled to build the equations systems that, in this approach, result in non-symmetric matrices.

Since we goal to build symmetric, positive and defined (SPD) semi-implicit schemes, we applied the strategy developed by Casulli [3]. This strategy is a semi-implicit scheme for the momentum equations, where the water velocities are the unknowns, and for the continuity equation, where the levels are the unknowns. If the velocity terms $U^{n+1}_{(i\pm\frac{1}{2},j)}$ and $V^{n+1}_{(i,j\pm\frac{1}{2})}$ in the momentum equations are substituted in the continuity equation, one can obtain ALES with an SPD structure.

In this way, the level gradient and the stress term $\chi = g\sqrt{U^2+V^2}\,/\,C_h^2 H$ in the momentum equations (1) and (2), and the velocity divergent of the continuity equation (3) are dicretized implicitly. The convective terms, the stress coefficient, the Coriolis force term and the dissipation term are discretized explicitly.

Using an approach based on centered finite difference on a space-staggered grid, after some algebraic manipulation and reordering of terms, the discretization process results in two ALES (one for each direction) with SPD coefficient matrices. These systems are described below:

$$E\eta^{n+\frac{1}{2}}_{(i-1,j)} + C\eta^{n+\frac{1}{2}}_{(i,j)} + D\eta^{n+\frac{1}{2}}_{(i+1,j)} = f^{n}_{(i,j)} \,. \tag{4}$$

$$E\eta^{n+1}_{(i-1,j)} + C\eta^{n+1}_{(i,j)} + D\eta^{n+1}_{(i+1,j)} = f^{n+\frac{1}{2}}_{(i,j)} \,. \tag{5}$$

where the coefficients, for the first half time step $(n+\frac{1}{2})$ are:

$$C_{(i,j)} = 1 + \frac{g}{2}\left(\frac{\Delta t}{\Delta x}\right)^2 \left(\frac{Hu^n_{(i+\frac{1}{2},j)}}{1+\chi^n_{x(i+\frac{1}{2},j)}\Delta t} + \frac{Hu^n_{(i-\frac{1}{2},j)}}{1+\chi^n_{x(i-\frac{1}{2},j)}\Delta t} \right) . \tag{6}$$

$$E_{(i-1,j)} = D_{(i+1,j)} = -\frac{g}{2}\left(\frac{\Delta t}{\Delta x}\right)^2 \left(\frac{Hu^n_{(i-\frac{1}{2},j)}}{1+\chi^n_{x(i-\frac{1}{2},j)}\Delta t} \right) . \tag{7}$$

$$f^{n}_{(i,j)} = \eta^{n}_{(i,j)} - \left(\frac{\Delta t}{2\Delta x}\right)\left[\left(\frac{Hu^{n}{}_{(i+\frac{1}{2},j)}}{1+\chi^{n}_{x\,(i+\frac{1}{2},j)}\Delta t}\right)\left(Fu^{n-\frac{1}{2}}_{(i+\frac{1}{2},j)}\right) - \left(\frac{Hu^{n}{}_{(i-\frac{1}{2},j)}}{1+\chi^{n}_{x\,(i-\frac{1}{2},j)}\Delta t}\right)\left(Fu^{n-\frac{1}{2}}_{(i-\frac{1}{2},j)}\right)\right] - \left(\frac{\Delta t}{2\Delta y}\right)\left(Hv^{n}{}_{(i,j+\frac{1}{2})}v^{n}_{(i,j+\frac{1}{2})} - Hv^{n}{}_{(i,j-\frac{1}{2})}v^{n}_{(i,j-\frac{1}{2})}\right) \quad (8)$$

The coefficients for the second half time step (in the *Y direction*) are analogous. The velocities $U^{n+\frac{1}{2}}_{(i\pm\frac{1}{2},j)}$ and $V^{n+1}_{(i,j\pm\frac{1}{2})}$ are recovered after the calculation of the levels $\eta^{n+\frac{1}{2}}_{(i\pm1,j)}$ and $\eta^{n+1}_{(i,j\pm1)}$. The expressions $Fu^{n}_{(i\pm\frac{1}{2},j)}$ and $Fv^{n}_{(i,j\pm\frac{1}{2})}$ aggregate the explicit discretizations that correspond to the convective terms, wind stress at the surface and viscosity terms. The velocities are obtained using the following expressions:

$$U^{n+1}_{(i\pm\frac{1}{2},j)} = \frac{\Delta x Fu^{n}_{(i\pm\frac{1}{2},j)} - g\Delta t\left(\eta^{n+1}_{(i\pm1,j)} - \eta^{n+1}_{(i,j)}\right)}{\left(1+\Delta t\chi^{n}_{x\,(i\pm\frac{1}{2},j)}\right)\Delta x} . \quad (9)$$

$$V^{n+1}_{(i,j\pm\frac{1}{2})} = \frac{\Delta y Fv^{n}_{(i,j\pm\frac{1}{2})} - g\Delta t\left(\eta^{n+1}_{(i,j\pm1)} - \eta^{n+1}_{(i,j)}\right)}{\left(1+\Delta t\chi^{n}_{y\,(i,j\pm\frac{1}{2})}\right)\Delta y} . \quad (10)$$

It is possible to increase the time step of the numerical scheme when the convective terms in $Fu^{n}_{(i\pm\frac{1}{2},j)}$ and $Fv^{n}_{(i,j\pm\frac{1}{2})}$ are discretized using some methods such as upwind, semi-Lagrangian, and others.

4 Solution Strategy and Parallelization

Since in the ADI technique the time step is divided in two halves, it is necessary to obtain the numerical solution for the momentum and continuity in the *X* direction and for the momentum and continuity in the *Y* direction.

Thus, in this work, the parallelism was obtained splitting the domain in strips in the *Y* direction for the first half step and applying a pipelined algorithm in the *X* direction for the second half-step.

The direct Thomas method was chosen to solve the ALES $A\vec{\eta} = \vec{f}$ generated by (4), where A is the coefficient matrix, $\vec{\eta}$ is the vector of unknowns and $\vec{f}$ is the independent terms vector. Alternatively, it would be possible to apply an iterative method of the Krylov subspace, such as the conjugate gradient, using a Schwarz splitting as a preconditioner.

$$\begin{bmatrix} C_{(1,1)} & D_{(1,2)} & & \\ E_{(2,1)} & C_{(2,2)} & D_{(2,3)} & \\ & \ddots & \ddots & \ddots \\ & & E_{(n,n-1)} & C_{(n,n)} \end{bmatrix} \begin{bmatrix} \eta_{(1,1)} \\ \vdots \\ \vdots \\ \eta_{(n,1)} \end{bmatrix} = \begin{bmatrix} f_{(1,1)} \\ \vdots \\ \vdots \\ f_{(n,1)} \end{bmatrix}$$

Fig. 2. Structure of the tridiagonal system.

The matrix generated by expression (4) has the structure shown in Fig. 2, where the index *j* refers to column in the domain, and is solved by Thomas method in two stages [10]. The first stage is named forward and defined by:

$$D'_{(1,j)} = D_{(1,j)} / C_{(1,j)} \cdot \tag{11}$$

$$D'_{(i,j)} = D_{(i,j)} / \left(C_{(i,j)} - E_{(i,j)} D'_{(i-1,j)}\right), \quad i = 2,...,n. \tag{12}$$

$$f'_{(1,j)} = f_{(1,j)} / C_{(1,j)} \cdot \tag{13}$$

$$f'_{(i,j)} = \left(f_{(i,j)} - E_{(i,j)} f'_{(i-1,j)}\right) / \left(C_{(i,j)} - E_{(i,j)} D'_{(i-1,j)}\right), \quad i = 2,...,n. \tag{14}$$

Second stage, called backward, determines the solution for $\eta_{(i,j)}$ and is defined by:

$$\eta_{(n,j)} = f'_{(n,j)} \cdot \tag{15}$$

$$\eta_{(i,j)} = f'_{(i,j)} - D'_{(i,j)} \eta_{(i+1,j)}, \; i = n-1,...,1 \cdot \tag{16}$$

The solution for the PDEs in the *X* direction, where it is considered the domain splitting in strip, uses one halo with two lines of overlapping in the artificial boundary. With this procedure, it is guaranteed that the levels and velocities values are the same in the boundaries of the overlapped subdomains.

The solution in the *Y* direction, where the second half step is solved, is the part of the algorithm that demands more inter-processors communication, because the Thomas algorithm has global data dependency. In this stage, the matrices are split and allocated in different processors. To minimize the idleness of the processors and improve the method efficiency, it was used the pipelined Thomas, as defined in [7]. The scheduling of this algorithm is shown in Fig. 6.

4.1 Domain Splitting in the *Y* Direction

The core of the splitting strategy is the building of overlapped subdomains that satisfy discretized PDEs requirements concerning to mass conservation. Since the numerical scheme considers at most three cells to define some derivatives, the used overlapping must include at least two rows of the grid for each frontier side, which was artificially introduced by the domain splitting (Fig. 3).

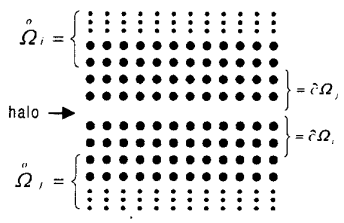


Fig. 3. Artificial boundary introduced by the domain splitting into subdomains.

In Fig. 3, $\partial\Omega_k$ and $\overset{o}{\Omega}_k$ denote, respectively, the boundary and the interior of a specific subdomain Ω_k. The global domain was stripped split in the Y direction, generating subdomains for the Guaíba River, which can be viewed in Fig. 4.

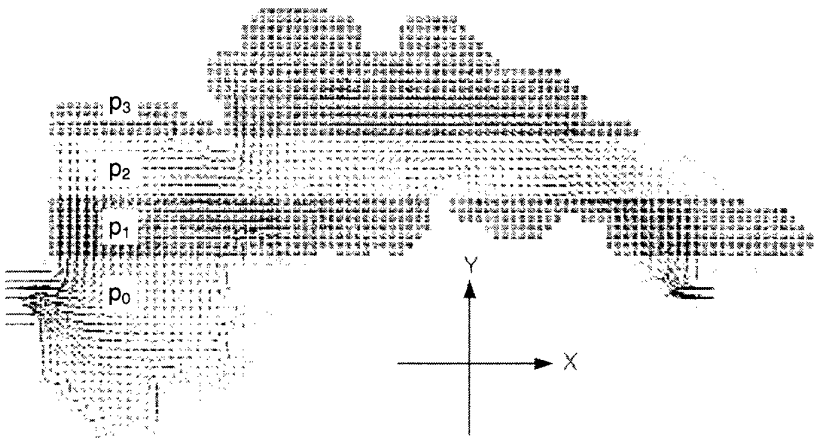


Fig. 4. Splitting of Guaiba River domain into subdomains strip type in the *Y* direction (results of an execution in 4 processors with a refinement of 2×2 for 10000 cycles).

Fig. 5. Velocity field of Fig. 4, where $|U_{max}| = 0.42$ m/s; $|V_{max}| = 0.36$ m/s; $|\eta_{max}| = 0.21$ m. The opened boundary conditions were defined as velocity type and, in the correspondent cells, $|U|$ was set as a constant equal to 0.4 m/s.

The results shown in Fig. 4 and Fig. 5 were obtained using the following parameters:

- $\eta_0 = 0.20$ m;
- $\Omega = 7.27E\text{-}5$ m^2/s;
- $C_h = 65.0$;
- $C_D = 3.0.E\text{-}6$;
- $W = 6.0$ m/s^2;
- $\varepsilon = 15.0$ m^2/s.

4.2 Pipelined Thomas in the *X* Direction

The direction *Y* of the ADI technique is solved in the second half step. However, due to dependencies existent in this in this direction, the solution of the matrices generated for one column can not be done in parallel. The coefficient matrices can be generated by each processor in parallel with the others, but the solution of the matrix in processor p depends on the solution of the matrix in processor p-1. These dependencies cause some idleness in the processors and make necessary some communication between them. At data transference, the processors are synchronized using blocking communication primitives, which simplify the synchronization process.

The solution for the matrices, which are subdivided and allocated in different processors, is done applying the Thomas algorithm in the pipelined form, as shown in Fig. 6.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
p_3				S_1F	S_2F	S_3F	S_1B	S_2B	S_3B			
p_2			S_1F	S_2F	S_3F			S_1B	S_2B	S_3B		
p_1		S_1F	S_2F	S_3F					S_1B	S_2B	S_3B	
p_0	S_1F	S_2F	S_3F							S_1B	S_2B	S_3B

Fig. 6. Scheduling of the pipelined Thomas with three columns (S_1, S_2 and S_3) in the X direction for a 4-processors machine (p_0, p_1, p_2 and p_3). S_iF and S_iB are, respectively, the forward and backward stages for the *i-th* column in each processor.

In the solution of a column, the p^{th} processor receives the values of $D'_{(n,p-1)}$ and $f'_{(n,p-1)}$ from the $(p-1)^{th}$ processor and proceeds to the forward stage in the matrix generated for its part of that column. After that, it sends the values of $D'_{(n,p)}$ and $f'_{(n,p)}$ to the $(p+1)^{th}$ processor. In a P processors machine, this algorithm considers that the forward stage is concluded for each tridiagonal matrix when the P^{th} processor receives the scalars $D'_{(n,p-1)}$ and $f'_{(n,p-1)}$ from the $(P-1)^{th}$ processor. Then the algorithm proceeds until it finishes every step of the forward. After that, the backward is started and, since in this stage the global dependency is restricted to the scalar $f''_{(n,1)}$, the P^{th} processor sends this scalar to the $(P-1)^{th}$ processor. This procedure is done for each column of the entire domain, obtaining the vector of unknowns for each tridiagonal matrix, which contains the values of the level in each cell. Finally, the levels in the lines of the halos are exchanged and the velocity values in the Y direction are calculated for the entire domain. Then, the algorithm is restarted for the next time step.

The scheduling in Fig. 6 represents a regular domain, where each column involves every subdomain. In an irregular domain, as in this work, some processors could start the forward stage of the columns beginning in their subdomains, what could reduce their idleness. However, it would increase the communication, because it would be necessary to send additional information about the column to be calculated.

5 Implementation and Results

The pipelined Thomas algorithm was implemented in C language using MPI message passing library and was run in a PC cluster based on Linux operating system. This cluster has four homogeneous nodes interconnected by a Fast Ethernet switch and a Myrinet switch. Each node consists of a Dual Pentium Pro 200 motherboard with 64 Mbytes of main memory.

The discretized region (Guaíba River) was approximated by cells, using different resolutions, where the dimensions of the cells vary from $\Delta x = \Delta y = 1000\ m$ to $\Delta x = \Delta y = 50\ m$. With this size of cell, the entire domain is composed of 300.000 cells. For each different resolution, the time step Δt is automatically calculated based on the Courant number and, when the real bathymetry is included, an interpolation is performed.

The Thomas algorithm is extremely efficient when it is applied to tridiagonal matrices and its complexity is $O(n)$. Thus, a parallel realization that uses this semi-implicit numerical scheme with two time steps must be efficient to be competitive.

One strategy to improve its performance is sending the data of more than one column per message. Povitsky, in his work [7] comments that it is possible to calculate the optimal number, as a function of the computation and communication times.

Several executions of this parallel implementation were performed varying the number of processors and the refinement in the PC cluster using the Fast Ethernet network. Fig. 7 and Fig.8 resumes some of these results.

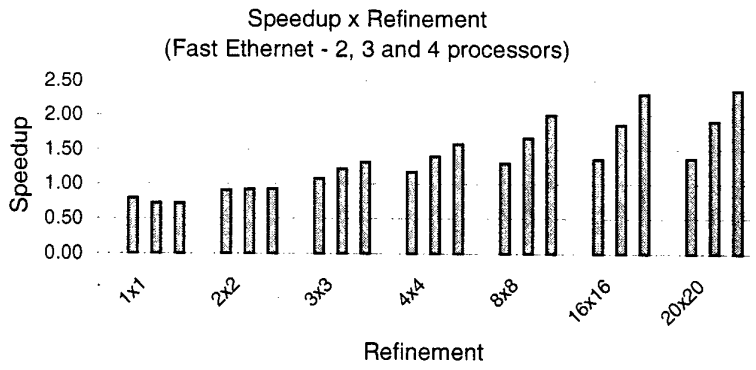


Fig. 7. Speedup × refinement using Fast Ethernet network (in each group of three bars, these bars represent the speedup obtained with 2, 3 and 4 processors, from left to right).

Fig. 7 shows that there is no speedup for refinements smaller than 3x3. This occurs because the sequential Thomas is a very efficient algorithm and the communication overhead in the parallel execution is bigger than the time saved with the distribution of the workload between the processors. However, when the refinement is bigger, the workload increases and the speedup grows, as is shown in Fig. 7.

Fig. 8 shows that the efficiency increases when the refinement grows, because the workload increases quadratically and the communication load has a linear growth. Furthermore, for each refinement, the efficiency decreases with more processors, as a consequence of the communication/computation rate. In other words, the communication for each processor remains the same but the computation is distributed among all the processors.

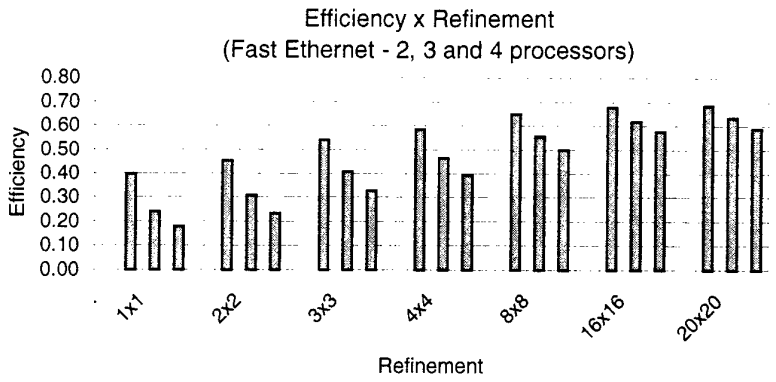


Fig. 8. Efficiency x refinement using Fast Ethernet network (in each group of three bars, these bars represent the speedup obtained with 2, 3 and 4 processors, from left to right).

6 Conclusions and Future Works

In this work it was presented a semi-implicit numerical scheme with tridiagonal and SPD algebraic equations systems. The parallelism was obtained using domain splitting for the X direction and a version of the pipelined Thomas algorithm for the Y direction. The numerical results are close to that obtained from data measurements of real events, although the computational model still requires some additional work in order to calibrate it to the real data of the Guaíba River. The numerical results of the sequential and parallel realizations are fully coincident.

Some works in development include a parallel version using Krylov-Schwarz methods, specially the preconditioned conjugated gradient. We have used a different discretization, similar to that described in [8] and [9], which results in a pentadiagonal SPD matrix, solved by conjugate gradient. We have already a parallel version and we are currently working in the Schwarz splitting as a preconditioner.

Once we have completed the implementation of the methods described, we will focus in the use of numerical schemes with generalized coordinates aiming to include adaptive meshes and dynamic load balancing; and the inclusion of mass transport equation to simulate the transport of industrial and domestic discharge.

Acknowledgments

This research was supported in part by CNPq and CAPES.

References

1. Casalas, A.B.: Modelo Matemático de Correntologia do Estuário do Rio Guaíba. Technical Report 12, Instituto de Pesquisas Hidráulicas da UFRGS, Porto Alegre (1985)
2. Messinger, F.: Numerical Methods: The Arakawa Approach, Horizontal Grid. Global and Limited-Area Modeling. Camp Springs: Academic Press (1998)
3. Casulli, V.: Semi-Implicit Finite Difference Methods for the Two-Dimensional Shallow Water Equations. Journal of Computational Physics, Vol. 86 (1990) 56-74
4. Leendertse, J.J.: A Water-Quality Simulation Model for Well-Mixed Estuaries and Coastal Seas: vol. I, Principles of Computation. Technical Report RM-6230-RC, The Rand Corp., Santa Monica California, (1970)
5. Leendertse, J.J., Gritton, E.C.: A Water-Quality Simulation Model for Well-Mixed Estuaries and Coastal Seas: vol. II, Computation Procedures. Technical Report R-708-NYC, The Rand Corp., Santa Monica California, (1971)
6. Kaplan, E. A.: Shallow Water Model Distributed using Domain Decomposition. Master's Thesis. Department of Numerical Analysis and Computing Science. The Royal Institute of Technology. Sweden. Available http://www.fing.edu.uy/~elias. (1998)
7. Povitsky, A.: Parallelization of the Pipelined Thomas Algorithm. NASA/CR-1998-208736 ICASE Report n° 98-48. Institute for Computer Aplication in Science and Engineering. NASA Langley Research Center. Hampton Virginia (1998)
8. Goossens, S.; Tan, K.; Roose, D. An Efficient FGMRES Solver for the Shallow Water Equations based on Domain Decomposition. Proc. 7th International Conf. On Domain Decomposition Methods in Scientific and Engineering Computing. Available http://www.ddm.org (1993).
9. Chefter, J.G.;Chu, C.K.; Keyes, D.E.; Domain Decomposition for the Shallow Water Equations. Available Proc. 9th International Conf. On Domain Decomposition Methods in Scientific and Engineering Computing. Available http://www.ddm.org/DD9. (1998)
10. Fletcher, C.A.J.: Computational Techniques for Fluid Dynamics 1. Springer Series in Computational Physics. Springer-Verlag. (1988)

Application of Parallel Simulated Annealing and CFD for the Design of Internal Flow Systems

Xiaojian Wang[1] and Murali Damodaran[1,2]

[1] Center for Advanced Numerical Engineering Simulations
Nanyang Technological University
Singapore, 639798
mxjwang@ntu.edu.sg
[2] Associate Professor, School of Mechanical and Production Engineering
mdamodaran@ntu.edu.sg

Abstract. Stochastic and deterministic optimization methods have their own advantages and shortcomings when applied to the problem aerodynamic shape design using computational fluid dynamics to evaluate the objective functions that are being optimized. Stochastic optimization methods such as Simulated Annealing and Genetic Algorithms appear to be robust techniques for solving different kinds of hard design optimization problems, such as discrete and non-convex problems. In this work the problem independent sequential and parallel Simulated Annealing implemented on a shared memory machine and applied to the problem of optimum shape design of internal flow systems whereby the objective functions that are to be optimized are analyzed using modern state-of-the art computational fluid dynamics flow solvers. The algorithms are parallelized using MPI libraries and the speedup and efficiency of the computational problem on a shared memory multi-processor computer examined. The results show that the parallel simulated annealing algorithm using message passing library can be efficiently applied to the problem of aerodynamic shape design optimization using computational fluid dynamics. The methods also demonstrate promising applications for solving complex multi-disciplinary design optimization problem.

1 Introduction

In the recent past, various optimization algorithms have been extensively applied for solving complex engineering design problems, such as multi-disciplinary optimization (MDO), which generally consists of discrete domain, non-convex space and is multi-model. Deterministic methods such as gradient based, direct search and sensitive analysis methods have been widely used in aerodynamic shape design and MDO problems [1]. The best feature of gradient based methods is that they are very efficient in searching for local minima of continuously differentiable functions. Stochastic algorithms, including simulated annealing (SA) and Genetic Algorithm (GA) are robust in searching for the global minimum of smooth and non-smooth functions are easy to implement and does not require gradient information. However stochastic design optimization method such as

SA and GA takes a far greater number of objective function evaluations [2] than that of deterministic methods to reach the global optima. Hence it is imperative that the associated computational costs and time be reduced by using parallel computing.

The main aim of current investigation is to address the speedup and efficiency of implementing a problem independent parallel Simulated Annealing method exploiting the shared memory $ccNUMA$ architecture of the SGI multiprocessor computer using the MPT v1.3 software, an implementation of the Message Passing Interface (MPI) library optimized for the SGI Origin 2000 series of multiprocessor computer. The design of high speed internal flow systems using compressible flow solvers based on modern state-of-the-art computational fluid dynamics and simulated annealing is chosen as the test problem to illustrate the application of parallel computing technologies on the multi-processor computer. Numerical studies of the internal flow systems design optimization problems show that parallel SA methods can serve as suitable candidates for obtaining optimum aerodynamic shape design configurations and the use of these parallel SA method can effectively reduce the wall-clock time in the calculation of design process. Although the methods are being developed for a single disciplinary setting in this study, the efforts in principle supports and can be further applied to complex MDO problems in a wide engineering area.

2 Sequential and Parallel SA Algorithms

Simulated annealing is a search of the solution space of a combinatorial optimization problem, with the goal of finding a solution of global minimum cost[3]. In SA, the optimization problem is simulated as an annealing process analogous to the situation when a metal is heated to a very high temperature to molten states and then cooled (annealed) till it reaches a solid state. The natural process which takes place in a slow cooling of the molten metal guarantee that the structure of the metal reaches the crystal structure corresponding to a minimum energy state. The final state depends on the cooling schedule. Comparing with GA, one major difference is that SA possesses a formal proof of convergence to global optima, although this proof relies on the use of a very slow cooling schedule and sufficiently large initial temperature. A typical procedure of SA consists of three major steps: a) moving from current solution to a new solution, b) computing the cost function of new solution, and c) making a decision for accepting or rejecting the new solution using given criteria. The procedure of the basic sequential SA algorithm as outlined in Table 1, which have been used widely for many different applications as outlined in Aarts and Korst[3], can be programmed easily as the steps are simple and concise.

It is well known that the major disadvantage of SA stems from the the large number computations that are necessary for evaluating objective functions. This may not be a big problem if the objective functions are simple and easily evaluated. However if the objective functions require sophisticated computational analysis methods such as CFD or finite element methods as is the case with en-

Table 1. Simulated Annealing method

Standard SA procedure
Start loop (1) for given temperature T_k
Start loop (2) for searching new solution at T_k
with a reasonable length of search (L) obtain new solution and cost
function F ΔF = new cost - current cost
if $\Delta F \leq 0$ or $e^{-\Delta F/T} > random()$
accept new solution and cost function F
end if
Continue loop (2)
Update $T_{k+1} = \alpha T_k$. If convergence criterion is satisfied, terminate loop (1)
Continue loop(1)

gineering design problems, then it is important to seek ways to reduce the computational effort by exploiting advances made in computer architecture. This has motivated the development of algorithms which can reduce the computational effort in the design process. One of the most widely used technique to speed up SA is to implement the algorithm on parallel computer architecture. It has been shown by Gallego et al.[4] that parallel implementation of SA not only results in speedup, but also increases the chances of global minimization. Bhandarkar and Machaka [5] states that there are three main parallelization strategies available at present for SA. These are broadly classified as functional parallelism within a move, control parallelism, and data parallelism.

In present work the strategy of multiple searches which can be classified as control and data parallelism is introduced and implemented on the 32-processor SGI Origin 2000 series of shared-memory multiprocessor computer. The divisions algorithm of Aarts and Korst [3] which is a control parallelism strategy is carried out by dividing the effort of generating a Markov main chain over the available processors. The main chain is divided into p subchains of length $N_{sp} = N_s/p$, where p is the number of processors and N_s is the length of the main chain. Each processor works on its associated subchain, and continues the generation of the subsequent subchain. The current subchain can be given as the outcome of the previous trial of the preceding subchains obtained by the same processor or found by choosing the best result from available processors. Table 2 outlines the procedure implementing this algorithm and in the rest of this paper this algorithm will be referred to as PSA1. This method is problem independent, and efficient. However, with the decrease of the length N_{sp}, which corresponds to the increase in the number of processors, the speedup will be reduced as mentioned in Deikmann et al. [6] and Laarhoven and Aarts [8].

The PSA1 algorithm can be further improved by using the clustered method in the lower temperature regions of the annealing process. Within the lower temperature, the acceptance ratio can be decreased very fast, as mentioned in Bhandarkar and Machaka [5]. The clustered method is an efficient method which

Table 2. Parallel SA algorithm (PSA1)

PSA1 Procedure
Initial multiple processors
Generate random seeds for each processor
Start loop (1) for given temperature T_k
Start loop (2) for searching new solution at T_k
set loop (2) length = L/p obtain new solution calculate cost
function F ΔF = new cost - current cost
if $\Delta F \leq 0$ or $e^{-\Delta F/T} > random()$
accept new solution and new cost function F
end if
Continue loop (2)
Choose the best cost function F and the solution from each processor, each procesor starts from the collected solution.
Update $T_{k+1} = \alpha T_k$. If convergence criterion is satisfied, terminate loop (1)
Continue loop(1)

works well at lower temperatures and is implemented in the following manner. The information on the objective function is gathered at the end of each search step and the solution is chosen randomly based on the accepted cost function. The acceptance ratio R_t is defined as the ratio of the numbers of the accepted moves to the numbers of all of the search moves. The incorporation of these modifications to PSA1 results in the PSA2 algorithm whose implementation is outlined in Table 3. It is worth noting that there are other methods which have been introduced by different researchers, such as the speculative method of Witte and Franklin [7] which has shown some promise for fine-grained multiple processors but not for coarse-gained computer system. Another method is the systolic method of Laarhoven and Aarts [8] which is also limited by the N_{sp}, and it has been combined with other methods to implement parallel strategies. In the present study, the parallel methods (PSA1 and PSA2) are used as basic approaches for implementing the parallel SA in the design optimization of aerodynamic shapes of nozzles and diffusers.

3 Implementation of Parallel SA on SGI Origin 2000 Computer Using MPT

The most widely used programming languages and libraries on parallel computers are High Performance Fortran (HPF), Parallel Virtual Machine (PVM)and the Message Passing Interface (MPI). The HPF is still being developed improve data parallelization. Both PVM and MPI system transparently handles message routing, data conversion for incompatible computer architectures and other tasks necessary for operations in a heterogeneous or a homogeneous computing network. Although PVM was one of the earliest parallel software environment

Table 3. Parallel SA algorithm (PSA2)

PSA2 procedure
Initial multiple processors
Generate random seeds for each processor
Start loop (1) for given temperature T_k
Start loop (2) for searching new solution at T_k
set the length of chain = L/p obtain new solution and cost
function F ΔF = new cost - current cost
if $\Delta F \leq 0$ or $e^{-\Delta F/T} >$ random()
accept new solution and cost function F
end if
if the acceptance ratio $\leq R_t$
Randomly gather the accepted solution from each processor.
Reset the length = L
end if
Continue loop (2)
Choose the best cost function F and the solution from each processor, each procesor starts from the collected solution.
Update $T_{k+1} = \alpha T_k$. If convergence criterion is satisfied, terminate loop (1)
Continue loop(1)

to be developed and which is also continually being improved, MPI has fast become the industry standard for parallel software environment which is available on most parallel computer platforms.

For the parallelization of SA in this study two parallel software environment tools which were made available under the framework implementing the MPI libraries. One is LAM v6.1 [9], a parallel software tool and processing environment for a network of independent computers, and heterogeneous computer networks and developed by the Ohio Supercomputer Center at Ohio State University [9]. The other tool is the MPT v1.3 [10], which was developed by SGI as a compatible message passing tool on SGI machines. The MPI implementations used is the MPT v1.3 which is fully compliant with the current MPI 1.2 specification. Preliminary tests show that MPT v1.3 is more efficient than Lam v6.1 on SGI IRIX 6.5 machine. More information and details on the MPI libraries and parallel applications can be found in Pacheco[11] and in Foster[12].

4 Optimum Shape Design of Internal Aerodynamic Flow Systems

The application of parallel SA algorithms PSA1 and PSA2 and the CFD flow solver which is used to calculate the objective functions are described in following sections in the design of optimal shapes of nozzles and diffusers which are aerodynamic devices facilitating high speed internal flow systems commonly

used in rockets, wind-tunnels, jet engines and large scale industrial applications using jet flow technologies The design test problems are outlined briefly first and then followed by presentation of computed results and discussions.

4.1 Diffuser Shape Design - Design Test Case 1

This test case is concerned with optimal shape design of an axisymmetric diffuser for which the design flow field condition and the pressure distribution along the centerline are defined. The aim is to find a shape of the diffuser which will satisfy this design flow condition. The objective function which has to be minimized is expressed in normalized form as follows:

$$F(X) = \frac{1}{\rho_0 u_0^2} \int (P - P_t)^2 dx, \tag{1}$$

where P_t is the target pressure distribution and P is the initial or evolving pressure distribution defined along the length of the diffuser center-line; ρ_0 is the density, u_0, P_0 are stagnation flow conditions which are taken as reference values for scaling flow quantities in internal flow simulations using CFD analysis and X is the vector of design variables, and $X = (x_1, x_2, ...x_n)$. The inflow Mach number is supersonic at 1.5. Other pertinent details of this problem can be found in Hoffman and Chiang [13]. This test problem is chosen because it is a representative design problem where the objective function is non-smooth in view of the presence of a a shock wave which is a flow discontinuity. To start the design process the target pressure distribution or flow condition and a guess of the initial starting shape of the diffuser is specified. The target pressure is that defined for a diffuser which is generated by the following distribution of cross-sectional area ($S(X)$) along the length of the diffuser:

$$S(X) = 1.398 + 0.349 tanh(0.8X - 4). \tag{2}$$

The initial shape of the diffuser is that of a conical frustum obtained by connecting the radii at the inflow and outflow with a straight line. The starting and target pressure distribution along the length of the diffuser are shown in Fig. 1(a).The corresponding shape of the starting and the target diffusers are shown in Fig. 1(b). To initiate the design minimization process the nozzle shape is defined in terms of design variables. The curve defining the shape of the diffuser is parameterized by Berstein basic functions as outlined in Faux and Pratt[14] and defined as follows:

$$Y = \Sigma_{i=0}^{6} \frac{6!}{(6-i)!i!} u^i (1-u)^{6-i} r_i. \tag{3}$$

For this case the CFD analysis is based on the quasi-one-dimensional Euler equations for inviscid nonlinear compressible flow which can be solved by the finite-difference methods based on the Steger-Warming upwind scheme (first order TVD). The governing equations and details of the numerical algorithm

can be found in Hoffmann[13] which readers can consult and hence details of the CFD methods are not repeated here.

The inverse design of the diffuser is carried out using parallel SA algorithms PSA1 and PSA2. The diffuser shape is represented using eleven design variables with blending functions defined by Eq. (3). The flow field corresponding to the diffuser shape defined by Eq. (2) is computed using 51 uniformly distributed points in the flow direction and for the specified design values and inflow conditions to obtain the target flow conditions. The design inflow conditions are used to calculate the flow field inside the starting shape(initial guess)of the diffuser using the Euler equations. Then Parallel SA is used to minimize the objective function until the desired shape of the diffuser is attained. By comparing the final shape of the designed diffuser with the shape defined in Eq. (2) (i.e. the shape corresponding to the target pressure distribution), it is possible to make assessments on the effectiveness of the parallel SA.

4.2 Tunnel Wall Shape Design - Design Test Case 2

The second test case is concerned with the application of parallel SA and 2-dimensional CFD analysis to design the shape of the lower wall of a converging nozzle inside which there is high speed compressible flow. The initial shape of the lower wall is a straight line and oriented like a wedge (see Fig. 2(a)). The top wall is parallel to the direction of flow along the x-direction. The shape of the bottom wall which looks like a wedge generates a strong reflective shock wave between bottom and top walls, see Fig. 2(a). The objective of the design is to re-design the shape of the lower wall so as to eliminate or weaken the shock wave. Since it is not possible to have a priori knowledge of the type of target (or desired) pressure distribution P_t corresponding to a pressure distribution which does not create a shock at the lower wall, the knowledge from the basic theory of gas-dynamics which states that a compressible flow a gradually turning wall (compressive) can produce a weak shock forms the basis for the selection of the target pressure distribution by computing the compressible flow past the lower wall which is curved according to the equation $y = x^{3.5}$.

In this case, the objective function is defined as

$$F(X) = \frac{1}{P_0} \int (P - P_t)^2 dx, \tag{4}$$

where P_0 is the pressure of the inflow. The integration of the difference between the target and design pressure distribution is calculated along the bottom wall line where the shock effects are felt the strongest. The inflow Mach number is 2.2. The shape of the lower wall is matched using cubic splines[15], with continuity of function, its first and second derivative at the boundaries. Four design variables are chosen for the design optimization namely two first derivatives at the inlet and outlet boundaries, and two height parameters, y_1 and y_2 at $x_1 = 5.0$ and $x_2 = 8.0$, respectively. Here the CFD method for solving the Euler equations to evaluate the objective functions is based on an implicit high resolution LU-SGS TVD scheme as outlined in Yoon et al.[16] and Yee et al.[17]

which readers can peruse for details. A structured grid of 101x40 points is used for calculating the compressible flow field using these CFD methods for each evolving design shape till the final shape is attained.

4.3 Nozzle Shape Design - Design Test Case 3

The final test case is concerned with the optimal design of a nozzle shape which maximizes the thrust of the nozzle. As nozzles are used in many industrial applications from rocket engines, water-jet cutting, fire-man's hose, the computation of the thrust requires the integration of the forces induced by fluid mechanics and this involves an integration of the pressure in the direction normal to the flow direction. The objective function is defined as

$$F(X) = \int (P/P_0)dy. \tag{5}$$

The integration is calculated along the surface of nozzle wall. The flow field is calculated using the CFD method used in case 2 except that for this case the full Navier-Stokes Equations are used for the CFD analysis. Here the CFD method for solving the Navier-Stokes equations to evaluate the objective functions is based on the same scheme as outlined in Test case 2. A structured grid of 101x40 girds with clustered grids near the nozzle wall surface is used for the CFD analysis. The inflow Mach is 4.84, the values of the radius are 0.5m at the inlet cross section and 1.0m at the exit. The length of nozzle is 2.52m. The curve defining the shape of the nozzle is defined using a cubic spline. Two design variables, namely inlet expansion half angle and outlet expansion half angle of the nozzle wall, is adopted in the nozzle optimization.

5 Results and Discussions

5.1 Results from Design Test Case 1

Design Test case 1 forms the basis for a comparative study on the performance of using the parallel SA methods, i.e. PSA1 and PSA2 which can be considered as user-written parallel codes using MPT library functions. The SA parameters are chosen as follows: the initial cooling temperature is set to a value at which the initial cooling acceptance ratio is greater than 0.95; the length N_s for the cooling scheme is first tested and taken to be as short as possible for a single processor; a constant cooling scheme, $T_{k+1} = \alpha T_k$, is chosen with $\alpha = 0.1$. Based on the experience gained in applying SA to aerodynamic design problems the value for α ranges between 0.30 and 0.05 [18]. The parameter R_t has a value ranging between 0.4 and 0.6. As many multi-processors on the SGI machine are used for the implementation of the PSA1 and PSA2 algorithms, the termination criteria to stop the program is defined in such a way that iteration is terminated if the objective function reaches a value lower than F_{min} = 1.4E-4 on the collected result of multiple processors.

On Fig. 1(a) a comparison is made of the target pressure distribution and the final pressure distribution achieved by inverse design process from the starting pressure distribution. Also shown on this figure is the starting pressure distribution which corresponds to the starting shape of the diffuser. Fig. 1(b) shows the initial shape and compares the final optimized shape of the diffuser (attained by inverse design) with the shape of the diffuser corresponding to the target pressure distribution specified for this inverse design problem i.e. Eq. (1). The solid lines in these figures represent the targets which the final designs ought to converge to after the application of parallel SA algorithms PSA1 and PSA2. It can be seen from these figures that parallel SA has done a good job in carrying out the inverse design problem.

Fig. 1(c) shows the convergence histories (for processor No. 0 only) of the objective function as it gets minimized from its initial values to the final designed value which should correspond to zero as a result of using the parallel SA algorithms PSA1 and PSA2 on 1 to 32 processors. The plots show the variation of the objective function value vs. the number of iterations for the three cases corresponding to the application of the standard SA (SSA) and the parallel SA (PSA1 and PSA2). It can be seen that the number of iterations to reach the final optimal design shape according to the termination criteria reduces ten-fold if MPT is used as opposed to the sequential parallelization option on the SGI machine.

Tables 4 and 5 show some measures of performance for the implementation of parallel SA on the tightly coupled processors which make up the 32-processor SGI Origin 2000 machine. In these tables p refers to the number of processors that are used at one time for doing the computational task, M_p is the number of evaluations of objective function, and t_p represents the wall-clock time for the computational task, S_p and S_{pi} are the realistic and ideal speedup (without communication overhead of multiple processors), E_p and E_{pi} are the realistic efficiency and ideal efficiency, respectively. It appears that the parallel SA algorithm PSA1 shows efficient speedups if 1 to 16 processors are used in the computational task and also registers a reduction in the wall clock time from 4 to 0.55 hours. This also shows that using 32 processors will result in only marginal benefits as the wall clock time has reduced to 0.47 hours. It can be seen from these results that the number of function evaluation decreases for each processor if more processors are used in the computation. It can also be seen from Table 5 that the parallel SA algorithm PSA2, reduced the value of M_p from over 6000 on single processor to around 500 on 32 processors.

Tables 4 and 5 compare the wall clock time for PSA1 and PSA2 using different numbers of processors in the computation. It can be concluded that the speedup and efficiency using PSA2 is generally better than those of using PSA1. However the communication overhead, obtained by subtracting realistic curve from ideal curve of PSA2, $E_{pi} - E_p$, is higher than PSA1.

5.2 Results of Test Case 2

Based on the results of Test Case 1, the parallel SA algorithm PSA2 is used for the minimization of the objective function of Eq. 4. This is a more intensive computation than test case 1 because this is a 2-dimensional problem and solves the 2-dimensional Euler equations for CFD analysis. For this design calculation, some of the simulated annealing parameters appearing in the cooling schedule are changed i.e. $\alpha = 0.15$ and the tolerance criteria is set at F_{min} = 4.0E-2. Table 6 shows that the measures of performance such as speedup and efficiencies attained by using 1, 4 and 8 processors for this calculation. It can be seen that the number of evaluations of the objective functions decreased from 653 on single processor to 185 on 4 processors and to 77 on 8 processors. The initial shape of the lower wall of the nozzle and the computed flow field which contains a strong oblique shock wave which is reflected off the top wall can be seen in Fig. 2(a) which shows the contours of the normalized pressure in the flow field. This flow field is the starting flow field condition used to initiate the design process so that the algorithm PSA2 can find the shape of the lower wall which will eliminate the shock wave for the same flow conditions. It can be seen from Fig. 2(b) that the PSA2 algorithm has designed an optimal lower wall shape which has a close agreement with the target flow conditions. Fig. 2(c) shows the contours of the normalized pressure in the flow field corresponding to the flow past the optimized lower wall shape for the same flow conditions. It can been seen that the shock wave has been eliminated from the flow field. The convergence history of the optimization by PSA2 algorithm on 1, 4 and 8 processors is shown in Fig. 2(d).

Table 4. Measures of Performance on SGI, Design Test Case 1 Using PSA1

p	M_p	$t_p(hour)$	$S_p(t_1/t_p)$	$S_{pi}(M_1/M_p)$	$E_p(S_p/p)$	$E_{pi}(S_{pi}/p)$
1	6161	4.00				
2	4040	2.88	1.39	1.53	0.695	0.765
4	1740	1.36	2.94	3.54	0.735	0.885
8	1300	0.984	4.00	4.74	0.500	0.595
16	756	0.549	7.29	8.15	0.456	0.509
32	661	0.477	8.39	9.32	0.262	0.291

5.3 Results of Test Case 3

Design Test Case 3 is the most computationally intensive of all the three cases considered in this study in view of the fact that this is constrained design problem and the objective function is evaluated using the CFD solver solving the Navier-Stokes equations. The PSA2 algorithm is used for the optimization and the termination criteria for the convergence is set such that the residual $|F(X)_{n+1} - F(X)_n| \leq 0.001$ is satisfied. The α for the cooling schedule is set to

Table 5. Measures of Performance on SGI, Design Test Case 1 Using PSA2

p	M_p	$t_p(hour)$	$S_p(t_1/t_p)$	$S_{pi}(M_1/M_p)$	$E_p(S_p/p)$	$E_{pi}(S_{pi}/p)$
1	6161	4.00				
2	3697	2.72	1.47	1.66	0.733	0.83
4	1519	1.24	3.24	4.06	0.810	1.02
8	1134	0.898	4.45	5.43	0.556	0.679
16	595	0.468	8.57	10.4	0.535	0.650
32	562	0.439	9.12	11.0	0.285	0.344

Table 6. Measures of Performance on SGI, Design Test Case 2 Using PSA2

p	M_p	$t_p(hour)$	$S_p(t_1/t_p)$	$S_{pi}(M_1/M_p)$	$E_p(S_p/p)$	$E_{pi}(S_{pi}/p)$
1	653	35.71				
4	185	12.41	2.88	3.53	0.719	0.883
8	77	5.4	6.61	8.4	0.826	1.02

Table 7. Measures of Performance on SGI, Design Test Case 3 Using PSA2

p	M_p	$t_p(hour)$	$S_p(t_1/t_p)$	$S_{pi}(M_1/M_p)$	$E_p(S_p/p)$	$E_{pi}(S_{pi}/p)$
1	67	20.95				
4	21	6.73	3.11	3.19	0.778	79.8
8	19	6.17	3.40	3.53	0.425	0.441

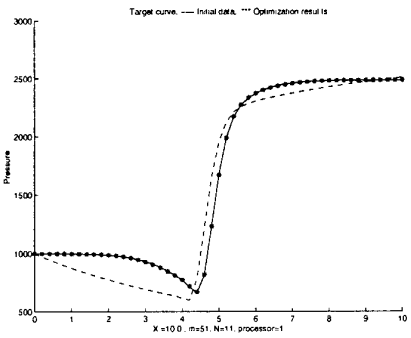


Fig. 1(a). Pressure distribution

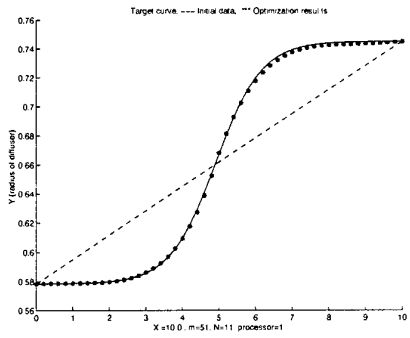


Fig. 1(b). Diffuser contours

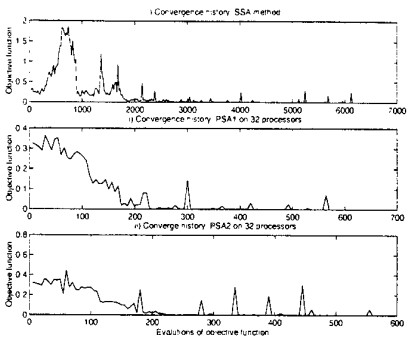


Fig. 1(c). Convergence history

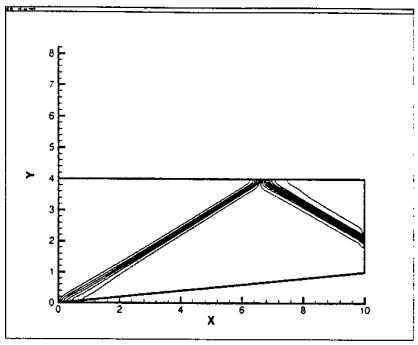


Fig. 2(a). Initial flow field

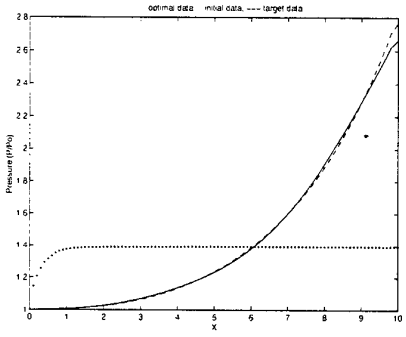


Fig. 2(b). Pressure distributions

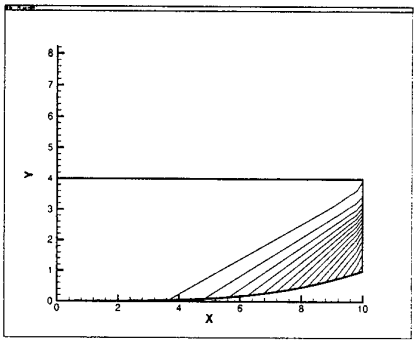


Fig. 2(c). Optimized flow field

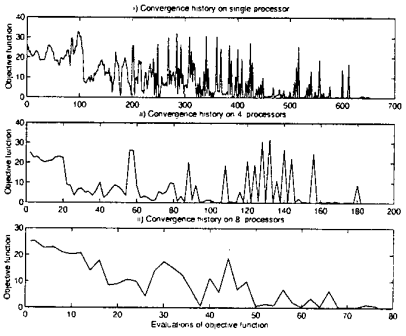


Fig. 2(d). Convergence history

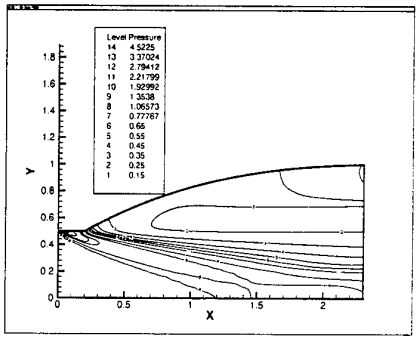


Fig. 3(a). Initial flow field

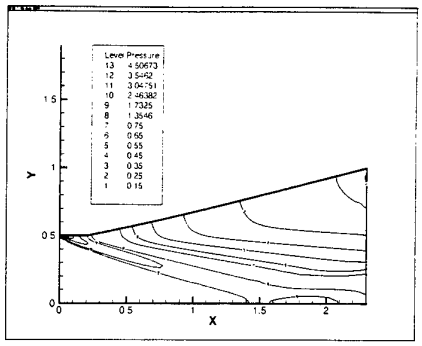


Fig. 3(b). Optimized flow field

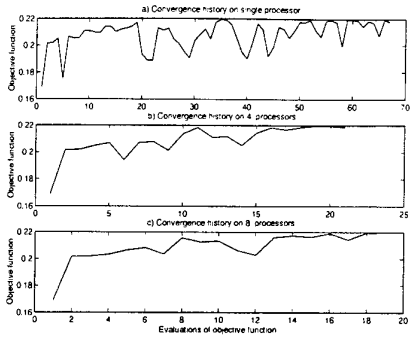


Fig. 3(c). Convergence history

be 0.25. Table 7 shows that the measures of performance such as speedup and efficiencies attained from using 1, 4 and 8 processors for this calculation. Fig. 3(a) shows the computed local pressure contours of the original nozzle shape which is delivering a certain amount of thrust. This flow field corresponds to the starting flow field for the optimization studies. Fig. 3(b) shows the computed flow field corresponding to the optimized nozzle shape which has been designed from the original shape to satisfy the thrust delivery constraint. Fig. 3(c) shows the convergence histories for Case 3. This case shows that high speedup also can be achieved, which is the same as that in the inverse design optimization on Case 1 and 2.

6 Conclusions

In conclusion it can be observed from the design test cases 1,2 and 3 that as the number of design variables pertaining to an optimization problem reduces (11 design variables for case 1 to 4 design variables for case 2 to 2 design variables for case 3) fewer number of processors are required for maintaining higher efficiency. The three test cases show that with the increase in the number of design variables for a design problem, larger N_s value is needed i.e. the length of the Markov chain thereby suggesting that more processors can be used efficiently in design calculation to reduce wall-clock time. The parallel SA approaches implemented in this investigation with applications in the shape designs of internal flow using CFD have demonstrated that wall-clock times can be reduced considerably by reduction in the number of evaluations of objective function for each processor. The PSA2, which couples the division and cluster methods, performed slightly better than PSA1, which uses division method only. Although the speedup limitation exists while more processors are used, a reasonable number of processors can be chosen according to the efficiency desired by the designer. The case studies demonstrate promising application in more complex problems, such as MDO problems, which are time-consuming when solving using analysis methods for

coupled multi-disciplinary engineering problems. Further studies including the use of adaptive SA and hybrid optimizer, will be carried out in the near future to enhance the performance of current parallel SA.

References

1. Sobieszczanski-Sobieski, J., Haftka, R.T.: Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments. AIAA Paper 96-0711, 34th Aerospace Sciences Meeting and Exhibit, Reno, Nevada (1996)
2. Frank, P.D., Booker, A.J., Caudell, T.P., Healy, M.J.: A Comparison of Optimization of Optimization and Search Methods for Multidisciplinary Design. AIAA Paper 92-4827 (1992)
3. Aarts, E., Korst J.: Simulated Annealing and Boltzmann Machines, A Stochastic Approach to Combinatorial Optimization and Neural Computing. John Wiley and Sons (1989)
4. Gallego, R.A., Alves, A.B., Monticelli, A.A., Romero, R.: Parallel Simulated Annealing Applied To Long Term Transmission Network Expansion Planning. IEEE Transactions on Power Systems, Vol. 12, No. 1.(1997)
5. Bhandarkar, S.M., Machaka, S.: Chromosome Reconstruction from Physical Maps Using a Cluster of Workstations. Journal of Supercomputing, Vol. 11, (1997) 61-87
6. Diekmann, R., Luling, R., Simon, J.: Problem Independent Distribution Simulated Annealing and its Applications. Lecture Notes in Economics and Mathematical Systems 396, Applied Simulated Annealing, Springer-Verlag (1993)
7. Witte, E.E., Franklin, M.A.: Parallel Simulated Annealing Using Speculative Computation. IEEE Transactions on parallel and distributed systems, Vol. 2, No. 4 (1991) 483-493
8. Laarhoven, P.J.M. van, Aarts, E.H.L.: Simulated Annealing: theory and applications. Norwell, MA, U.S.A., Kluwer Academic Publishers (1987)
9. MPI Primer/Developing with LAM. Ohio Super computer Center, The Ohio State University (1996)
10. Message Passing Toolkit: MPT Programmer's Manual. Document Number 007-3687-002, SGI Inc. (1999)
11. Pacheco, P.S.: Parallel Programming with MPI. Morgan Kaufmann Publishers, Inc. San Francisco, California (1997)
12. Foster, I.: Designing and Building Parallel Programs. On-line book, *www – unix.mcs.anl.gov./dbpp/*, also published by Addison-Wesley Publishing (1995)
13. Hoffmann, K.A., Chiang, S.T.: Computational Fluid Dynamics For Engineers. A Publication of Engineering Education System, Wichita, Kansas (1993)
14. Faux, I.D., Pratt, M.J.: Computational Geometry for Design and Manufacture. Ellis Horwood Limited Publicashers (1979)
15. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in Fortran: The Art of Scientific Computing. Second Edition, Press Syndicate of University of Cambridge (1997)
16. Yoon, S., Jameson, A.: Lower-Upper Symmetric-Gauss-Seidal method for the Euler and Navier-Stokes Equations. AIAA Journal. Vol 26, No.9, (1988) 1025-1026
17. Yee, H.C., Harten, A.: Implicit TVD Schemes for Hyperbolic Conservation Laws in Curvilinear Coordinates. AIAA Journal, Vol. 25, (1987) 266-274
18. Aly, S., Marconi, F., Ogot, M., Peiz, R., Siclari, M.: Stochastic Optimization Applied To CFD Shape Design. AIAA paper 95-1647 (1995)

Parallel Algorithm for Fast Cloth Simulation*

Sergio Romero, Luis F. Romero, and Emilio L. Zapata

Universidad de Málaga, Dept. Arquitectura de Computadores, Campus Teatinos, 29071, Málaga, Spain,
ezapata@ac.uma.es

Abstract. The computational requirements of cloth and other non-rigid solids simulations are high and often the running time is far from real time simulations. In this paper, we present an efficient parallel solution of the problem, which is a consequence of a wide analysis of the data distributions and the parallel behavior of some different iterative system solvers and preconditioners. Our parallel code combines data parallelism with task parallelism, achieving a good load balancing and minimizing the communication cost. The execution time obtained for a typical problem size, its superlinear speed-up, and the isoscalability shown by the model, will allow to reach real-time simulations in sceneries of growing complexity, using the most powerful parallel computers

1 Introduction

Cloth and flexible material simulation is an essential topic in computer animation of realistic virtual humans and dynamic sceneries. New emerging technologies, as interactive digital television and multimedia products make necessary the development of powerful tools able to perform real time simulations. There are many approaches to simulate flexible materials. Geometrical models are usually considered the fastest but they require a high degree of user intervention, making them unusable for interacting applications. In this paper, a physically-based model, that provides a reliable representation of the behavior of the materials (e.g. garments, plants), has been chosen. In a physical approach, clothes and other non-rigid objects are usually represented by interacting discrete components (finite elements, springs-masses,patches) each one numerically modeled by an ordinary differential equation (1), where x is the vector of positions of the masses M. The derivatives of x, are the velocities $\dot{x} = v$ and the accelerations $\ddot{x}$.

$$\ddot{x} = M^{-1} f(x, \dot{x}) \qquad \frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ M^{-1} f(x, v) \end{pmatrix} \tag{1}$$

Also, in most energy-, forces- or constraints-based formulations, equations contain non-linear components, that are typically linearized by means of a Newton method. The use of explicit integration methods, such as forward Euler and Runge-Kutta, results in easily programmable code and accurate simulations [3],

* Candidate to the Best Student Paper Award

and have been broadly used during the last decade, but recent works [1] demonstrate that implicit methods overcome the performance of explicit ones, assuming a non visually perceptible lost of precision. In the composition of virtual sceneries, appearance, rather than accuracy is required, so, in this work, an implicit technique, backward Euler method, has been used. In section 2, a description of the models used and the implementation technique for the implicit integrator is presented. Also, the resolution of the resulting system of algebraic equations by means of the Conjugate Gradient method is analyzed. In section 3, the parallel algorithm and the data distribution technique for a scenery is shown. Finally, in section 4, we present some results and conclusions.

2 Implementation

To create animations, a time-stepping algorithm has been implemented. Every step is mainly performed in three phases: computation of forces, determination of interactions and resolution of the system. The iterative algorithm is shown below:

```
do {
   computeForces();
   collisionDetection();
   solveSystem();
   updateSystem();
   time = time + timeStep
}while(time<FinalTime)
```

The `updateSystem` procedure computes the new state, made up of the position and velocity of each element, calculated from the previous one. `computeForces`, `collisionDetection` and `solveSystem` stages are described below.

2.1 Forces

Forces and constraints are evaluated on every discrete element in order to compute the equation coefficients for the second Newton's law. Our model considers both spring-mass discretization of 2D-3D objects, and triangles patches for the special case of 2D objects like garments. The particular forces considered are: Visco-Spring forces mapped in the grid for the former model; and Stretch, Shear and Bend forces for the later. In both cases, gravity and air drag forces have been also included. The backward Euler method approximates the second Newton law by the equation (2) in the k-th time step,

$$\Delta v = \Delta t \cdot M^{-1} \cdot f(x_{k+1}, v_{k+1}) = \Delta t \cdot M^{-1} \cdot f(x_k + \Delta x, v_k + \Delta v) \qquad (2)$$

This is a non-linear system of equation which has been time-linearized by one step of the Newton method as follows:

$$f_{k+1} = f(x_{k+1}, v_{k+1}) = f_k + \left|\frac{\partial f}{\partial x}\right|_k \Delta x + \left|\frac{\partial f}{\partial v}\right|_k \Delta v \qquad (3)$$

being $\Delta x = \Delta t(v_k + \Delta v)$. An energy function E_α for every discrete element α is analytically described; the forces acting on particle i are derived from $f_i = \sum_\alpha -\partial E_\alpha / \partial x_i$ and the arising coefficients from the analytical partial derivations in equation 3 has been coded for its numerical evaluation. All above gives a large system of algebraic linear equations with a sparse matrix of coefficients.

2.2 Collisions

To detect possible interactions and forbidden situations, like colliding surfaces or body penetrations, we have used a hierarchical approach based on bounding-boxes. In these cases, we introduce additional forces to maintain the system in a legal state. This forces will be included as described in the previous section. In the case of cloth simulation, the self-collision detection and the human-cloth collisions may be critical and the computational cost can be extremely high. [4]

2.3 Solver

In the `solveSystem` procedure, the unknowns Δv are computed. As stated above, implicit integration methods requires the resolution of a large, sparse linear system of equations that must be simultaneously fulfilled. An iterative solver, the preconditioned conjugate gradient method, has proven to work well, in practice. This method requires relatively few, and reasonably cheap, iterations to converge [1]. The choice of a good preconditioner can result in a significant reduction of the computational cost in this stage. In this work, five preconditioners have been studied. We have tested every preconditioner for a wide set of sceneries. In every case, Block-Jacobi has shown to be the fastest although the required number of iteration for a given tolerance is larger than that in the incomplete factorization techniques. In table 1, the number of iterations and the elapsed execution time for each preconditioner of an example simulation is presented. We have chosen the Block-Jacobi preconditioner for our algorithm, not only for the execution time but also for its better parallel behavior [2].

Table 1. Elapsed Time and Number of Iterations of CG

Preconditioners	Number of iterations	Time
Jacobi	15022	28.118
Block-Jacobi	10001	21.632
$(L+D)D^{-1}(D+U)$	5298	22.240
Incomplete-LU	4326	24.651
I-Cholesky	4319	25.483

3 Parallelization

The parallelization of the model has been performed on a SGI Origin2000, a NUMA multiprocessor architecture. Automatic parallelizing tools are not able to extract enough parallelism from this kind of problems. We have used a SGI specific cache-coherent shared memory programming model, and a data parallelism strategy. Task parallelism has been also considered for the collision detection stage. The distribution of the objects in a scenery between the processors is performed using a proportional rule based on the number of particles/triangles. The redistribution and reordering of the particles/triangles inside an object among the assigned processors have been performed using domain decomposition methods. In figure 1, the non-zero coefficient of the system matrix for three different reordering of the particles/triangles is shown. It can be observed that the stripped ordering results in a thin banded diagonal which will result in the parallel distribution with less communication expenses. The Multiple Recursive Distribution (MRD), has more locality, which will result in a better cache usage. The choice of the method will depend of the scenery and the computational platform.

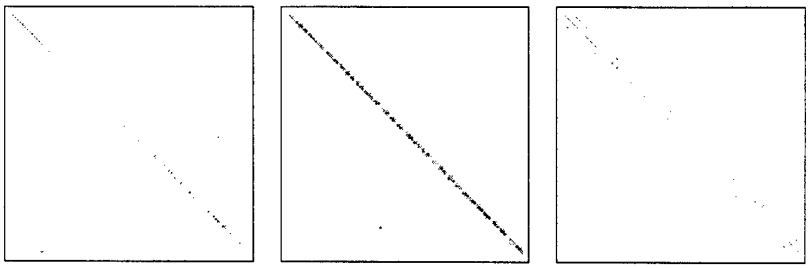


Fig. 1. Coefficient matrices for original, MRD, and stripped ordering.

3.1 Solver

The PCG algorithm has been parallelized following a well-known strategy in which the successive parts of the vectors and the properly aligned rows of the matrices are distributed among the processors. Using this scheme and the Chronopoulos and Gear variant of PCG [2], very few messages and synchronization points are required along the iterative process. The parallelization of the operations have been performed by using the data distribution previously computed for the forces calculation stage. In this paper, an innovative strategy for the parallel implementation of the PCG algorithm for cloth simulation is proposed. In our scheme, global synchronization points (GSPs) and message exchanges (MEs) among the processors in the iterative process can be handled in

three different ways. First, GSPs and MEs can be eliminated when they are not required (for example, when two sets of processors are solving the corresponding subsystems for separate pieces of clothes). In this case, the two subsystem become independent, and the inherent parallelism of different elements in a scenery is considered. Second, GSPs and MEs are used in the usual way. This will be the general case for the parallelization inside a garment. A third case has been tested, keeping every GSP in the algorithm, but leaving the messages to be sent without any local synchronization between neighbour processors. Although the number of iterations usually increases, the simulation time has been reduced in about a 10%. An heuristic to recover the system, if any of the mentioned strategies fails has also been considered. We are working on a fourth model which will separate the convergence processes between the different processors working on the same piece of cloth.

4 Results and Conclusions

In figure 2, the execution time, in seconds (excluding the collision detection phase), of one second of simulated time and the speed-up of two example models is shown. These results has been obtained using R10000-250Mhz processors. The first model corresponds to a flag and the second to a table cloth (figure 3) (both model are under windy conditions). A real time simulation is observed for the former model using six processors. The speed-up of the second and more complex model shows a superlinear behavior with up to four processors, mainly due to the memory hierarchy (with n processors the number of internal registers and the capacity of the cache are n-times higher). These, and other preliminary results has been considered to extract several conclusions.

The use of more recent computers and a higher number of processors for models with more particles/triangles will allow real time simulations, taking into account the isoscalability shown in our preliminary results. The scenery complexity, considering interaction between several objects will be improved as the speed of the microprocessors increases.

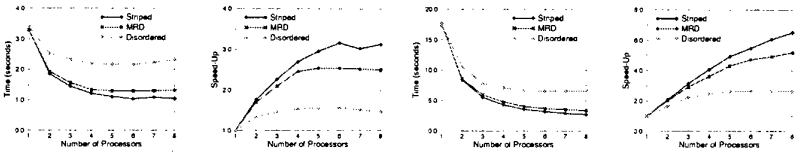


Fig. 2. Time and Speed-UP of two different simulations

References

1. Baraff, D., Witkin A.: Large Steps in Cloth Simulation. In Michael Cohen, editor, SIGGRAPH 98 Conference Proceeding, Annual Conference Series, pages 43–54. ACM SIGGRAPH, Addison Wesley, July 1998.
2. Dongarra, J., Duff, I.S., Sorensen, D.C., Van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers Software, Environments and Tools series. SIAM, 1998.
3. Volino, P., Courchesne, M., Thalmann, N.: Versatile and efficient techniques for simulating cloth and other deformable objects. Computer Graphics, 29 (Annual Conference Series):137–144, 1995.
4. Volino, P., Thalmann, N.: Collision and Self-collision detection: Efficient and Robust Solutions for Highly Deformable Objects Computer Animation and Simulation95: 55-65. Eurographics Springer-Verlag, 1995.

Fig. 3. Several frames from a simulation of a table cloth under windy conditions

Parallel Approximation to High Multiplicity Scheduling Problems via Smooth Multi-valued Quadratic Programming*

Maria Serna and Fatos Xhafa

Department of LSI
Universitat Politcnica de Catalunya
Campus Nord, C6, Jordi Girona Salgado, 1-3
08034-Barcelona, Spain
Email: {mjserna,fatos}@lsi.upc.es

Abstract. We consider the parallel approximability of two problems arising from High Multiplicity Scheduling problems, namely the *Unweighted Model with Variable Processing Requirements* and the *Weighted Model with Identical Processing Requirements.* We first show a parallel *additive* approximation procedure to a subclass of Multi-valued Quadratic Programming, called Smooth Multi-valued QP, that is defined by imposing certain restrictions on the coefficients of the instance, and then we use this procedure to obtain parallel approximation to *dense* instances of the two problems mentioned above by observing that dense instances of these problems are modeled by Smooth Multi-valued QP. The *dense* instances of the problems considered here are defined similarly as for other combinatorial problems in the literature. For such instances we can find in parallel a schedule whose completion time is at most $(1+\varepsilon)$ times the minimum schedule and it schedules at least $(1-\varepsilon)$ of jobs of any type.

1 Introduction

In High Multiplicity Scheduling Problems the jobs are partition into groups (or *types*) and in each group all the jobs are identical. The number of jobs of a certain type is called the multiplicity of that type. The goal is to find a schedule that minimizes a specified parameter such as completion time, lateness, tardiness, etc. (see, e.g., [6] for definitions and known results on different subclasses of the problem.) Multiplicity Scheduling Problems, in their general form, are NP-hard. In fact, the problem remains NP-hard even for the simple case where there is only one job of each type, there are two identical machines released at time zero, having processing capacity n (see, e.g. [4]). However, several subclasses of interest are obtained by restricting to the model where all job types have the same processing requirements. Among others, there is the subclass of *Unweighted Model with Variable Processing Requirements* and *Weighted Model with Identical Processing Requirements.* In [5] were given polynomial time algorithms to the

* This research was partially supported by ALCOM-FT (IST-99-14186) and CICYT project TIC1999-0754-C03.

problems by modeling them as Convex Separable QP. (Even strong[2] polynomial time algorithms are known [6] for the case of a single machine.) To the best of our knowledge the parallel approximability of the problems has not been considered previously. Unfortunately, the algorithm of [5] cannot be efficiently parallelized, unless P=NC, since it is based on solving convex-separable quadratic programs which are shown even non-approximable in parallel [14]. We will limit ourselves to those instances of the abovementioned problems that are *"dense."* The definition of the dense instances is done similarly as for other combinatorial optimization problems in the literature [3, 2, 9, 8, 7]. Such instances have minimum completion time $\Omega(n^2)$ and satisfy some restrictions on the weights, the released times as well as on the processing times of the jobs. For dense instances we can find in parallel a schedule of jobs into machines whose completion time is at most $(1+\varepsilon)$ times the minimum schedule and it schedules at least $(1-\varepsilon)n_j$ jobs for any group of jobs of type j, $j = 1, \ldots, n$.

We obtain the result by showing first a parallel *additive* approximation procedure to a subclass of Multi-valued Quadratic Programming, called Smooth Multi-valued QP, that is defined by imposing certain restrictions on the coefficients of the instance, and then we use this procedure to obtain parallel approximation to *dense* instances of the two problems mentioned above by observing that dense instances of these problems are modeled as Smooth Multi-valued QP.

The Smooth Quadratic Programming (Smooth QP) in 0/1 variables was first defined by Arora et al. [3] by imposing restrictions on the magnitudes of the coefficients appearing in the instances of the problem. Later on, this subclass was extended, in the same spirit, to c-Smooth QP by Arora et al. [2]. Both Smooth QP and c-Smooth QP were shown to have additive approximation procedures in polynomial time, i.e., procedures that find in polynomial time approximate solutions whose measure is within an additive error from the optimum measure [3, 2]. Interestingly, there is a close relation between *smooth* instances of QP and *dense* instances of several combinatorial optimization problems. Indeed, it was shown in [3, 2] that many combinatorial optimization problems can be casted by quadratic programs and the QPs corresponding to their dense instances are smooth or c-smooth. From this connection were obtained Polynomial Times Approximation Schemes for dense instances of several NP-hard problems, including Max CUT, Max kSAT, Linear Arrangements Problems, etc.

The approximability of Smooth QP has been also considered in the parallel setting [15] where it was proven that the scheme of [3] can be also done in parallel. It should be mentioned, however, that the Smooth QP considered in [3, 2] are in 0/1 variables. Clearly, a more vast subclass of QP is that defined in multi-valued integer variables, we call this class Smooth Multi-valued QP (Smooth MQP). We extend the result on Smooth QP in 0/1 variables to Smooth Multi-valued QP, by showing that there is a parallel additive approximation procedure to the instances of the problem. The extension to the multi-valued case is done by reducing in NC the instance of QP to an instance of LP in packing/covering

[2] A strongly polynomial time algorithm runs in polynomial time whenever the arithmetic operations can be done in polynomial time

form whose near-optimal solution can be found in NC through the algorithm of Luby and Nisan [10] and then the fractional solution is rounded to an integer one.

The paper is organized as follows. In Section 2 we formally define the problems used through the paper and briefly recall some known techniques that we will make use of. The approximation procedure of Smooth Multi-valued QP is given in Section 3 and in Section 4 we apply the approximation procedure on two problems arising from High Multiplicity Scheduling. We conclude with some open questions.

2 Preliminaries and Definitions

Let us give the definition of the Smooth Multi-valued QP.

Definition 1 (Smooth MQP). Let $A = (a_{ij})$ be an $n \times n$ matrix, $W = (w_{ij})$ an $m \times n$ matrix, b an n-vector and d an m-vector, over rationals and a a rational. Multi-valued Quadratic Programming (Multi-valued QP) is

$$\begin{array}{l} \min \sum a_{ij}x_ix_j + \sum b_ix_i + a \\ \text{s.t.} \\ \quad Wx \geq d \\ \quad 0 \leq x_i \leq c_i,\ 1 \leq i \leq n \\ \quad x_i \text{ integral},\ 1 \leq i \leq n\ . \end{array} \tag{1}$$

An instance of Multi-valued QP is called smooth if a_{ij} are $O(1)$, b_i are[3] 0 or $\Theta(n)$, a is 0 or $\Theta(n^2)$ and w_{ij} are $O(1)$. The entries of W and d are non-negative.

The definition of Smooth MQP intents to capture subclasses of the Multi-valued QP problem that represent advantages with respect to the approximability. Clearly, the imposed conditions restrict the problem, yet Smooth MQP will result strong enough to cast instances of interest for several problems.

In proving the result we will make use of the following known techniques. The first one is a standard technique for the estimation of the sum of n numbers by random sampling (see, e.g., [11]).

Lemma 2 (Sampling Lemma). *Let $\{a_i\}_{i=1}^n$ be a set of n numbers, where each a_i is $O(1)$. Let $p = \sum_{i=1}^n a_i$ be their sum. If we pick uniformly at random a subset of $s = O(\log n/\varepsilon^2)$ of a_i's and compute their sum q, then with high probability, i.e., with probability at least $1 - O(1/n)$, we have that $p - \varepsilon n \leq qn/s \leq p + \varepsilon n$.*

The second technique that we will use is the Randomized Rounding in NC by Alon and Srinivasan [1] to a subclass of linear programs. This can be seen as a parallel counterpart of the Randomized Rounding of Raghavan and Thompson [13] and Raghavan [12] that work in the sequential setting. Additionally, the

[3] Some of the b_i's as well as a may not appear in the objective function, i.e., they are equal to 0.

scheme of Alon and Srinivasan can deal with the more general case of Packing Integer Programs (PIPs, see definition below), namely the ones with multi-valued variables. Their result is expressed in terms of the Raghavan's result, which states, informally, that if v^* is the optimal fractional value of the PIP then an integral feasible solution to the PIP can be found such that its objective function value is $v' = v^*(1-\delta)$, for a small $\delta > 0$.

Alon and Srinivasan's Rounding Scheme. This Rounding Scheme applies to the class of *Packing Integer Programs* (PIP's).

Definition 3. A Packing Integer Program is to maximize $q^T x$ subject to $Mx \leq p$ where $M \in [0,1]^{m \times n}$, p is an m-vector, and q is an n-vector such that the entries of p and q are non-negative rationals, with the integrality constraint on variables $x_j \in \{0, 1, \ldots, d_j\}$; some of d_j could be also infinite.

Essentially, the technique starts from a fractional solution of the linear program and shows how to do the rounding efficiently in NC. Their result is summarized in the next theorems.

Theorem 4 [1]. *Given an instance of PIP, if the right hand sides b_i are constants bounded by $O(\log(m+n))$ than it can be approximated in NC to within a (1+o(1)) factor of the sequential bounds of [12].*

There are applications, however, in which simply an integer feasible solution of the PIP might be required. In such a case the following version of the above theorem can be applied. Note that in this case we lose the "reasonable" performance guarantee of the feasible integral solution.

Theorem 5 [1]. *For any constant $c > 1$, PIPs can be approximated in NC to within $1/c$ factor of the sequential guarantee of [12].*

Note also that in both cases the integer feasible solution is found *deterministically* in NC.

3 Approximating Smooth Multi-valued Quadratic Programming

In this section we show an NC additive approximation procedure that, given in input an instance of Smooth MQP, finds a feasible solution to the problem whose measure is within an additive error from the optimum value, i.e., $g(x) \leq g(x^*) + \varepsilon n^2$, where g is the objective function, n is the number of variables and ε a positive constant. As a corollary, when the optimal value $g(x^*)$ of the MQP instance is $\Omega(n^2)$, we obtain an $(1+\varepsilon)$-approximation for any (constant) value of ε, thus the problem has an NC Approximation Scheme. In particular, we obtain NCASs for instances of Positive Smooth MQP since for such instances $g(x^*) = \Omega(n^2)$.

We prove the following theorem.

Theorem 6. *Given an instance of Smooth MQP such that $Wx \geq d$, $0 \leq x_i \leq c_i$, $1 \leq i \leq n$, is feasible and a fixed ε, we can find in NC an integral vector x with $0 \leq x_i \leq c_i$, such that x satisfies $Wx \geq d$, and*

$$g(x_1, x_2, \ldots, x_n) \leq g(x^*) + \varepsilon n^2 \ , \tag{2}$$

where $g(x)$ is the objective function and $g(x^*)$ is its optimal value.

For ease of exposition and notation we will present the result for the Positive Smooth MQP, i.e. we suppose without loss of generality that the instance has all the coefficients non-negative and then show how to deal with the general case in Section 3.2.

3.1 Approximating Positive Smooth MQP

Theorem 7. *Given a positive instance of Smooth MQP such that $Wx \geq d$, $0 \leq x_i \leq c_i$, $1 \leq i \leq n$, is feasible and a fixed ε, we can find in NC an integral vector x with $0 \leq x_i \leq c_i$, such that x satisfies $Wx \geq d$, and*

$$g(x_1, x_2, \ldots, x_n) \leq g(x^*) + \varepsilon n^2 \ , \tag{3}$$

where $g(x)$ is the objective function and $g(x^*)$ is its optimal value.

To prove the theorem, we write the program (1) equivalently as[4]:

$$\begin{array}{l} \min c \\ \text{s.t. } x^T A x + bx \leq c \\ \quad Wx \geq d \\ \quad 0 \leq x_i \leq c_i,\ 1 \leq i \leq n \\ \quad x_i \text{ integral},\ 1 \leq i \leq n \ . \end{array} \tag{4}$$

Notice that, by using a binary search, for Theorem 7 it suffices to prove the following theorem.

Theorem 8. *Suppose there is an integral solution to the following Positive Multivalued Quadratic System*

$$\begin{array}{l} x^T A x + bx \leq c \\ Wx \geq d \\ 0 \leq x_i \leq c_i,\ 1 \leq i \leq n \ . \end{array} \tag{5}$$

Then, for any fixed $\varepsilon > 0$, we can find in NC an integer vector x with $0 \leq x_i \leq c_i$, $1 \leq i \leq n$, that satisfies $Wx \geq d$ and such that

$$x^T A x + bx \leq c + \varepsilon n^2 \ . \tag{6}$$

The main steps of the proof are the followings:

- the Smooth MQP is reduced in NC to an appropriate LP program in packing form to which a fractional solution is found via Luby-Nisan's [10] algorithm.
- the fractional solution is rounded to an integer one through the technique of Alon and Srinivasan [1].

[4] The constant a in the objective function is unimportant.

Reducing Smooth MQP to LP

Let $\overline{x}$ be a feasible solution to (5), as supposed, and let us write $\overline{r} = \overline{x}A+b$. Notice that $r_i = \Theta(n)$ due to the magnitudes of the coefficients. Since $x^TAx + bx = (x^TA + b)x$, we can express the quadratic program (5) as a linear program

$$\begin{array}{l} x^TA + b \leq \overline{r} \\ \overline{r}x \leq c \\ Wx \geq d \\ 0 \leq x_i \leq c_i, \quad 1 \leq i \leq n \ , \end{array} \tag{7}$$

(LP1)

for which $\overline{x}$ is also a feasible solution. The following observation is immediate.

Proposition 9. *If x is a feasible solution to (LP1) then x is also feasible to (5).*

Clearly, any coefficient in (LP1) is non-negative. However, (LP1) is not in the packing/covering form because in it we have both types of restrictions. To overcome this, we modify (LP1) appropriately by introducing variables $z_i = c_i - x_i$. Thus, (LP1) is written as

$$\begin{array}{rl} x^TA + b \leq \overline{r} & \\ \overline{r}x \leq c & \\ \sum_{j=1}^{n} w_{kj}z_j \leq \sum_{j=1}^{n} w_{kj}c_j - d_k, & 1 \leq k \leq m \\ x_i + z_i = c_i, & 1 \leq i \leq n \\ 0 \leq x_i, z_i, & 1 \leq i \leq n \ . \end{array}$$

(LP2)

We can suppose, without loss of generality, that $\forall k$, $1 \leq k \leq m$, $\sum_{j=1}^{n} w_{kj}c_j - d_k \geq 0$ because otherwise the system $\{Wx \geq d,\ x_i \leq c_i,\ 1 \leq i \leq n\}$ would not be feasible. So, the above program (LP2) is still positive. The relation between (LP1) and (LP2) is given as follows.

Proposition 10. *(a) if x is a feasible solution to (LP1) then (x,z), where $z_i = c_i - x_i$, is also feasible to (LP2); (b) if (x,z) is a feasible solution to (LP2) then x is a feasible solution to (LP1).*

Finally, to transform the conditions $x_i + z_i = c_i$ into $x_i + z_i \leq c_i$, we add to (LP2) an adequate objective function resulting in the following program:

$$\begin{array}{lll} \max & \sum_{i=1}^{n}(x_i + z_i) & \\ \text{s.t.} & x^TA + b \leq \overline{r} & (8) \\ & \overline{r}x \leq c & (9) \\ & \sum_{j=1}^{n} w_{kj}z_j \leq \sum_{j=1}^{n} w_{kj}c_j - d_k, & 1 \leq k \leq m \\ & x_i + z_i \leq c_i, & 1 \leq i \leq n \\ & 0 \leq x_i, z_i, & 1 \leq i \leq n \ . \end{array}$$

(LP3)

Notice that (LP3) is a Positive Packing Program. The programs (LP2) and (LP3) are related as follows:

Proposition 11. *(a) if (x, z) is a feasible solution to (LP2) then it is also feasible to (LP3); (b) if (x, z) is an $(1-\varepsilon)$-***optimal** *solution to (LP3) then we can construct (x', z') in NC such that it is* **feasible** *to $\{\sum_{j=1}^{n} w_{kj} z_j \leq \sum_{j=1}^{n} w_{kj} c_j - d_k,\ 1 \leq k \leq m,\ x_i + z_i \leq c_i,\ 1 \leq i \leq n\}$ and*

$$x'^T A + b \leq \overline{r} + K_1 \varepsilon$$
$$\overline{r} x' \leq c + K_2 \varepsilon n$$

where K_1 and K_2 are two constants computable in NC from the instance.

Proof. b) A near-optimal solution (x, z) to (LP3) can be found via the Luby and Nisan's algorithm. Such solution will have cost at least $(1-\varepsilon)\sum_{i=1}^{n} c_i$ (supposing that (LP2) is feasible). Now, we define $z' = z$ and x', by taking $x'_i = c_i - z_i$. Since, due to near optimality of solution (x, z),

$$\sum_{i=1}^{n} (x_i + z_i) \geq (1-\varepsilon) \sum_{i=1}^{n} c_i$$

we will have that $x_i + z_i \approx (1-\varepsilon) c_i$ therefore, intuitively, the values of x'_i will not increase too much. More precisely, let, for any i, $x'_i = x_i + \varepsilon_i$, where $\varepsilon_i = c_i - x_i - z_i$ and also let $e = (\varepsilon_1, \ldots, \varepsilon_n)$. Notice that

$$\sum_{i}^{n} \varepsilon_i = \sum_{i}^{n} (c_i - x_i - z_i) = \sum_{i=1}^{n} c_i - \sum_{i}^{n} (x_i + z_i) \leq \sum_{i=1}^{n} c_i - (1-\varepsilon) \sum_{i=1}^{n} c_i = \varepsilon \sum_{i=1}^{n} c_i \ . \tag{10}$$

Clearly, the system $\{\sum_{j=1}^{n} w_{kj} z_j \leq \sum_{j=1}^{n} w_{kj} c_j - d_k,\ 1 \leq k \leq m,\ x_i \leq c_i,\ 1 \leq i \leq n\}$ will be satisfied by (x', z') but the rest of constraints (8) and (9) might be violated by x'. However, because of the magnitudes of $a_{ij} = O(1)$ and $\overline{r}_i = \Theta(n)$, for x' we have that

$$x'^T A + b = (x^T + e) A + b \leq (x^T A + b) + eA \leq \overline{r} + K_1 \varepsilon \tag{11}$$
$$\overline{r} x' = \overline{r}(x + e) \leq c + K_2 \varepsilon n \tag{12}$$

where K_1 and K_2 are two constants defined as follows: $K_1 = (\sum_{i=1}^{n} c_i) \cdot \max_{i,j} a_{ij}$ and $K_2 = (\sum_{i=1}^{n} c_i) \cdot \max_i \alpha_i$ where α_i are the upper bounds on $\overline{r}_i$, $\overline{r}_i \leq \alpha_i \cdot n$. □

Rounding of the Fractional Solution

Having the fractional solution (x', z'), we apply the Randomized Rounding of Alon and Srinivasan to $\{\sum_{j=1}^{n} w_{kj} z_j \leq \sum_{j=1}^{n} w_{kj} c_j - d_k,\ 1 \leq k \leq m\}$. Rounding the feasible fractional solution z' gives an integral solution u, that satisfies

$\sum_{j=1}^{n} w_{kj}u_j \leq \sum_{j=1}^{n} w_{kj}c_j - d_k$, $1 \leq k \leq m$. Letting y such that $y_i = c_i - u_i$, we have

$$\begin{aligned} Wy &\geq d \\ y_i \leq c_i, \quad 1 \leq i \leq n \ . \end{aligned}$$

Furthermore, for y we will have:

$$\begin{aligned} y^T A_i + b_i &\leq (\overline{r}_i + K_1\varepsilon) + O(\sqrt{n \log n}) \\ \overline{r}y &\leq (c + K_2\varepsilon n) + O(n) \cdot O(\sqrt{n \log n}) \ . \end{aligned} \tag{13}$$

Consequently,

$$\begin{aligned} y^T Ay + by &= (y^T A + b)y \leq \left((\overline{r} + K_1\varepsilon) + O(\sqrt{n \log n})\right) y \\ &\leq \overline{r}y + K_1\varepsilon \mathbf{1} \cdot y + O(\sqrt{n \log n})\mathbf{1} \cdot y \\ &\leq (c + K_2\varepsilon n) + O(n) \cdot O(\sqrt{n \log n}) + K_1\varepsilon \mathbf{1} \cdot y + O(\sqrt{n \log n})\mathbf{1} \cdot y \\ &\leq c + (K_1 + K_2 + 2)\varepsilon n^2 \ , \end{aligned} \tag{14}$$

where we have denoted by $\mathbf{1}$ the vector all whose entries are equal to 1. Thus, if we find the fractional solution with $\varepsilon' = \varepsilon/(K_1 + K_2 + 2)$ we will have, from above, $y^T Ay + by \leq c + \varepsilon n^2$, as desired.

But, we can write (7) only if we knew the values $\overline{r}_i$, i.e., the vector $\overline{r}$. Instead, it is shown in [3] that using estimates r_i for them such that $|\overline{r}_i - r_i| < \varepsilon n$ then (13) and (14) still hold. These estimates are found similarly as in [3, 2], through the Sampling Lemma, and we omit the details. □

Corollary 12. *If the instance of Smooth MQP has optimal value $g(x^*) = \Omega(n^2)$, i.e., $g(x^*) \geq \delta n^2$, for some $\delta > 0$, then Theorem 6 implies an $(1+\varepsilon)$-approximation to such instances, i.e. an NC Approximation Scheme.*

Indeed, in this case the solution x satisfies $g(x) \leq (1 + \varepsilon/\delta)g(x^*)$.

3.2 The general case: Smooth MQP

A close observation of the proof of Theorem 7 shows that the assumption on the coefficients a_{ij}, b_i and a of the MQP objective function to be non-negative can be removed[5]. Indeed, this assumption is used only in the reduction step to obtain a positive linear program (LP3) in order to enable the Luby and Nisan's algorithm. More precisely, the assumption assures that the linear restrictions

$$\begin{aligned} x^T A + b &\leq r^{\#} \\ r^{\#} x &\leq c \end{aligned} \tag{15}$$

have positive coefficients. This last fact can be assured without the positiveness assumption as follows:

[5] To be precise, the notation on the magnitudes of the coefficients given in the definition of Smooth MQP becomes now $a_{ij} = O(1)$, $b_i = O(n)$ and $c = O(n^2)$; also, $r_i^{\#} = O(n)$ (the notation used in [3]).

a) if some $a_{ij} < 0$ then substitute the term $a_{ij}x_i$ by $-a_{ij}(z_i - c_i)$.
b) if some $r_i^{\#} < 0$ then substitute the term $r_i^{\#}x_i$ by $-r_i^{\#}(z_i - c_i)$.

Notice that by applying a) the left-hand sides of restrictions (15) become non-negative and therefore the right-hand sides so do (otherwise the program would be infeasible). Doing these (possible) variable changes we obtain the positive linear program (LP3). The variable change effects the right-hand sides of $\{x^T A + b \leq r^{\#}, r^{\#}x \leq c\}$ but yet it doesn't effect Proposition 11 due to the definition of (x', z') and the magnitudes of the coefficients. On the other hand, removing the positiveness condition on a_{ij}, b_i and a doesn't effect the rounding scheme of Alon and Srinivasan since it applies to the program $Wz \leq W \cdot \mathbf{1} - d$. From this observation we can restate Theorem 7 without the positiveness condition on the coefficients of the objective function of the MQP instance thus yielding Theorem 6.

4 Applications

In this section we show that the additive approximation on Smooth MQP can be applied to a couple of problems arising from High Multiplicity Scheduling Problems, namely the *Weighted Model with Identical Processing Requirements* and *Unweighted Model with Variable Processing Requirements*. We first give the description of the general model of High Multiplicity Scheduling.

General Model

The Multiplicity Scheduling Problem, in its general form, is stated as follows [5]. There are n types of jobs, $J_1, J_2, \ldots, J_n$ and there are n_j identical jobs of type J_j. There are m parallel machines $M_1, M_2, \ldots, M_m$, to process the jobs. Machine M_i is released at time r_i and it can process at most c_i jobs. Each job should be processed in its entirety in one of the m machines. All the jobs are available for processing at time 0. The processing requirement of job of type j on machine i is p_{ij} time units. There is a non-negative weight w_{ij} associated to job j when processed by machine i. The objective is to schedule the jobs on m machines so that the total weighted completion time is minimized.

4.1 Weighted Model with Identical Processing Requirements

In this case we have identical processing requirements (the p_{ij} are written as $p_{ij} = 1/s_i$) and general weights w_{ij}. For any machine i, $i = 1, \ldots, m$, let σ_i be a permutation of $\{1, \ldots, n\}$ which arranges n types of jobs in a non-increasing order of their weights. Let also the variable x_{ij} denote the number of jobs of type $\sigma_i(j)$ processed in machine i. The cost of assigning a job of type $\sigma_i(j)$ to the kth place ($k = 1, \ldots, c_i$) is $(r_i + k(1/s_i))w_{i,\sigma_i(j)}$. The integer programming [5]

formulation of the problem follows.

$$\min \sum_{i=1}^{m} \sum_{j=1}^{n} r_i w_{i\sigma_i(j)} x_{ij} + \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{s_i} w_{i\sigma(j)} \sum_{k=1}^{x_{ij}} \left(\sum_{l=1}^{j-1} x_{il} + k \right)$$

s.t.

$$\sum_{j=1}^{n} x_{ij} = c_i, \ (i = 1, \ldots, m) \tag{16}$$

$$\sum_{\{(i,k), i=1,\ldots m, \sigma_i(k)=j\}}^{m} x_{ij} = n_j, \ (j = 1, \ldots, n)$$

$$0 \leq x_{ij} \leq c_i, \text{integral}, \ (i = 1, \ldots, m; \ j = 1, \ldots, n) \ .$$

(WIP)

Notice that the restriction (16) is written with equality since we can always introduce an additional type of job $(n+1)$ with weight $w_{i,n+1} = 0$. In [5] this program was written as a convex separable quadratic program and then was solved through known polynomial time algorithms (e.g. Minoux, 1986).

In order to apply the additive approximation procedure, we will limit ourselves to *dense* instances of the problem as specified below.

Dense Instances. An instance of the High Multiplicity Scheduling Problem (Weighted Model with Identical Processing Requirements) is dense if it satisfies the following restrictions:

(a) The weights w_{ij}, the released times r_i and the processing times p_{ij} are bounded by constants);
(b) The completion time of the minimum schedule is $\Omega(n^2)$.

Further, we consider a relaxation of the integer program (WIP) by relaxing the equality restrictions into covering type restrictions as follows.

$$\min \sum_{i=1}^{m} \sum_{j=1}^{n} r_i w_{i\sigma_i(j)} x_{ij} + \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{s_i} w_{i\sigma(j)} \left(x_{ij} \left(x_{i1} + \cdots + x_{i,j-1} + \frac{1}{2} x_{ij} \right) \right)$$

s.t.

$$\sum_{j=1}^{n} x_{ij} \geq (1-\varepsilon) c_i, \ (i = 1, \ldots, m)$$

$$\sum_{\{(i,k), i=1,\ldots m, \sigma_i(k)=j\}}^{m} x_{ij} \geq (1-\varepsilon) n_j, \ (j = 1, \ldots, n)$$

$$0 \leq x_{ij} \leq c_i, \ (i = 1, \ldots, m; \ j = 1, \ldots, n) \ .$$

(WQP)

It is straightforward to observe the following relation between dense instances of our scheduling problem and the smooth instances of Multi-valued QP.

Proposition 13. *If the instance of scheduling is dense then the corresponding (WQP) instance is a Smooth Multi-valued QP instance.*

Hence by applying Theorem 6 and Corollary 12 to such instances we obtain the following:

Theorem 14. *Given a dense instance of Multiplicity Scheduling (Weighted Models with Identical Processing Requirements) of n jobs and m machines there is an NC algorithm that finds a schedule whose completion time is at most $(1+\varepsilon)$ times the minimum schedule and it schedules at least $(1-\varepsilon)n_j$ jobs for any group of jobs of type j, $j = 1, \ldots, n$.*

4.2 Approximating Unweighted Model with Variable Processing Requirements

This problem is obtained from the general model by letting $w_{ij} = v_i$ for all types of job j and machine i. The processing time of each one of the n_j jobs of type j on the ith machine is p_{ij}.

Let us suppose that, for any machine i, $i = 1, \ldots, m$, the processing times are ordered in non-increasing order given by the permutation σ_i. Let also the variable x_{ij} denote the number of jobs of type $\sigma_i(j)$ processed in machine i. With this notations, we have the following integer programming formulation of the problem [5].

$$\min \sum_{i=1}^{m} \sum_{j=1}^{n} r_i v_i x_{ij} + \sum_{i=1}^{m} \sum_{j=1}^{n} v_i p_{i\sigma(j)} \sum_{k=1}^{x_{ij}} \left(\sum_{l=1}^{j-1} x_{il} + k \right)$$

s.t.

$$\sum_{j=1}^{n} x_{ij} = c_i, \ (i = 1, \ldots, m)$$

$$\sum_{\{(i,k), 1 \le i \le m, \sigma_i(j) = k\}} x_{ij} = n_j, \ (j = 1, \ldots, n)$$

$$0 \le x_{ij}, \text{integral}, \ (i = 1, \ldots, m; \ j = 1, \ldots, n) \qquad \text{(IP)}$$

Now we apply the approximation scheme in the same way as in the case of the Weighted Model with Identical Processing Requirements and obtain the following theorem.

Theorem 15. *Given a dense instance of Multiplicity Scheduling (Unweighted Model with Variable Processing Requirements) of n jobs and m machines there is an NC algorithm that finds a schedule whose completion time is at most $(1+\varepsilon)$ times the minimum schedule and it schedules at least $(1-\varepsilon)n_j$ jobs for any group of jobs of type j, $j = 1, \ldots, n$.*

Open Questions

Our approximation procedure applies to a restricted class of instances of the scheduling problems. It would be interesting to extend the result to other instances of the problem without the denseness condition. Also, applying our procedure to other problems modeled by Multi-valued QP would be of interest.

References

1. Alon, N., Srinivasan, A.: Improved Parallel Approximation of a Class of Integer Programming Problems. *Algorit.*, **17** (1997) 449–462
2. Arora, S., Frieze, A., Kaplan, H.: A New Rounding Procedure for the Assignment Problem with Applications to Dense Graph Arrangement Problems. In *Proc. of the FOCS'96*, (1996) 21–30
3. Arora, S., Karger, D., Karpinski, M.: Polynomial Time Approximation Schemes for Dense Instances of NP-hard Problems. *J. Comput. System Sci.*, **58**(1) (1999) 193–210
4. Garey, M.R., Johnson, D.S.: *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co. (1979)
5. Granot, F., Skorin-Kapov, J., Tamir, A.: Using Quadratic Programming to Solve High Multiplicity Scheduling Problems on Parallel Machines. *Algorit.*, **17** (1997) 100–110
6. Hochbaum, D.S., Shamir, R.: Strongly Polynomial Algorithms for the High Multiplicity Scheduling Problem. *Op. Res.*, **39**(4) (1991) 648–653
7. Karpinski, M., Wirtgen, J., Zelikovsky, A.: An Approximation Algorithm for the Bandwidth Problem on Dense Graphs. Tech. Rep. **TR97-017**, ECCC, (1997)
8. Karpinski, M., Wirtgen, J., Zelikovsky, A.: Polynomial Times Approximation Schemes for Some Dense Instances of NP-hard Problems. Tech. Rep. **TR97-024**, ECCC, (1997)
9. Karpinski, M., Zelikovsky, A.: Approximating Dense Cases of Covering Problems. Network Design: Connectivity and Facilities Location (Princeton, NJ, 1997) Amer. Math. Soc., (1998) 169–178
10. Luby, M., Nisan, N.: A Parallel Approximation Algorithm for Positive Linear Programming. In *Proc. of 25th ACM Symp. on Theory of Comp.*, (1993) 448–457
11. Motwani, R., Raghavan, P.: *Randomized Algorithms.* Cambridge Univ. Press (1995)
12. Raghavan, P.: Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs. *J. of Comp. and Syst. Sci.*, **37** (1988) 130–143
13. Raghavan, P., Thompson, C.: Randomized Rounding: a Technique for Provably Good Algorithms and Algorithmic Proofs. *Combinat.*, **7** (1987) 365–374
14. Serna, M.: Approximating Linear Programming is Logspace Complete for P. *Inf. Proc. Lett.*, **37** (1991) 233–236
15. Serna, M., Xhafa, F.: The Parallel Approximability of a Subclass of Quadratic Programming. In *Proc. of Int. Conf. on Parallel and Distributed Systems, ICPADS'97*, IEEE, (1997) 474–482 *To appear in Theoret. Comp. Sci.*

This article was processed using the LaTeX macro package with LLNCS style

High Level Parallelization of a 3D Electromagnetic Simulation Code With Irregular Communication Patterns

Emmanuel Cagniot[1]*, Thomas Brandes[2], Jean-Luc Dekeyser[1], Francis Piriou[3], Pierre Boulet[1] and Stéphance Clénet[3]

[1] Laboratoire d'Informatique Fondamentale de Lille (LIFL)
U.S.T.L. Cité Scientifique, F-59655 Villeneuve d'Ascq Cedex, France
[2] Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
[3] Laboratoire d'Electrotechnique et d'Electronique de Puissance (L2EP)
U.S.T.L. Cité Scientifique, F-59655 Villeneuve d'Ascq Cedex, France

Abstract. 3D simulation in electrical engineering is based on recent research work (Whitney's elements, auto-gauged formulations, discretization of the source terms) and it results in complex and irregular codes. Generally, explicit message passing is used to parallelize this kind of applications requiring tedious and error prone low level coding of complex communication schedules to deal with irregularity. In this paper, we focus on a high level approach using the data-parallel language High Performance Fortran. It allows both an easier maintenance and a higher software productivity for electrical engineers. Though HPF was initially conceived for regular applications, it can be successfully used for irregular applications when using an unstructured communication library that deals with indirect data accesses.

Topics: cellular automata and physics.

1 Introduction

[1] Electrical engineering consists of designing electrical devices like iron core coils (example 1) or permanent magnet machines (example 2) (see Fig. 1). As prototypes can be expensive, numerical simulation is a good solution to reduce development costs. It allows to predict device performance from physical design information. Accurate simulations require 3D models, inducing high storage capacity and CPU power needs. As computation times can be very important, parallel computers are well suited for these models.

3D Electromagnetic problem modeling is based on Maxwell's equations. Generally, the resolution of these partial differential equations requires numerical methods. They transform the differential equations into an algebraic

* Corresponding author, e-mail: `cagniot@lifl.fr`, Tel: +33-0320-434730, Fax: +33-0320-436566

[1] Candidate to the best student paper award

equation system whose solution gives an approximation of the exact solution. The space discretization and the time discretization can be done respectively by the finite element method (FEM) and by the finite difference method (FDM). The finite elements used are Whitney's elements (nodes, edges, facets and volumes) [1]: they allow to keep the properties of continuity of the field at the discrete level. In function of the studied problem, the equation system can be linear or non-linear.

As for many other engineering applications, FEM codes use irregular data structures such as sparse matrices where data access is done via indirect addressing. Therefore, finding data distributions that provide both high data locality and good load balancing is difficult. Generally, parallel versions of these codes use explicit message passing.

In this paper, we focus on a data-parallel approach with High Performance Fortran (HPF). Three reasons can justify this choice. First, a high level programming language is more convenient than explicit message passing for electrical engineers. It allows both an easier maintenance and a higher software productivity. Second, libraries for optimizing unstructured communications in codes with indirect addressing exist [4]. The basic idea is that these codes reuse several times the same communication patterns. Therefore, it is possible to compute these patterns one time and to reuse them if possible. Third, the programmer can further optimize its code by mixing special manual data placements and simple HPF distributions to provide both high data locality and good load balancing.

Section 2 presents the main features of our electromagnetic code. Section 3 presents the HPF parallelization of the magnetostatic part of this code. Section 4 presents the unstructured communication library we used to efficiently parallelize the preconditioned conjugate gradient method. Section 5 presents the results we obtained on a SGI Origin and an IBM SP2. Conclusion and future works are given in Section 6.

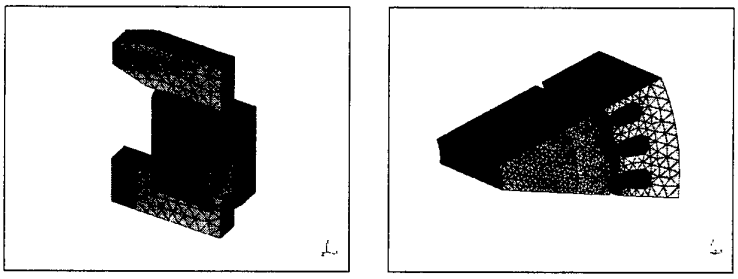


Fig. 1. An iron core coil and a permanent magnet machine.

2 The Code

The L2EP at the University of Lille has developed a 3D FORTRAN 77 code for modeling magnetostatic (time-independent) and magnetodynamic (time-dependent) problems [5]. The magnetostatic part uses formulations in terms of scalar or vector potentials where the unknowns of the problem are respectively the nodes and the edges of the grid. The magnetodynamic part uses hybrid formulations that mix scalar and vector potentials.

The great particularity of this code is the computation of the source terms using the *tree* technique. Generally, in the electromagnetic domain, gauges are required to obtain an unique solution. They can be added into the partial differential equations but this solution involves additional computations during the discretization step. Another solution results from a recent research work [7]. A formulation is said compatible when all its entities have been discretized onto Whitney's elements. This work has shown that problems are auto-gauged when both a compatible formulation and an iterative solver are used. Therefore, to obtain a compatible formulation, source terms must be discretized using the *tree* technique, actually a graph algorithm.

As the meshing tool only produces nodes and elements, the edges and the facets must be explicitly computed. The time discretization is done with Euler's implicit algorithm. The FEM discretization of a non-linear problem results in an iterative loop of resolutions of linear equation systems. Two non-linear methods are used: Newton-Raphson's method and the fixed-point method. Their utilization is formulation-dependent. The linear equation systems are either symmetric positive definite or symmetric semi-positive definite. Therefore, the preconditioned conjugate gradient method (PCG) is used. The preconditioner results from an incomplete factorization of Crout.

The overall structure of this code is as follows:

```
1. define the media, the inductors, the magnets, the boundary
   conditions, etc.
2. read the input file and compute the edges and the facets.
3. compute the source terms.
4. check the boundary conditions and number the unknowns.
5. time loop:
   5.1. create the Compressed Spare Row (CSR) representation of the
        equation system.
   5.2. non-linear loop:
        5.2.1. assembly loop:
                 - compute and store the contribution of all the
                   elements in the equation system.
        5.2.2. compute the preconditioning matrix.
        5.2.3. solving loop:
                 - iterate preconditioned conjugate gradient.
```

This structure shows that computation times can be very important when we use time-dependent formulations in the non-linear case.

The matrices for the equation systems are represented by the Compressed Sparse Row (CSR) format as shown in Fig. 2. As they are symmetric, the upper triangular part is not explicitly availabe to save memory.

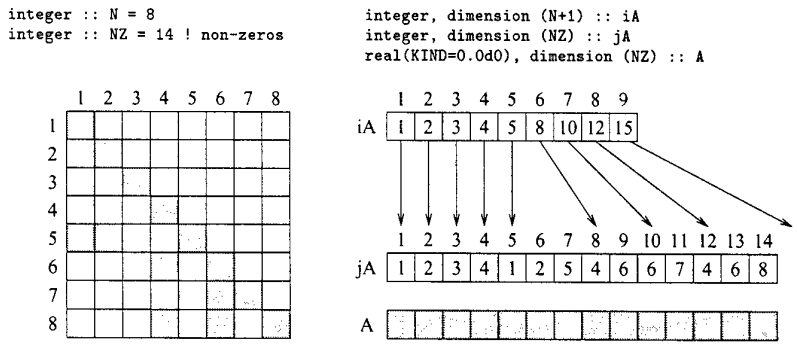


Fig. 2. Compress sparse row format of a matrix.

3 HPF Parallelization of the Magnetostatic Part

For software engineering reasons, the magnetostatic part of the code has been ported to Fortran 90 and it has been optimized. The Fortran 90 version makes extensive use of modules (one is devoted to the data structures), derived data types and array operations. It consists of about 9000 lines of code. This new version has been converted to HPF considering only the parallelization of the assembling and the solver that take about 80% of the whole execution time in the linear case. All other parts of the code remained serial.

In a first step, all the data structures including the whole equation system have been replicated on all the processors. Each of them has to perform the same computations as the others until the CSR structure of the equation system is created. Some small code changes were necessary to reduce the number of broadcasts implied by the serial I/O operations.

The assembling loop is a parallel loop over all the elements, each element contributing some entries to the equation system (see Fig. 3). Though one unknown can belong to more than one element, and so two elements might add their contribution at the same position, the addition is assumed to be an associative and commutative operation. The order in which the loop iterations are executed does not change the final result (except for possible round-off errors). Therefore it is safe to use the `INDEPENDENT` directive of HPF together with the `REDUCTION` clause for the arrays containing the values of `A` (matrix) and `B` (right hand side). A one-dimensional block-distributed template, whose

size is given by the number of elements, has been used to specify the work distribution of this loop via the `ON HOME` clause.

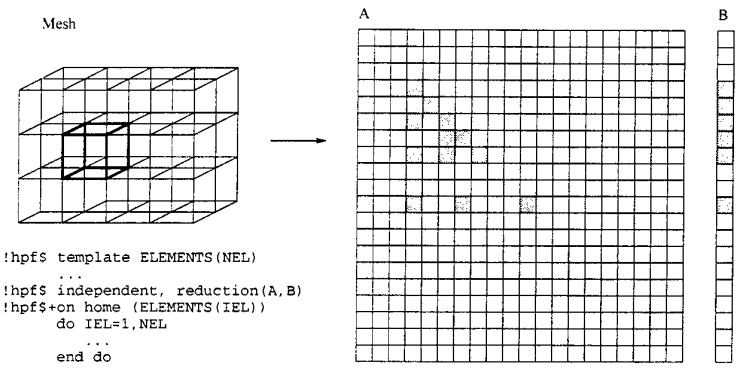


Fig. 3. Assembling the contributions for the equation system.

In a second step, the equation system and the preconditioning matrix have been general block distributed in such a way that all the information of one row $A(i,:)$ resides on the same processor as vector element $z(i)$. By the `RESIDENT` directive, the HPF compiler gets the information that avoids unnecessary checks and synchronizations for accesses to these matrices.

The algorithm for the preconditioned conjugate gradient method is dominated by the matrix-vector multiplications and by the forward/backward substitutions required during the preconditioning steps.

Due to the dependences in the computations, the incomplete factorization of Crout before the CG iterations and the forward/backward substitution per iteration are not parallel at all. The factorization has been replaced with an incomplete block factorization of Crout that takes only the local blocks into account forgetting the coupling elements. By this way, the factorization and the forward/backward substitution do not require any communication. As the corresponding preconditioning matrix becomes less accurate, the number of iterations increases with the number of blocks (processors). But the overhead of more iterations is less than the extra work and communication needed otherwise. HPF provides `LOCAL` extrinsic routines where every processor sees only the local part of the data. This concept is well suited to restrict the factorization and the forward/backward substitution to the local blocks where local dependences in one block are respected and global dependences between the blocks are ignored.

The matrix-vector multiplication uses halos [3] (see next section) that provide an image of the non-local part of a vector on each processor, and the communication schedule needed for the related updating operations. The

computation time for the halo and its communication schedule is amortized over the number of iterations.

For running our HPF code on parallel machines we used the ADAPTOR HPF compilation system of GMD [2]. Only ADAPTOR supported the features needed for the application (`ON` clause, general block distributions, `RESIDENT` directive, `REDUCTION` directive, `LOCAL` routines, halos and reuse of communication schedules). By means of a source-to-source transformation, ADAPTOR translates the data parallel HPF program to an equivalent SPMD program (single program, multiple data) in Fortran where this Fortran program is compiled with a native Fortran compiler. The generated SPMD program contains a lot of calls to the ADAPTOR specific HPF runtime system, called DALIB (distributed array library) that implements also the functionality needed for halos (see Figure 4).

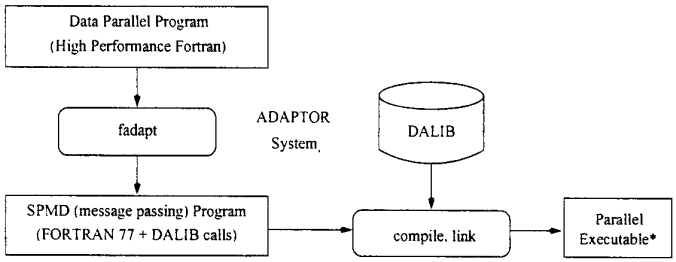


Fig. 4. Schematic view of ADAPTOR HPF compilation.

4 Unstructured Communication Library Approach

The ADAPTOR HPF compilation system provides a library that supports shadow edges (ghost points) for unstructured applications using indirect addressing, also called halos [4]. A halo provides additionally allocated local memory to keep on one processor also non-local values of the data that is accessed by the processor and a communication schedule that reflects the communication pattern between the processors to update these non-local copies. As the size of the halo and the communication schedule depend on the values of the indirection array, they can only be computed at runtime.

The use of halos (overlapping, shadow points) is common manual practice in message passing programs for the parallelization of unstructured scientific applications. But the calculation of halo sizes and communication schedules is tedious and error prone and requires a lot of additional coding. Up to now, commercial HPF compilers do not support halos. The idea of halos has already been followed within the HPF+ project and implemented in the

Vienna Fortran Compiler [3] where the use of halos is supported by additional language features instead of a library.

Fig. 5 shows the use of halos for the matrix-vector multiplication $B = A \times X$. The vectors X and B are block distributed among the available processors. The matrix A is distributed in such a way that one row $A(i,:)$ is owned by the same processor that owns $B(i)$. For the matrix-vector multiplication, every processor needs all elements $X(j)$ for the non-zero entries $A(i,j)$ owned by it. Though most of these values might be available, there remain some non-local accesses. The halo will contain these non-local values after an update operation using the corresponding halo schedule.

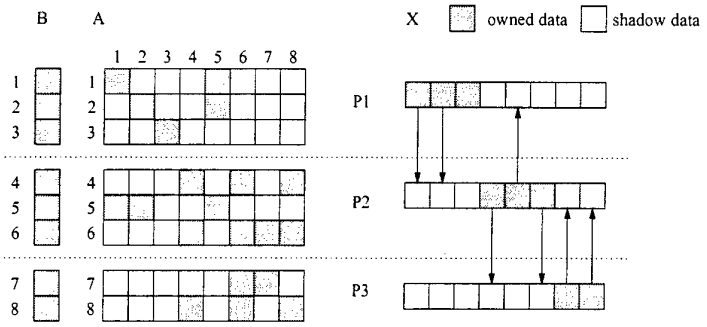


Fig. 5. Distribution of matrix with halo nodes for the vector.

As mentioned before, the upper triangular of `A` part is not explicitly available. For $j > i$ the elements $A(i,j)$ must be accessed via $A(j,i)$. To avoid the communication of matrix elements the values of $A(i,j) * X(j)$ are not computed by the owner of $B(i)$ but by the owner of $X(j)$ as here $A(j,i)$ is local. Therefore we have an additional communication step to reduce the non-local contributions of B. But for this unstructured reduction we can use the same halo structure for the vector B.

For the calculation of the halo, we had to provide the halo array, which is the indirectly addressed array, and the halo indexes that are used for the indirect addressing in the distributed dimension. In our case, the halo arrays are the vectors X and B, and the halo indexes are given by the integer array `jA` containg the column indices used in the CSR format. Beside the insertion of some subroutine calls for the HPF halo library, we had only to insert some HPF directives for the parallelization of the loop implementing the matrix-vector multiplication.

The calculation of the halo structure is rather expensive. But we can use the same halo structure for the update of the non-local copies of vector X and for the reduction of the non-local contributions of vector B. Furthermore,

this halo can be reused for all iterations of the iteration loop in the solver as the structure of the matrix and therefore the halo indexes do not change.

5 Results

Table 1 shows the characteristics of the problems of Fig. 1 in the case of a vector potential formulation. These results have been obtained for the test cases of Table 1 on a SGI Origin and on an IBM SP2 in the linear case (in the magnetodynamic case, each column would represent results associated with one time increment). Their quality has been measured by two ways: graphically with the dumped files and numerically with the computed magnetic energies. All the computed solutions are the same.

	example 1	example 2
nodes	8059	25730
edges	51356	169942
facets	84620	284669
elements	41322	140456
unknowns	49480	162939
non-zero entries	412255	1382596

Table 1. Test cases of the code

Table 2 presents results for example 1 on the SGI Origin with up to four processors. Both, the assembling and the solver scale well for a small number of processors.

	NP = 1	NP = 2	NP = 3	NP = 4
assembling	12.87 s	6.54 s	4.63 s	3.61 s
solver	30.47 s	18.89 s	9.74 s	8.80 s
iterations	225	260	213	237

Table 2. Results for example 1, SGI Origin

Table 3 presents results for example 1 on the IBM SP2 with up to 16 processors. In the assembling loop the computational work for one element is so high that the parallelization still gives good speed-ups for more processors even if the reduction overhead increases with the number of processors. The scalability of the solver is limited as the data distribution has not been optimized for data locality yet. On the other hand, the number of solver iterations does not increase dramatically with the number of processors and the higher inaccuracy.

	NP = 1	NP = 2	NP = 4	NP = 8	NP = 16
assembling	32.70 s	16.77 s	8.98 s	5.27 s	3.47 s
solver	48.45 s	36.99 s	23.51 s	14.45 s	10.59 s
iterations	224	247	257	247	258

Table 3. Results for example 1, IBM SP2

Comparison between Table 2 and Table 3 shows that for the two parallel machines the numbers of iterations are different for a same number of processors. This can be explained by the application of different aggressive optimizations of the native Fortran compilers (optimization level 3) that may generate (even slightly) different results.

	NP = 1	NP = 2	NP = 3	NP = 4
assembling	43.77 s	22.71 s	16.03 s	12.95 s
solver	625.06 s	160.84 s	116.63 s	93.25 s
iterations	1174	464	509	505
time/iter	532 ms	347 ms	229 ms	165 ms

Table 4. Results for example 2, SGI Origin

Table 4 presents results for example 2 on the SGI Origin. The assembling and one iteration of the solver scale well. Regarding the number of solver iterations, the results are surprising. For its explanation we must interest ourselves to the numbering of the unknowns which plays an important role in the conditioning of the equation system. In our case, the meshing tool sorts its elements by volumes, every volume corresponding to a magnetic permeability (air, iron, etc.). To avoid large jumps of coefficients in the matrix, the unknowns are numbered by scanning the list of elements. Therefore, to every volume of the grid corresponds a homogeneous block of the matrix. The conditioning of this matrix is directly linked to the uniformity of the magnetic permeability of these different blocks. When this uniformity is poor, block preconditioners resulting from domain decomposition methods can be used. By preconditioning each block independently, they allow to bypass the problem. In our case, the incomplete block factorization of Crout has led to the same result. For its verification we have re-sorted the elements of the grid by merging volumes with the same magnetic permeability. As a result, we have divided the number of PCG iterations by two in the Fortran 90 program.

6 Conclusions and Future Work

This HPF version achieves acceptable speed-ups for smaller number of processors. According to the few number of HPF directive added in the Fortran 90

code, it is a very cheap solution that allows both an easier maintenance and a higher software productivity for electrical engineers, compared to an explicit message passing version. This was its main objective. Results obtained for real electrical engineering problems show that HPF can deal efficiently with irregular codes when using an irregular communication library.

In order to improve data-locality and to reduce memory consumption, the next HPF version will use a conjugate gradient method where the preconditioner will be based on domain decomposition using a Schur complement method [6].

In the final step of this work we will add the magnetodynamic formulations.

References

1. A.Bossavit. A rational for edge elements in 3d fields computations. In *IEEE Trans. Mag.*, volume 24, pages 74–79, january 1988.
2. ADAPTOR. High Performance Fortran Compilation System. WWW documentation, Institute for Algorithms and Scientific Computing (SCAI, GMD), 1999. `http://www.gmd.de/SCAI/lab/adaptor`.
3. S. Benkner. Optimizing Irregular HPF Applications Using Halos. In Rolim, J. et al., editor, *Parallel and Distributed Processing, Proceedings of IPPS/SPDP Workshops*, volume 1586 of *Lecture Notes in Computer Science*, pages 1015–1024, San Juan, Puerto Rico, USA, April 1999. Springer.
4. T. Brandes. HPF Library, Language and Compiler Support for Shadow Edges in Data Parallel Irregular Computations. In *CPC 2000, 8th International Workshop on Compilers for Parallel Computers, Aussois, France, January 4-7*, pages 21–34, January 2000.
5. Y.Le Menach, S.Clénet, and F.Piriou. Determination and utilization of the source field in 3d magnetostatic problems. In *IEEE Trans. Mag.*, volume 34, pages 2509–2512, 1998.
6. Barry F. Smith, Petter E. Bjørstad, and William Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.
7. Z.Ren. Influence of R.H.S. on the convergence behaviour of curl-curl equation. In *IEEE Trans. Mag.*, volume 32, pages 655–658, 1996.

Large-Eddy Simulations of Turbulent Flows, from Desktop to Supercomputer

Ugo Piomelli[1], Alberto Scotti[2], and Elias Balaras[3]

[1] University of Maryland, College Park MD 20742, USA,
ugo@eng.umd.edu,
WWW: http://www.glue.umd.edu/~ugo
[2] University of North Carolina, Chapel Hill NC 27599-3300, USA
scotti@marine.unc.edu
[3] University of Maryland, College Park MD 20742, USA,
balaras@eng.umd.edu

Abstract. In this paper, a general introduction to the large-eddy simulation (LES) technique will be given. Modeling and numerical issues that are under study will be described to illustrate the capabilities and requirements of this techniques. A palette of applications will then be presented, chosen on the basis both of their scientific and technological importance, and to highlight the application of LES on a range of machines, with widely different computational capabilities.

1 Introduction

Turbulent flows are ubiquitous in nature and in technological applications. They occur in such diverse fields as metereology, astrophysics, aerospace, mechanical, chemical and environmental engineering. For this reason, turbulence has been the object of study for many centuries. In 1510, Leonardo da Vinci accompanied a drawing of the vortices shed behind a blunt obstacle (Fig. 1) with the following observation:

> Observe the motion of the water surface, which resembles that of hair, that has two motions: one due to the weight of the shaft, the other to the shape of the curls; thus, water has eddying motions, one part of which is due to the principal current, the other to the random and reverse motion.

Despite its importance, and the number of researchers that have studied it theoretically, experimentally and, recently, numerically, turbulence remains one of the open problems in Mechanics.

The equations that govern turbulent flows are the Navier-Stokes equations. For turbulent flows, no exact solutions are available, and their numerical solution is made difficult by the fact that an accurate calculation depends critically on the accurate representation, in space and time, of the coherent fluid structures (eddies) that govern to a very large extent the transfer of momentum and energy. The direct solution of the Navier-Stokes equations (also known as "direct

Fig. 1. Sketch from Leonardo da Vinci's notebooks.

numerical simulation", or DNS) is an extremely expensive endeavor in turbulent flows. Its cost depends on the cube of the Reynolds number, the dimensionless parameter that measures the relative importance of convective and diffusive effects. At present, DNS calculations are limited to flows with Reynolds numbers $O(10^4)$, while most engineering and geophysical applications are characterized by $Re = O(10^6 - 10^9)$.

Practical, predictive, calculations require the use of simplified models. The most commonly used one is the solution of the Reynolds-averaged Navier-Stokes equations (RANS), in which the flow variables are decomposed into a mean and a fluctuating part, as fore-shadowed in da Vinci's observations, and the effect of the turbulent eddies is parameterized globally, through some more-or-less complex turbulence model. This technique is widespread in industrial practice, but turbulence models are found to require *ad hoc* adjustments from one flow to another, due to the strongly flow-dependent nature of the largest eddies, which contribute most to the energy and momentum transfer, and which depend to a very significant extent on the boundary conditions. Furthermore, they fail to give any information on the wavenumber and frequency distribution of the turbulent eddies, which may be important in acoustics, or in problems involving the interaction of fluid with solid structures.

The large-eddy simulation (LES) is a technique intermediate between DNS and RANS, which relies on computing accurately the dynamics of the large eddies while modeling the small, subgrid scales of motion. This method is based on the consideration that, while the large eddies are flow-dependent, the small scales tend to be more universal, as well as isotropic. Furthermore, they react more rapidly to perturbations, and recover equilibrium quickly. Thus, the modelling of the subgrid scales is significantly simpler than that of the large scales, and can be more accurate.

Despite the fact that the small scales are modeled, LES remains a fairly computationally intensive technique. Since the motion of the large scales must

be computed accurately in time and space, fine grids (or high-order schemes) and small time-steps are required. Since the turbulent motions are intrinsically three-dimensional (3D), even flows that are two- or one-dimensional in the mean must be computed using a 3D approach. Finally, to accumulate the averaged statistics needed for the engineering design and analysis, the equations of motion must be integrated for long times.

As a result of these computational requirements, until recently LES has been a research tool, used mostly in academic environments and research laboratories to study the physics of turbulence. Most calculations were carried out on vector machines (Cray X-MP, Y-MP and C90, for instance). Typical computations of flows at moderate Reynolds number required up to 1 million grid points, and used times of the order of 100 CPU hours and more on such machines.

Recently, progress has been made on two fronts. First, the development of advanced models [4, 5] for the small-scale contribution to momentum transfer, the subgrid-scale stresses, allows the accurate prediction of the response of the small scales even in non-equilibrium situations. Secondly, the decreasing cost of computational power has made it possible to perform larger simulations on a day-to-day basis, even using inexpensive desktop workstations. Simulations using 3 million grid points can easily be run on Pentium-based computers. The turn-around time for a mixing-layer simulation that used 5 million points on a dedicated Alpha processor is of the order of two days per integral scale of the flow, a time comparable to what was achievable on a Cray, in which the greater processor speed was often offset by the load of the machine, and the end-user was frequently restricted to a few CPU hours per day. With the increased availability of inexpensive workstation clusters, the application of LES is bound to become more and more affordable. The use of large, massively parallel computers is, however, still required by very advanced, complex applications that may require very large numbers of grid points [$O(10^7)$], and correspondingly long integration times.

In this article, a general introduction to LES will be given. Although particular emphasis will be placed on numerical issues, the main thrust of the paper will not be the algorithmic problems and developments, but rather a discussion of the capabilities and computational requirements of this techniques. A palette of applications will then be presented, chosen on the basis both of their scientific and technological importance, and to highlight the application of LES on a range of machines, with widely different computational capabilities. This article should not be seen as a comprehensive review of the area; the reader interested in more in-depth discussions of the subject is addressed to several recent reviews [1–3].

2 Governing equations

The range of scales present in a turbulent flow is a strong function of the Reynolds number. Consider for instance the mixing layer shown in Fig. 2. The largest eddies in this flow are the spanwise rollers, whose scale is L; a very wide range of smaller scales is present. The energy supplied to the largest turbulent eddies

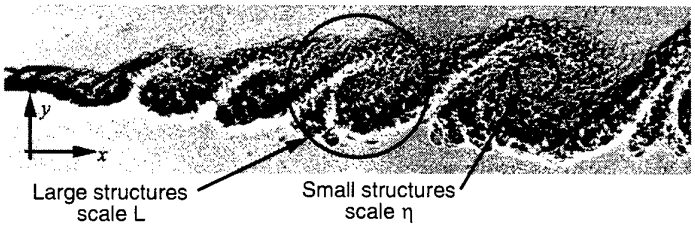


Fig. 2. Visualization of the flow in a mixing layer (from Brown & Roshko [6]). The flow is from left to right; a splitter plate (immediately to the left of the image) separates a high-speed flow (top) from a low-speed one. The two streams then mix, forming the large, quasi-2D rollers in the figure, as well as a range of smaller scales.

by the mean flow is transferred to smaller and smaller scales (energy cascade), and eventually dissipated into heat by the smallest ones. Most of the energy, in fact, is dissipated by eddies contained in a length scale band of about 6η to 60η, where η is the so-called Kolmogorov scale.

In DNS, all the scales of motion, up to and including the dissipative scales of order η must be resolved; since the computational domain must be significantly larger than the large scale L, while the grid size must be of order η, the number of grid points required is proportional to the ratio L/η. It can be shown that this ratio is proportional to $Re^{3/4}$, where the Reynolds number $Re = \Delta U L/\nu$ is based on the velocity difference between the two streams, ΔU, and an integral scale of the flow, L; ν is the kinematic viscosity of the fluid. Thus, the number of grid points needed to perform a three-dimensional DNS scales like the 9/4 power of the Reynolds number.

The time-scale of the smallest eddies also supplies a bound for the maximum time-step allowed: since the ratio of the integral time-scale of the flow to the Kolmogorov time-scale is also proportional to $Re^{3/4}$ the number of time-steps required to advance the solution by a fixed time has the same dependence on Re. Assuming that the CPU time required by a numerical algorithm is proportional to the total number of points N, the cost of a calculation will depend on the product of the number of points by the number of time-steps, hence to Re^3.

In an LES only the large scales of motion must be resolved. The similarity of the small scales, which only transmit energy to smaller scales, and the fact that the global dissipation level is set by the large scales (even though the dissipation takes place at the small-scale level) are exploited by SGS models, whose main purpose is to reproduce the energy transfer accurately, at least in a statistical sense. When the filter cutoff is in the inertial region of the spectrum (i.e., in the wave-number range in which the energy cascade takes place), therefore, the resolution required by an LES is nearly independent of the Reynolds number.

In wall-bounded flows, in which the scale of the large, energy-carrying eddies is Reynolds-number-dependent, the situation is less favorable. The cost of an LES is still, however, significantly reduced over that of a DNS.

To separate the large from the small scales, LES is based on the definition of a filtering operation: a filtered (or resolved, or large-scale) variable, denoted by an overbar, is defined as

$$\overline{f}(\mathbf{x}) = \int_D f(\mathbf{x}')G(\mathbf{x},\mathbf{x}';\overline{\Delta})d\mathbf{x}', \tag{1}$$

where D is the entire domain, G is the *filter* function, and $\overline{\Delta}$, the filter width, is a parameter that determines the size of the largest eddy removed by the filtering operation. The filter function determines the size and structure of the small scales. It is easy to show that, if G is a function of $\mathbf{x}-\mathbf{x}'$ only, differentiation and the filtering operation commute [7].

The most commonly-used filter functions are the sharp Fourier cutoff filter, best defined in wave space[1]

$$\widehat{G}(k) = \begin{cases} 1 \text{ if } k \leq \pi/\overline{\Delta} \\ 0 \text{ otherwise,} \end{cases} \tag{2}$$

the Gaussian filter,

$$G(x) = \sqrt{\frac{6}{\pi\overline{\Delta}^2}}\exp\left(-\frac{6x^2}{\overline{\Delta}^2}\right), \tag{3}$$

and the top-hat filter in real space:

$$G(x) = \begin{cases} 1/\overline{\Delta} \text{ if } |x| \leq \overline{\Delta}/2 \\ 0 \quad \text{otherwise,} \end{cases} \tag{4}$$

For uniform filter width $\overline{\Delta}$ the filters above are mean-preserving and commute with differentiation.

The effect of filtering a test function with increasing filter-width is shown in Fig. 3. Although an increasing range of small scales is removed as $\overline{\Delta}$ is increased, the large-scale structure of the signal is preserved. In RANS, on the other hand, the effect of all turbulent eddies would be removed by the averaging procedure.

In LES the filtering operation (1) is applied formally to the governing equations; this results in the filtered equations of motion, which are solved in LES. For an incompressible flow of a Newtonian fluid, they take the following form:

$$\frac{\partial\overline{u}_i}{\partial x_i} = 0. \tag{5}$$

$$\frac{\partial\overline{u}_i}{\partial t} + \frac{\partial}{\partial x_j}\left(\overline{u}_i\overline{u}_j\right) = -\frac{1}{\rho}\frac{\partial\overline{p}}{\partial x_i} - \frac{\partial\tau_{ij}}{\partial x_j} + \nu\frac{\partial^2\overline{u}_i}{\partial x_j\partial x_j}. \tag{6}$$

The filtered Navier-Stokes equations written above govern the evolution of the large, energy-carrying, scales of motion. The effect of the small scales appears through a subgrid-scale (SGS) stress term,

$$\tau_{ij} = \overline{u_iu_j} - \overline{u}_i\overline{u}_j, \tag{7}$$

that must be modeled to achieve closure of the system of equations.

[1] A quantity denoted by a caret $\widehat{\cdot}$ is the complex Fourier coefficient of the original quantity.

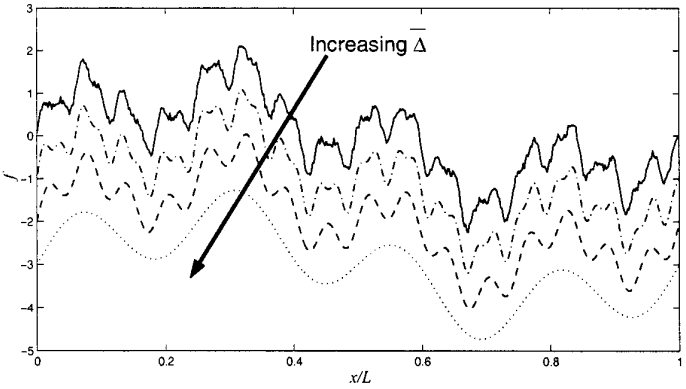


Fig. 3. Effect of filtering a test function with increasing filter-width $\overline{\Delta}$.

3 Subgrid-scale models

In LES the dissipative scales of motion are resolved poorly, or not at all. The main role of the subgrid-scale model is, therefore, to remove energy from the resolved scales, mimicking the drain that is associated with the energy cascade. Most subgrid scale models are eddy-viscosity models of the form

$$\tau_{ij} - \frac{\delta_{ij}}{3}\tau_{kk} = -2\nu_T \overline{S}_{ij}, \tag{8}$$

that relate the subgrid-scale stresses τ_{ij} to the large-scale strain-rate tensor $\overline{S}_{ij} = (\partial\overline{u}_i/\partial x_j + \partial\overline{u}_j/\partial x_i)/2$. In most cases the equilibrium assumption (namely, that the small scales are in equilibrium, and dissipate entirely and instantaneously all the energy they receive from the resolved ones) is made to simplify the problem further and obtain an algebraic model for the eddy viscosity [8]:

$$\nu_T = C\overline{\Delta}^2|\overline{S}|\overline{S}_{ij}; \qquad |\overline{S}| = (2\overline{S}_{ij}\overline{S}_{ij})^{1/2}. \tag{9}$$

This model is known as the "Smagorinsky model". The value of the coefficient C can be determined from isotropic turbulence decay [9]; if the cutoff in the inertial subrange, the Smagorinsky constant $C_s = \sqrt{C}$ takes values between 0.18 and 0.23 (and $C \simeq 0.032 - 0.053$). In the presence of shear, near solid boundaries or in transitional follows, however, it has been found that C must be decreased. This has been accomplished by various types of *ad hoc* corrections such as van Driest damping [10] or intermittency functions [11].

More advanced models, that do not suffer from the shortcomings of the Smagorinsky model (excessive dissipation, incorrect asymptotic behavior near solid surfaces, need to adjust the constant in regions of laminar flow or high shear) have been developed recently. The introduction of dynamic modeling ideas [4] has spurred significant progress in the subgrid-scale modeling of non-equilibrium

flows. In dynamic models the coefficient(s) of the model are determined as the calculation progresses, based on the energy content of the smallest resolved scale, rather than input *a priori* as in the standard Smagorinsky [8] model. A modification of this model was proposed by Meneveau *et al.* [5], which has been shown to give accurate results in non-equilibrium flows in which other models fail [12].

Turbulence theory (in particular the Eddy-Damped Quasi-Normal Markovian theory) has also been successful in aiding the development of SGS models. The Chollet-Lesieur [13, 14] model, as well as the structure-function [15] and filtered-structure-function models [16] have been applied with some success to several flows.

A detailed discussion of SGS models is beyond the scope of this paper. The interested reader is referred to the review articles referenced above [1–3].

4 Numerical methods

In large-eddy simulations the governing equations (5-6) are discretized and solved numerically. Although only the large scales of motion are resolved, the range of scales present is still significant. In this section, a brief overview of the numerical requirements of LES will be given.

4.1 Time advancement

The choice of the time advancement method is usually determined by the requirements that numerical stability be assured, and that the turbulent motions be accurately resolved in time. Two stability limits apply to large-eddy simulations. The first is the viscous condition, that requires that the time-step Δt be less than $\Delta t_v = \sigma \Delta y^2/\nu$ (where σ depends on the time advancement chosen). The CFL condition requires that Δt be less than $\Delta t_c = \mathrm{CFL}\Delta x/u$, where the maximum allowable Courant number CFL also depends on the numerical scheme used. Finally, the physical constraint requires Δt to be less than the time scale of the smallest resolved scale of motion, $\tau \sim \Delta x/U_c$ (where U_c is a convective velocity of the same order as the outer velocity).

In many cases (especially in wall-bounded flows, and at low Reynolds numbers), the viscous condition demands a much smaller time-step than the other two; for this reason, the diffusive terms of the governing equations are often advanced using implicit schemes (typically, the second-order Crank-Nicolson scheme). Since, however, Δt_c and τ are of the same order of magnitude, the convective term can be advanced by explicit schemes such as the second-order Adams-Bashforth method, or third- or fourth-order Runge-Kutta schemes.

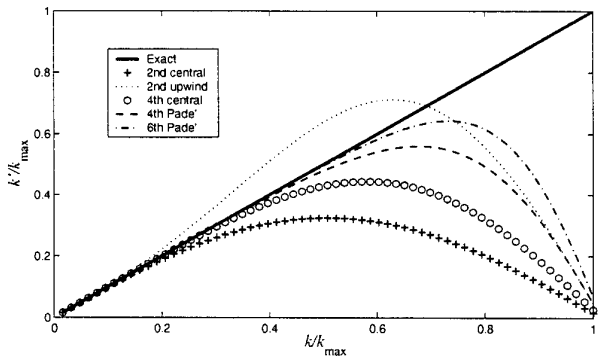


Fig. 4. Modified wave-number for various differencing schemes.

4.2 Spatial discretization

The analytical derivative of a complex exponential $f(x) = e^{ikx}$ is $f'(x) = ike^{ikx}$; if f is differentiated numerically, however, the result is

$$\frac{\delta f}{\delta x} = ik'e^{ikx}, \tag{10}$$

where k' is the "modified wave-number". A modified wave-number corresponds to each differencing scheme. Its real part represents the attenuation of the computed derivative compared to the actual one, whereas a non-zero imaginary part of k' indicates that phase errors are introduced by the numerical differentiation. Figure 4 shows the real part of the modified wave-numbers for various schemes. For a second-order centered scheme, for instance, $k' = k\sin(k\Delta x)/(k\Delta x)$. For small wave-numbers k the numerical derivative is quite accurate; high wave-number fluctuations, however, are resolved poorly. No phase errors are introduced.

The need to resolve accurately high wave-number turbulent fluctuations implies that either low-order schemes are used on very fine meshes (such that, for the smallest scales that are physically important, $k' \simeq k$), or that higher-order schemes are employed on coarser meshes. High-order schemes are more expensive, in terms of computational resources, than low-order ones, but the increase in accuracy they afford (for a given mesh) often justifies their use.

4.3 Conservation

It is particularly important, in large-eddy simulations of transitional and turbulent flows, that the numerical scheme preserves the conservation properties of the Navier-Stokes equations. In the limit $Re \to \infty$, the Navier-Stokes equations conserve mass, momentum, energy and vorticity in the interior of the flow: the integral of these quantities over the computational domain can only be affected

through the boundaries. Some numerical schemes, however, do not preserve this property. For instance, the convective term in the momentum equations can be cast in several ways:

$$\text{Advective form} : u_j \frac{\partial u_i}{\partial x_j}, \tag{11}$$

$$\text{Divergence form} : \frac{\partial}{\partial x_j}(u_i u_j), \tag{12}$$

$$\text{Rotational form} : \epsilon_{ijk} u_j \omega_k - \frac{\partial}{\partial x_i}(u_j u_j/2), \tag{13}$$

$$\text{Skew-symmetric form} : \frac{1}{2}\left[u_j \frac{\partial u_i}{\partial x_j} + \frac{\partial}{\partial x_j}(u_i u_j)\right], \tag{14}$$

where $\omega_k = \epsilon_{kij}\partial u_j/\partial x_i$. It is easy to show (Morinishi *et al.* [17]) that, if a typical co-located finite-difference scheme is used, the first form does not conserve either momentum or energy, the second conserves momentum but not energy, the others conserve both. If, on the other hand, a control-volume approach is used, the divergence form conserves energy but the pressure-gradient term does not. With a staggered grid, the divergence form preserves the conservation properties of the Navier-Stokes equations if central, second-order accurate differences are used.

Upwind schemes also have very undesirable effects on the conservation properties of the calculation, as does the explicit addition of artificial dissipation. Even mildly upwind-biased schemes result in a significant loss of accuracy. In incompressible flows, these type of methods are not suited to LES, and should be avoided.

4.4 Complex geometries

For applications to complex geometries, single-block, Cartesian meshes are inadequate, since they do not give the required flexibility. One alternative is the use of body-fitted curvilinear grids. LES codes in generalized coordinates have been used, among others by Zang *et al.* [21, 22] (who applied it to a Cartesian geometry, the lid-driven cavity [21], and to the study of coastal up-welling [22, 23]), Beaudan and Moin [24] and Jordan [25]. Jordan [25] examined the issue of filtering in curvilinear coordinates, and concluded that filtering the transformed (in the generalized coordinates) equations directly in the computational space is better than performing the filtering either of the transformed equations in real space, or of the untransformed equations in Cartesian space.

Even if curvilinear grids are used, the application of LES to complex geometries might be limited by resolution requirements. In the presence of a solid boundary, for instance, a very fine mesh is required to resolve the wall layer. Kravchenko *et al.* [26] used zonal embedded meshes and a numerical method based on B-splines to compute the flow in a two-dimensional channel, and around a circular cylinder. The use of the B-splines allows use of an arbitrarily high order of accuracy for the differentiation, and accurate interpolation at the interface

between the zones. A typical grid for the channel flow simulations is shown in Fig. 5, which evidences the different spanwise resolution in the various layers, in addition to the traditional stretching in the wall-normal direction. The use of zonal grids allowed Kravchenko *et al.* [26] to increase the Reynolds number of the calculations substantially: they performed an LES of the flow at $Re_c = 109\,410$ using 9 embedded zones allowed them to resolve the wall-layer using a total of 2 million points. A single-zone mesh with the same resolution would have under-resolved the wall layer severely. The mean velocity profile was in excellent agreement with the experimental data.

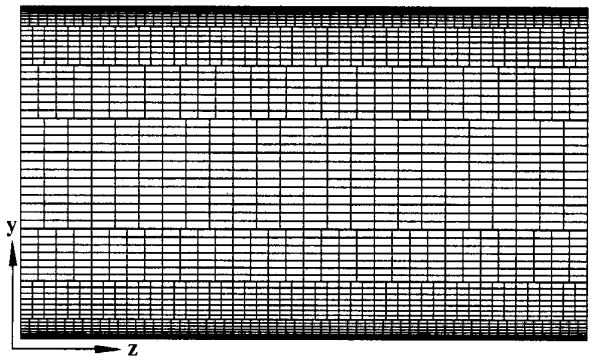


Fig. 5. Zonal embedded grid with fine grid zones near the walls and coarse zones in the middle of the channel. The flow is into the paper. Reproduced with permission from Kravchenko *et al.* [26]

Very few applications of LES on unstructured meshes have been reported to date. Jansen [29] showed results for isotropic turbulence and plane channel. For the plane channel, the results were in fair agreement with DNS data (the peak streamwise turbulence intensity, for instance, was 15% higher than that obtained in the DNS), but slightly better than the results of finite-difference calculations on the same mesh. Simulations of the flow over a low-Reynolds number airfoil using this method [28] were in fair agreement with experimental data. Knight *et al.* [30] computed isotropic turbulence decay using tetrahedral meshes, and compared the Smagorinsky model with results obtained relying on the numerical dissipation to drain energy from the large scales. They found that the inclusion of an SGS model gave improved results.

While high-order schemes can be applied fairly easily in simple geometries, in complex configurations their use is rather difficult. Present applications of LES to relatively complex flows, therefore, tend to use second-order schemes; the increasing use of LES on body-fitted grids for applications to flows of engineering interest, indicates that, at least in the immediate future, second-order accurate schemes are going to increase their popularity, at the expense of the spectral

methods that have been used frequently in the past. Explicit filtering of the governing equations, with filter widths larger than the grid size may be required in such circumstances.

5 Applications: flow in an accelerating boundary layer

A boundary layer is the region of fluid flow nearest to a solid body, in which viscous effects (i.e., diffusion) are important. Turbulent boundary layers occur in many technological applications, and are often subjected to favorable pressure gradients that result in an acceleration of the velocity at the edge of the boundary layer, the free-stream velocity. Figure 5 illustrates schematically the boundary layer that occurs at the leading edge of an airplane wing. The fluid is accelerated as it turns over the top side of the airfoil from the stagnation point, where its velocity is zero.

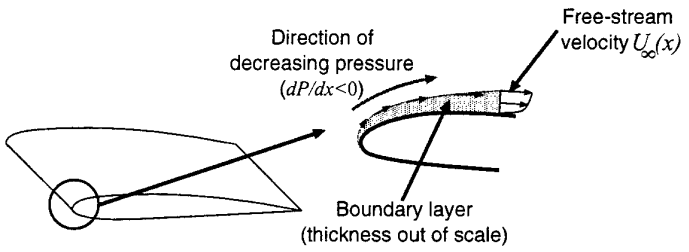


Fig. 6. Sketch of the flow near the leading edge of an airfoil.

Despite the importance of this type of flow fields, however, they are not as well understood as the canonical zero-pressure-gradient boundary layer, due to the much wider parameter space, and to the difficulty in determining universal scaling laws similar to those for the zero-pressure-gradient case. In fact, a large percentage of the investigations of accelerating flows to date have concentrated on self-similar cases, in which such scaling laws can be found.

It is recognized that, if the acceleration is sufficiently strong, turbulence cannot be sustained. In self-similar accelerating boundary layer, this phenomenon takes place when the acceleration parameter K reaches a critical value:

$$K = \frac{\nu}{U_\infty^2}\frac{dU_\infty}{dx} = -\frac{\nu}{\rho U_\infty^3}\frac{dP_\infty}{dx} \simeq 3 \times 10^{-6}. \qquad (15)$$

The RANS approach, which is often used in aeronautical applications, has difficulty dealing with reversion of a turbulent flow to a laminar one, and into the re-transition of the flow, that becomes turbulent again as the acceleration ceases on the suction (upper) side of the airfoil. Large-eddy simulation can help in understanding the physics that cause reversion and re-transition, as well as

provide accurate data that can be used for the development and validation of lower-level RANS models to be used in engineering design.

In particular, experimental evidence indicates that the dynamics of the coherent eddies play an important role in the reversion. An improved understanding of the dynamics of these eddies in boundary layers subjected to a favorable pressure gradient would be extremely beneficial. Apart from the considerations about momentum transfer and mixing also valid in other flows, an additional motivating factor is provided here by the consideration that most of the theoretical attempts to derive scaling laws are often based on multiple-scale approximations that assume little or no interaction between inner and outer layers. The most direct way to establish the validity of this assumption is by studying the coherent eddies in the wall layer. Unlike RANS solutions, in which only the average flow-field is computed, LES can supply information on the behavior of the coherent structures.

Piomelli and co-workers [31] studied the velocity fields obtained from the large-eddy simulation (LES) of accelerating boundary layers with the aim to improve the understanding of the dynamics of the coherent vortices in the re-laminarizing flows. To separate the effect of the pressure gradient from that of curvature, the calculation of the boundary layer on a flat plate with an accelerating free-stream was carried out; the configuration is similar to the flow on the lower wall of a wind-tunnel in which the upper wall converges, as sketched in Fig. 7. The computational domain is the shaded area in the figure.

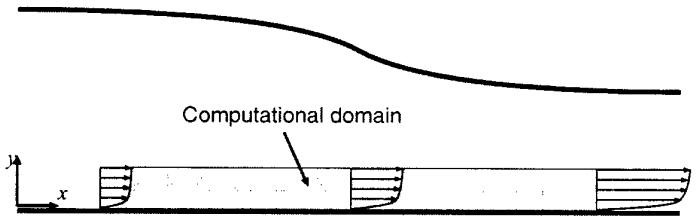


Fig. 7. Sketch of the physical configuration. Accelerating boundary layer.

Two computations were examined: one in which the acceleration is relatively mild (the maximum velocity increases by 35% over the computational domain, and $K < 3 \times 10^{-6}$ everywhere), and a strong-acceleration case in which the velocity increases by almost 150%, and $K > 3 \times 10^{-6}$ for a significant portion of the flow. The modification of the turbulence structure in accelerating flows was emphasized, and it was shown how the acceleration can be associated to lower turbulence levels and to the dynamics of the quasi-streamwise coherent vortices.

5.1 Numerical method

The governing equations(5–6) are integrated numerically using the fractional time-step method [18, 19], in which first the Helmholtz equation is solved to ob-

tain an estimate of the velocity field that does not satisfy mass conservation; the pressure is then computed by solving Poisson's equation, the estimated velocity field supplying the source term. When a pressure correction is applied, the resulting velocity will be a divergence-free solution of the Navier-Stokes equations. If the Navier-Stokes equations are written as

$$\frac{\partial \overline{u}_i}{\partial t} = -\frac{\partial \overline{p}}{\partial x_i} - H_i + \nu \nabla^2 \overline{u}_i, \tag{16}$$

where H_i contains the nonlinear term and the SGS stresses, the time-advancement sequence based on the second-order-accurate Adams-Bashforth method consists of the following steps:

1. Velocity prediction (Helmholtz equation):

$$v_j - u_j^n = \Delta t \left[\frac{3}{2}(-H_j^n + \nabla^2 \overline{u}_j{}^n) - \frac{1}{2}(-H_j^{n-1} + \nabla^2 \overline{u}_j{}^{n-1}) \right]; \tag{17}$$

2. Poisson solution:

$$\nabla^2 \overline{p} = \frac{1}{\Delta t} \frac{\partial v_j}{\partial x_j}; \tag{18}$$

3. Velocity correction:

$$\overline{u}_j{}^{n+1} = v_j - \Delta t \frac{\partial \overline{p}}{\partial x_j}; \tag{19}$$

v_j is the estimated velocity. This time-advancement scheme is second-order-accurate in time. The code uses central differences on a staggered mesh, and is second-order accurate in space as well. Discretization of the Poisson equation (18) results in an hepta-diagonal matrix that can be solved directly if the grid is uniform in at least one direction.

The calculations were performed on a domain of size $400 \times 25 \times 25$. All lengths are normalized with respect to the inflow displacement thickness δ_0^*; the displacement thickness is an integral length scale defined as

$$\delta^* = \int_0^\infty \left(1 - \frac{U}{U_\infty}\right) dy, \tag{20}$$

where U is the average streamwise velocity. The calculations used $256 \times 48 \times 64$ grid points. A grid-refinement study was performed in the strong-acceleration case, in which the number of grid points was increased by 50% in each direction. In the accelerating-flow region ($x/\delta_0^* < 320$) the results on the coarser mesh matched very well those obtained with the finer one. In the re-transitioning area, the qualitative behaviour of the flow was captured correctly, but some differences (of the order of 15%) were observed in the statistical quantities. The Lagrangian dynamic eddy viscosity model [5] was used to parameterize the SGS stresses.

The cost of the computations, was 2.2×10^{-5} CPU seconds per time-step and grid point on a 300 MHz Pentium II running Linux. Out of this CPU time,

37% is devoted to the computation of the RHS, 25% to the computation of the turbulent viscosity, 12% to solve the Poisson equation, and 10% to update the velocity field and impose boundary conditions. The rest of the CPU is consumed by I/O and computation of statistical quantities. Typically a computation on a 10^6 grid requires approximately 42 hours of CPU to obtain converged statistics (sampling over 1.5 flow-through times). It is interesting to observe that the cost of solving the Poisson equation is a small fraction of the total cost when a direct solver (as in the present case) is used. Any other choice of solution method, like multigrid methods, conjugate gradient methods, etc. would substantially increase the cost of this step, which can account for a large fraction of the total cost, depending on the problem and the computational grid.

5.2 Results

The free-stream velocity obtained from the calculation, U_∞, the pressure parameter K and the momentum-thickness Reynolds number, $Re_\theta = \theta U_\infty/\nu$, where θ is the momentum thickness defined as

$$\theta = \int_0^\infty \left(1 - \frac{U}{U_\infty}\right) \frac{U}{U_\infty} dy, \tag{21}$$

are shown in Fig. 8 for the two cases examined. In the strong acceleration case, despite the presence of a fairly extended region in which K exceeds 3×10^{-6}, the Reynolds number never goes below the critical value $Re_\theta \simeq 350$. Thus one would expect the flow to become less turbulent, but not to revert fully into a laminar one.

The streamwise development of several time-averaged and integral quantities is shown in Fig. 9. As a result of the free-stream acceleration, the boundary layer becomes thinner, as shown by the distributions of δ^* and θ. The skin friction coefficient based on the local free-stream velocity, $C_f = 2\tau_w/\rho U_\infty^2$ (where τ_w is the wall stress), initially increases, but, as the flow begins to relaminarize, it decreases in both the mild- and strong-acceleration case.

Although the pressure-gradient parameter K is well above the critical value of 3×10^{-6} in the strongly accelerating case, the acceleration is not sustained long enough for the Reynolds number to be reduced below the critical value, $Re_\theta \simeq 350$. Thus, full relaminarization does not occur; the shape factor H only reaches a value of 1.6 (the shape factor associated with the laminar Falkner-Skan similarity profile for sink flows of this type is 2.24). The mean velocity profile, however, is significantly affected by the acceleration, even in the mild acceleration case.

As the flow is gradually accelerated, the turbulence adjusts to the perturbation; the turbulent quantities, however, lag the mean flow. The turbulent kinetic energy, for instance, increases in absolute levels, although not as fast as the kinetic energy of the mean flow. Thus, the contours of the turbulent kinetic energy normalized by the kinetic energy of the mean flow, shown in Fig. 10, highlight a significant drop in the turbulent kinetic energy in the region of acceleration.

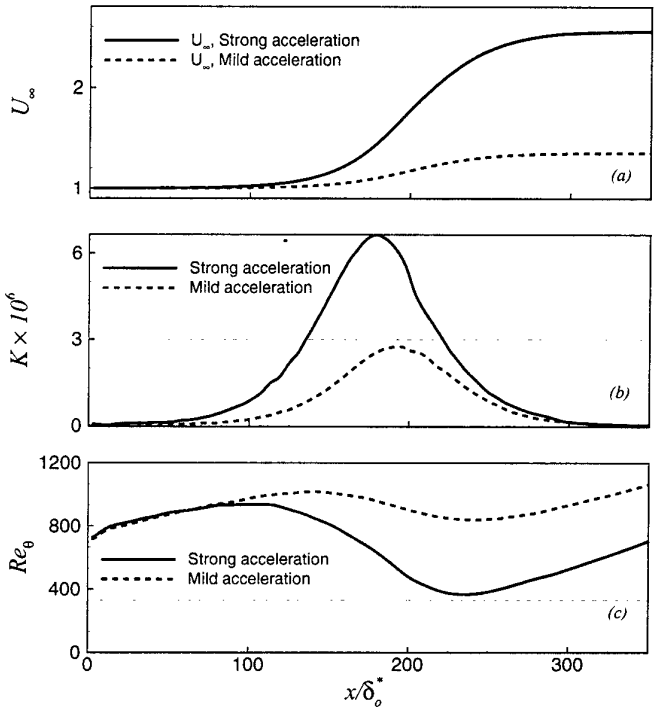


Fig. 8. Spatial development of the free-stream velocity U_∞, the acceleration parameter K, and the momentum-thickness Reynolds number Re_θ in the accelerating boundary layer.

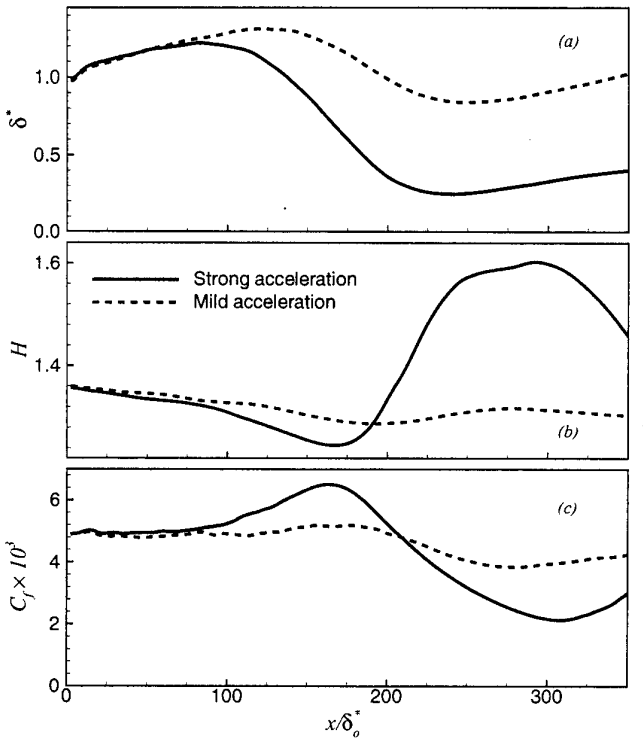


Fig. 9. Spatial development of mean quantities in the accelerating boundary layer. *(a)* Displacement thickness δ^*; *(b)* shape factor H; *(c)* skin-friction coefficient C_f.

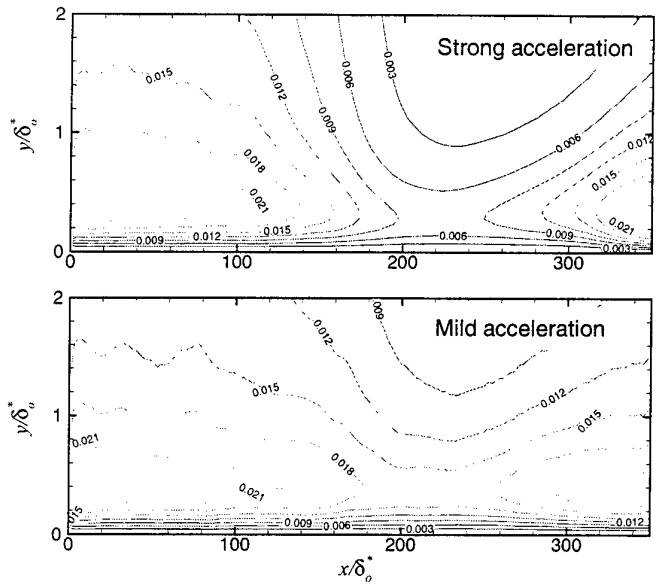


Fig. 10. Contours of the turbulent kinetic energy, normalized by the free-stream kinetic energy in the accelerating boundary layer.

Paradoxically, in many turbulent flows, whenever energy is added through the mean flow, the energy of the turbulence initially decreases, as the coherent vortices adapt to the perturbation. This process often involves disruption of the vortical structures prior to their re-generation. Such is the case in this configuration as well: the vortical structures are visualized in Fig. 11 as isosurfaces of the second invariant of the velocity-gradient tensor, Q, a useful quantity to visualize the regions of high rotation that correspond to the coherent vortices. In the zero-pressure-gradient region near the inflow (top picture) many vortices can be observed, and they are roughly aligned with the flow direction, but form an angle to the wall. This picture is typical of zero-pressure-gradient boundary layers. In the accelerating region, on the other hand, fewer eddies are observed, and those present are more elongated and more closely aligned in the streamwise direction. This structure can be explained based on the fact that the mean velocity gradient has the effect of stretching and re-orienting the vortices. As they are stretched, their vorticity is increased by conservation of angular momentum, while their radius is decreased. The smaller, more intense eddies thus generated are more susceptible to be dissipated by viscous effects.

This calculation highlights a significant advantage of LES over lower-level models. Whenever the coherent eddies play an important role in the flow evolution, RANS calculations (in which the effect of all turbulent eddies is averaged out) cannot predict the flow development accurately. LES, on the other hand,

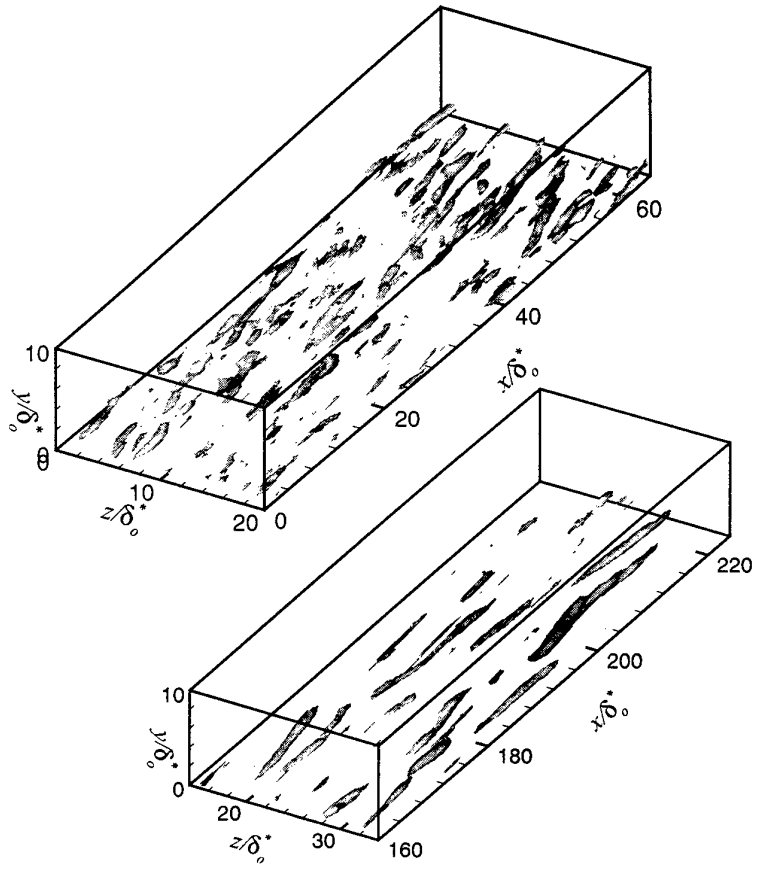


Fig. 11. Instantaneous iso-surfaces of $Q(\delta_o^*/U_o)^2 = 0.02$ in the strong-acceleration case. Top: zero-pressure-gradient region. Bottom: acceleration region.

has a better chance of following the dynamics of the coherent structures, as well as their response to the imposed perturbations.

6 Applications: flow in an oscillating channel

6.1 Motivation

Inherent unsteadiness of the driving conditions characterizes many turbulent flows, both natural (*e.g.* the gravity wave induced in ocean-bottom boundary layers, the blood flow in large arteries, the flow of air in lungs) and artificial (such as the flow in the intake of a combustion engine or the flow in certain heat exchangers). The characterization of unsteady boundary layers is crucial to many disciplines, such as the study of sediment transport in coastal waters, the biology of blood circulation, and so on; moreover, as was pointed out by Sarpkaya [32], by looking at features that are common to steady and unsteady boundary layers, we may better understand the underlying physics of turbulent flows altogether. As already recognized by Binder *et al.* [33], there are no special technical difficulties in performing DNS of pulsating flows. On the other hand, the same authors point out that the oscillating nature of the forcing is felt by the small scales too, so that before trusting the outcome of a LES based on standard closures, a careful (*a posteriori*) comparison with DNS has to be done. This is particularly true for eddy viscosity models, which rely on the combined assumptions that the SGS stress tensor τ_{ij} is aligned with the rate of strain and that the eddy viscosity is proportional to the magnitude of the stress. The latter postulate is somewhat relaxed for the dynamic Smagorinsky model of Germano *et al.* [4], since the eddy viscosity depends on the flux of energy towards the subgrid scales.

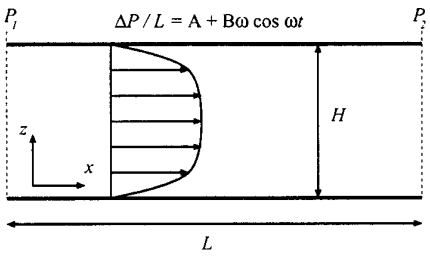


Fig. 12. Sketch of the physical configuration. Oscillating channel flow.

To study the response of turbulence to an oscillating mean flow, a plane-channel flow driven by an oscillating pressure gradient was studied. The physical configuration is illustrated in Fig. 12: the flow between two flat plates that extend to $\pm\infty$ in the streamwise (x) and spanwise (y) directions is simulated.

To drive this periodic flow, a pressure gradient per unit length is introduced on the right-hand-side of the Navier-Stokes equations as a source term. In the case under investigation, this pressure gradient is given by $1 \times 10^{-4} + \omega \cos \omega t$, where ω is the angular frequency of the oscillation. This is the kind of flow considered by Binder *et al.* [33]. The flow admits a laminar solution, which is a trivial extension of the Stokes problem. The flow first decelerates (as it is subjected to the adverse pressure gradient during the first half of the cycle), then accelerates again. During the acceleration phase, as observed before, the flow tends to relaminarize, whereas the adverse-pressure-gradient has the opposite effect, and makes the flow more turbulent.

Since the core of the flow, where the velocity is large, is dominated by convective effects, while the regions near the solid boundary, where the velocity gradients are significant, are dominated by diffusive effects, there is a disparity in time-scales between these two regions: the diffusive time-scale being smaller than the convective one by orders of magnitude. Thus, as the frequency is changed, one would expect a significantly different coupling between the near-wall region (the inner layer) and the core of the flow (the outer layer). To study this coupling, calculations were carried out for a range of frequencies.

Although the geometry is rather simple, and the grids used relatively coarse, this calculation still requires a large amount of CPU time. This is due to the long integration time necessary to achieve convergence. Since phase-averaged data is required, between eight and ten cycles of the oscillation are needed to obtain converged statistical samples. If the frequency is low, the equations of motion must be integrated for very long integration.

6.2 Numerical method

The starting point for this calculation is a well-known serial spectral code for the solution of the filtered Navier-Stokes equation in a channel geometry [34, 35]. Fourier expansions are used in the homogeneous (horizontal) directions, while Chebychev collocation is used in the vertical direction. The code is very highly optimized for a vector architecture. Time-advancement is performed using the fractional time-step method described above; however, the implicit Crank-Nicolson method is used for the vertical diffusion and a low-storage third-order Runge-Kutta scheme is employed for the remaining terms. The procedure described in Section 5.1 still applies, with few obvious modifications. Each sub-step of the Runge-Kutta scheme follows the sequence:

1. Compute the nonlinear terms H_i^n and the horizontal diffusive terms $\nu \nabla^2 \overline{u}_i{}^n$ with information at time t_n. Both these terms are computed in real space.
2. Transform the right-hand side $H^n + \nu \nabla^2 \overline{u}_i{}^n$ into Fourier space.
3. Update the predicted solution in Fourier space:

$$\left(1 - \frac{\nu \Delta t}{2} \frac{\partial}{\partial z}\right) \widehat{v}_j = \left(1 + \frac{\nu \Delta t}{2} \frac{\partial}{\partial z}\right) \widehat{\overline{u}}_j^n + \Delta t \widehat{H}^n \qquad (22)$$

by solving implicitly the vertical diffusive problem. Since a Chebychev collocation method is used in the vertical direction z a full matrix obtains for

each mode, which is inverted iteratively by a Generalized Minimum Residual method.

4. Solve the Poisson problem for the pressure in Fourier space:

$$\left(\frac{\partial}{\partial z} - k_x^2 - k_y^2\right)\widehat{\overline{p}} = \frac{1}{\Delta t}\left[-i(k_x + k_y) + \frac{\partial}{\partial z}\right]\widehat{v}_j \tag{23}$$

again using a Generalized Minimum Residual method to solve the system of linear equations.

5. Update the solution:

$$\widehat{\overline{u}}_j^n = \widehat{v}_j - \Delta t \nabla \overline{p}. \tag{24}$$

An initial series of numerical experiments was performed on an SGI Origin 2000 with 32 R10000 processors running at 195 MHz, each equipped with 640 Mb of Ram and 4Mb of cache, owned by the University of North Carolina. Each processor is rated at 390 MFLOPS. Four discretizations were chosen, 32×32×32, 64×64×64, 128×128×96 and 128×192×128. The serial code experience a drop in performance as the domain grows, from 60 MFLOPS to about 19. This is due to the limited cache, which acts as a bottleneck. The problem is made more acute by the fact that the discrete Fourier transform, which is the heart of the code, is a nonlocal operation. Frigo and Johnson [36] performed extensive testing of different FFT routines on cache based machines, and, without exception, all routines showed a marked slowdown when a certain critical size (both machine- and routine-dependent) is reached (see, for instance, Fig. 4 of their paper).

6.3 The parallel code

The current trend in supercomputer technology is towards achieving raw computational power by assembling a large number of relatively inexpensive nodes, based on mass produced RISC CPUs connected by high-speed data path. Examples are the Origin 2000 by SGI (R10000), the IBM SP/6000 (Power PC) and the Cray T3D (ALPHA). While it is appealing to be able to obtain large theoretical computational speeds at a fraction of the cost of traditional vector based machines, this paradigmatic shift requires a re-examination of the existing codes. A case in point is the spectral Navier-Stokes solver discussed in the previous section. To parallelize it, we begin by noticing that the computationally intensive steps of solving the Helmholtz and Poisson problems amount to solving $i_{max} \times j_{max}$ 1D problems, where i_{max} (j_{max}) is the number of collocation points in the streamwise (spanwise) direction.

The load then can be distributed among p processor. A Single Program Multiple Data (SPMD) approach was adopted. Each process executes essentially the same operations on different portions of the domain, which are private to them. Message passing is used to exchange information between processes, using Message Passing Interface (MPI) library calls.

The domain is sliced along either the z or the x direction, and each processor owns a slice. During the computation of the nonlinear term, the domain is

sliced along the z direction (see Fig. 13). When vertical derivatives are needed, a transposition is performed, in which process j sends sub-block i of the domain to process i, and receives in turn the sub-block j from process i. MPI implements this kind of alltoall scatter/gather operation transparently. After the nonlinear term is calculated, the domain is swapped so that each process owns vertical slices, and the Helmholtz and Poisson problems are solved without further swapping. At the end, the solution is swapped back into horizontal slices and the cycle begins again. Incidentally, this approach predates the use of parallel computers, being used for DNS on Cray X-MP to feed the fields into core memory one slice at a time (see, for example, [37]).

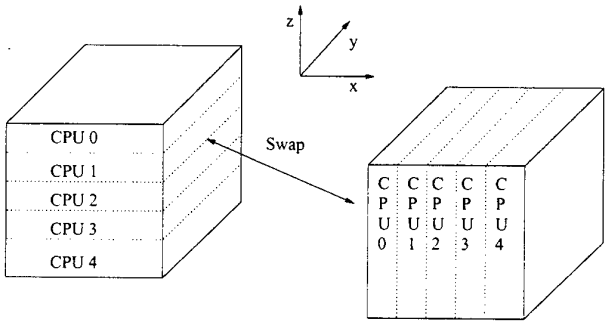


Fig. 13. The domain is split among processes (CPUs) either along the z (left) or the x (right) coordinate. An alltoall scatter/gather is used to go from one configuration to the other.

6.4 Speedup and scalability

The performance of a parallel program is measured by the speedup factor S, defined as the ratio between the execution time T_σ of the serial program and the execution time of the parallel version T_π (see Pacheco [38]). In our approach, the load is evenly balanced between processes (with the negligible exception of I/O, which is handled by one process), so that an equivalent measure is the efficiency E, defined as the ratio between T_σ and the total time consumed by the p processes. In general, for a given machine, $E = E(n, p)$, where n is the size of the problem being solved. In Table 1 we show the efficiency for different values of n and p, with the relative MFLOPS in parenthesis.

The striking result is that it is possible to achieve a super-linear speedup. This is made possible by the fact that the smaller parallel threads use the cache more efficiently than the serial code. For instance, for the grid $128\times128\times96$, the serial code reuses on average a L2 cache line 4.6 times before discarding it; using 4 processes the value per process increases to 7.3, while with 8 becomes as high as 21.4. The gain is of course offset by the overhead generated by message

Size	$p = 2$	$p = 4$	$p = 8$	$p = 16$
32×32×32	.8 (84)			
64×64×64	.93 (81)	.81 (120)		
128×128×96		1.1 (88)	1.3 (168)	
128×192×128			1.1 (168)	.91 (276)

Table 1. Efficiency of spectral parallel Navier Stokes solver and (in parentheses) achieved MFLOP rate.

passing. However, due to the efficient implementation of MPI on the Origin 2000, we found that the time spent swapping data between processes represents less that 10% of the total time, in the worst case.

6.5 Results

The Reynolds number based on channel height and the time-averaged centerline velocity was 7500 for all calculations. Simulations were carried out for several values of the frequency of the driving pressure-gradient, resulting in a Reynolds number, based on the thickness of the laminar oscillating layer, $\delta = (2\nu/\omega)^{1/2}$ and the oscillating component of the velocity, ranging between $Re_\delta = 100$ and 1000. The low Re_δ case was simulated using both a DNS on a $128 \times 128 \times 96$ grid, and an LES using the dynamic eddy-viscosity model [4] on the same domain, discretized using a $32 \times 32 \times 49$ grid. All the other cases were simulated only using the LES approach.

Figure 14 shows the centerline velocity (normalized by the $u_\tau = (\tau_w/\rho)^{1/2}$, where ρ is the fluid density and τ_w is the shear stress at the wall) and τ_w itself. The abscissa is the normalized phase, $\phi = \omega t$. While the centerline velocity is in phase with the imposed pressure gradient, the wall stress is not. At high frequencies a sinusoidal shape is preserved, whereas for low frequencies the distribution of τ_w becomes very asymmetric. This is due to quasi-relaminarization of the flow during the acceleration phase, which is followed by a dramatic instability in the deceleration one. Good agreement with the DNS can be observed.

Figure 15 shows the mean velocity profiles at several phases. Good agreement is again observed between the LES and the DNS for the $Re_\delta = 100$ case. At this frequency a region of reversed flow is present, since the thickness of the oscillating Stokes layer reaches into the buffer layer and the flow reverses near the wall during the decelerating phase (without detachment of the boundary layer). For lower frequencies such reversal is not observed.

Different behaviors of the near-wall region as the frequency is decreased are evident in Fig. 15. A more dramatic illustration of the same phenomena can be seen in Fig. 16, in which contours of the turbulent kinetic energy are shown. At the highest frequency the inner and outer layers appear largely decoupled. A thickening of the inner layer can be observed at the end of the deceleration phase ($\phi/2\pi \simeq 0.5$), which, however, does not propagate far into the outer layer: by

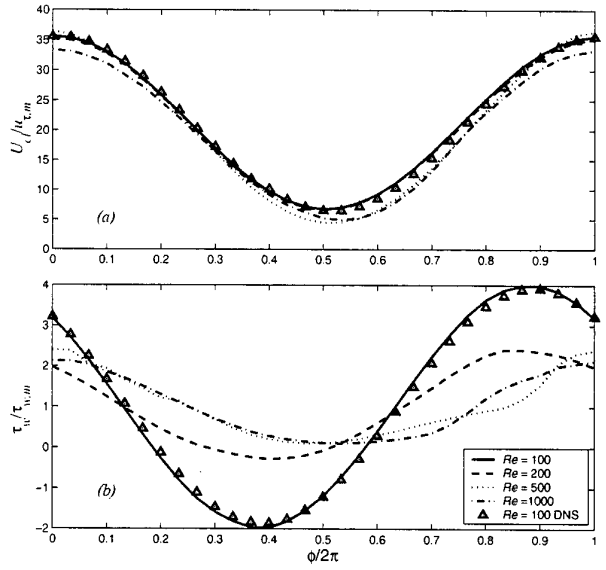


Fig. 14. (a) Centerline velocity and (b) wall stress in the oscillating channel.

$z/H \simeq 0.2$ the contours are nearly undisturbed. At lower frequencies, however, the inner layer has the time to adapt to the perturbation introduced by the pressure pulse; at the lowest frequencies in particular the flow can be observed to relaminarize, as indicated by the absence of turbulent kinetic energy. A shift of the more quiescent region of the flow from $\phi/2\pi \simeq 0.8$ towards $\phi/2\pi \simeq 0.5$ can also be observed, which can also be explained based on the increased time that the inner layer has to adapt to the outer-flow perturbation.

The turbulent eddy viscosity, Fig. 17, adjusts to the unsteady perturbation. It is not in phase with the local shear and vanishes as the flow relaminarize during the earlier portion of the accelerating phase. This is in agreement with results from the DNS concerning the evolution of the turbulent kinetic energy production term.

7 Conclusions

Large-eddy simulations have shown the ability to give accurate prediction of the turbulent flow in configurations in which the flow is not in equilibrium, albeit in fairly simple geometric configurations. This type of calculation can now be routinely carried out on desktop workstations, with reasonable throughput times. Parallel computers are required in more complex geometries, in flows in which large computational domains are necessary, and in cases in which long averaging times are required to obtain converged statistics.

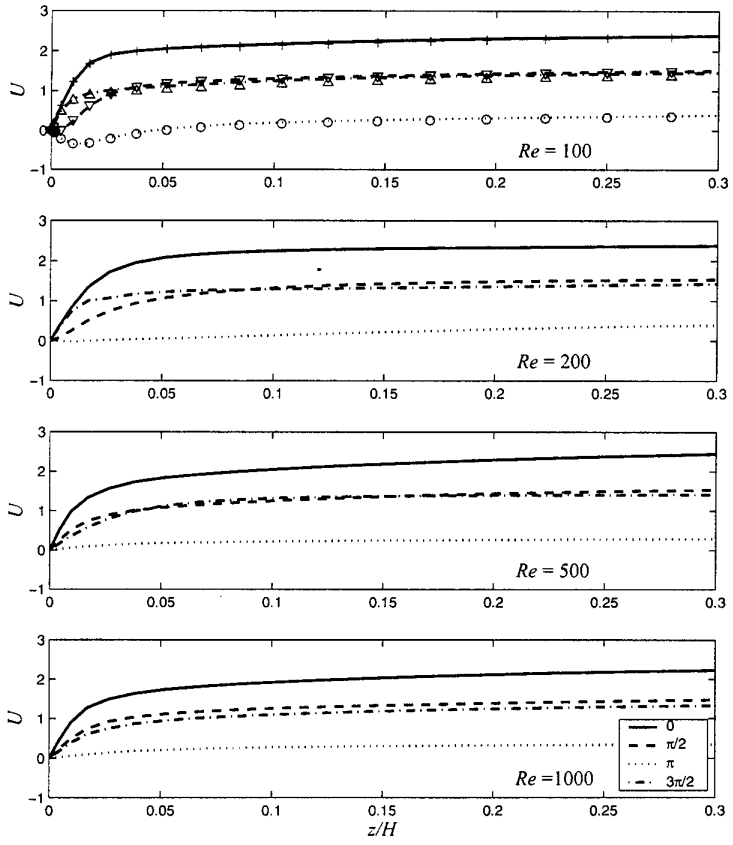


Fig. 15. Velocity profiles in the oscillating channel.

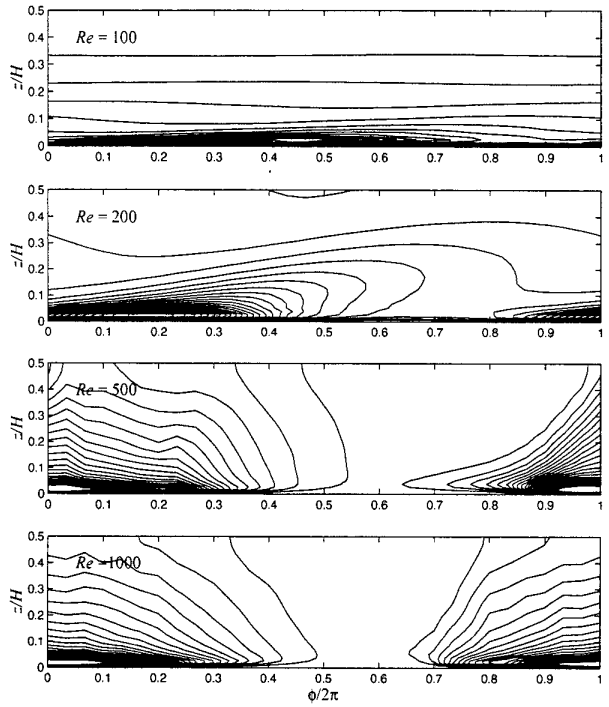


Fig. 16. Contours of the turbulent kinetic energy (normalized by the mean wall stress) in the oscillating channel. 26 equi-spaced contours between 0 and 12.5 are shown

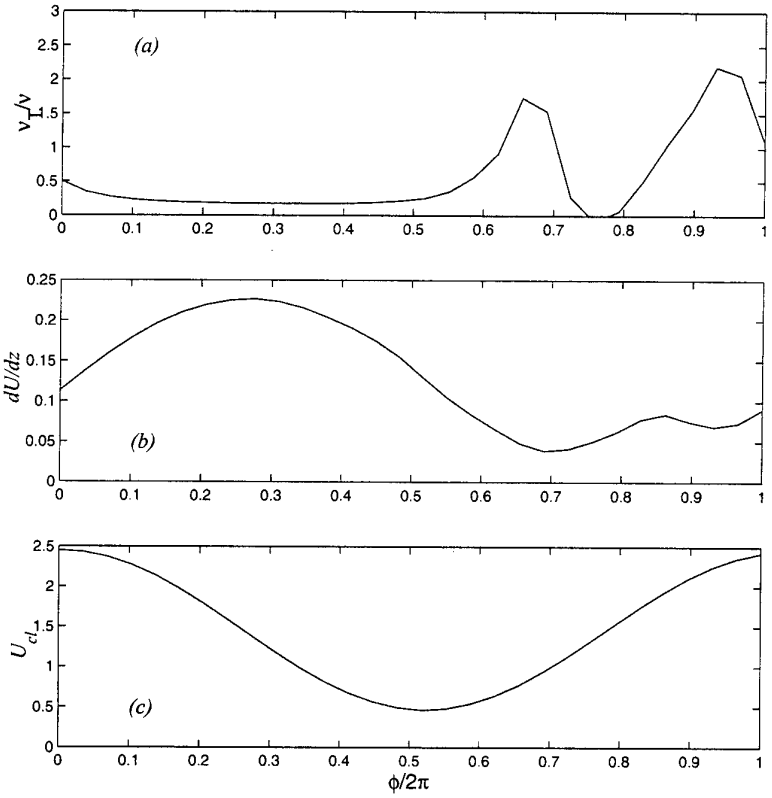


Fig. 17. (a) Phase-averaged eddy viscosity (normalized by the molecular viscosity) at $z/H = 0.0265$. (b) Phase-averaged dU/dz at $z^+ = 15$. (c) Phase-averaged mid-channel velocity. $Re_\delta = 100$.

The next stage in the development of this technique will involve the use of LES in more complex geometries. Challenges that need to be met to achieve this goal include the development of energy-conserving, high-order schemes in generalized coordinates or on unstructured meshes, and of accurate wall models to simulate the near-wall region without resolving in detail the inner-layer eddies. Combustion models, and SGS models for compressible applications are other areas in which additional research is required to exploit fully the potential of LES. Applications in complex geometries, especially those including combustion, multi-phase flows, or mean-flow unsteadiness, are not likely to be feasible on desktop workstations. Memory-intensive problems will also require parallel machines.

Researchers who use large-eddy simulations are typically end-users of the algorithmic improvements developed by mathematicians and computer scientists. A close collaboration between workers in these fields is, therefore, desirable in order to achieve some progress in the challenging area of turbulence prediction and control.

Acknowledgments

UP and EB acknowledge the support by the NASA Langley Research Center, under Grant No. NAG 1-1828, monitored by Dr. Craig L. Streett. AS acknowledges the support by the National Science Foundation under Grant OCE 99-10883, monitored by Dr. Stephen P. Meacham.

References

1. M. Lesieur and O. Metais. *Ann. Rev. Fluid Mech.* **28**, 45 (1995).
2. U. Piomelli. *Progress Aero. Sci.* **35**, 335 (1999).
3. C. Meneveau and J. Katz. *Annu. Rev. Fluid Mech.* **32**, 1 (2000).
4. M. Germano, U. Piomelli, P. Moin, and W. H. Cabot. *Phys. Fluids A* **3**, 1760 (1991).
5. C. Meneveau, T. S. Lund, and W. H. Cabot. *J. Fluid Mech.* **319**, 353 (1996).
6. G. L. Brown and A. Roshko *J. Fluid Mech.* **64**, 775 (1974).
7. A. Leonard *Adv. Geophys.* **18A**, 237 (1974).
8. J. Smagorinsky. *Mon. Weather Rev.* **91**, 99 (1963).
9. D. K. Lilly. In *Proc. IBM Scientific Computing Symposium on Environmental Sciences.* Yorktown Heights, N.Y., 195 (1967).
10. E. R. Van Driest. *J. Aero. Sci.* **23**, 1007 (1956).
11. U. Piomelli, T. A. Zang, C. G. Speziale, and M. Y. Hussaini. *Phys. Fluids A* **2**, 257 (1990).
12. F. Sarghini, U. Piomelli, and E. Balaras. *Phys. Fluids* **11**, 1607 (1999).
13. J. P. Chollet and M. Lesieur. *J. Atmo. Sci.* **38**, 2747 (1981).
14. J. P. Chollet. In *Turbulent Shears Flow IV*, edited by F. Durst and B. Launder, (Springer-Verlag, Heidelberg), 62 (1984).
15. O. Métais and M. Lesieur. *J. Fluid Mech.* **235**, 157 (1992).
16. F. Ducros, P. Comte, and M. Lesieur. *J. Fluid Mech.* **326**, 1 (1996).

17. Y. Morinishi, T. S. Lund, O. V. Vasilyev, and P. Moin. *J. Comput. Phys.* **143**, 90 (1998).
18. A. J. Chorin. *Math. Comput.* **22**, 745 (1969).
19. J. Kim and P. Moin. *J. Comput. Phys.* **59**, 308 (1985).
20. C. Canuto, M. Y. Hussaini, A. Quarteroni, and T. A. Zang. *Spectral methods in fluid dynamics* (Springer-Verlag, Heidelberg) (1988).
21. Y. Zang, R. L. Street, and J. Koseff. *Phys. Fluids A* **5**, 3186 (1993).
22. Y. Zang, R. L. Street, and J. Koseff. *J. Comput. Phys.* **114**, 18 (1994).
23. Y. Zang, R. L. Street, and J. Koseff. *J. Fluid Mech.* **305**, 47 (1995).
24. P. Beaudan, and P. Moin. *Report No. TF-62*, Dept. Mech. Eng., Stanford University, Stanford, CA 94305 (1994).
25. S. A. Jordan *J. Comput. Phys.* **148**, 322 (1999).
26. A. Kravchenko, P. Moin, and R. D. Moser. "Zonal embedded grids for numerical simulations of wall-bounded turbulent flows." *J. Comput. Phys.* **127**, 412 (1996).
27. D. K. Lilly."A proposed modification of the Germano subgrid-scale closure method." *Phys. Fluids A* **4**, 633 (1992).
28. K. E. Jansen. In *Ann. Res. Briefs-1996.* Center for Turbulence Research, NASA Ames/Stanford Univ., 225 (1996).
29. K. E. Jansen. In *Advances in DNS/LES*, edited by C. Liu and Z. Liu (Greyden Press, Columbus), 117 (1997).
30. D. Knight, G. Zhou, N. Okong'o, and V. Shukla. "Compressible large eddy simulation using unstructured grids." *AIAA Paper 98-0535* (1998).
31. U. Piomelli, E. Balaras, and A. Pascarelli. To appear, *J. of Turbulence*, (2000).
32. T. Sarpkaya. *J. Fluid Mech.*, **253**, 105 (1993).
33. G. Binder, S. Tardu and P. Vezin. ' *Proc. R. Soc. Lond.* A, **451**, 121 (1995).
34. T. A. Zang and M. Y. Hussaini. *Appl. Math. Comput.* **19**, 359 (1986).
35. U. Piomelli. *Phys. Fluids A* **5**, 1484 (1993).
36. M. Frigo and S. G. Johnson. In *ICASSP Conf. Proc.*, **3**, 1381 (1998).
37. R. D. Moser and P. Moin. *NASA TM-85074.*
38. P. S. Pacheco. *Parallel Programming with MPI*, (Morgan Kaufmann, San Francisco) (1997).

A Neural Network Based Tool for Semi-Automatic Code Transformation

V. Purnell, P. H. Corr and P. Milligan.

School of Computer Science,
The Queen's University of Belfast
Belfast BT7 1NN
N. IRELAND
p.corr@qub.ac.uk

Abstract. A neural network based tool has been developed to assist in the process of code transformation. The tool offers advice on appropriate transformations within a knowledge-driven, semi-automatic parallelisation environment. We have identified the essential characteristics of codes relevant to loop transformations. A Kohonen network is used to discover structure in the characterised codes thus revealing new knowledge that may be brought to bear on the mapping between codes and transformations or transformation sequences. A transform selector based on this process has been developed and successfully applied to the parallelisation of sequential codes.

1 Introduction

Over the past decade there has been a dramatic increase in the range of different multiprocessor systems available in the marketplace ranging from high-cost supercomputers, such as the Cray T3D, to low-cost workstation clusters. It is fair to say that the low-cost architectures have proven attractive to the majority of potential scientific and engineering users, particularly for applications that need to exploit the raw power now available. However, their appeal is often tempered by deficiencies in the attendant program development environments. Indeed, these deficiencies are evident in all multiprocessor development environments. It may be pejorative, but nonetheless accurate, to characterise the majority of typical users of multiprocessor systems as unskilled in the arts of parallelisation. Typically, they will be confirmed in the use of essentially sequential languages such as Fortran and have neither the time nor the desire to understand the intricacies of parallel development techniques or the peculiarities of target architectures in their endless search for enhanced performance. For such users there is a compelling need for the development of environments which minimise user involvement with such complications. In short, if novice users are to realise the potential of multiprocessor systems then much of the expert knowledge required to develop parallel code on multiprocessor architectures must be provided within the development environment.

The ideal solution is to provide a fully automated solution in which a user can simply input a sequential program to the system, be it migrated or newly developed, and receive as output an efficient parallel equivalent. Automatic parallelisation scores

high on expression, as programmers are able to use conventional languages, but is problematic in that it requires inherently complex issues such as data dependence analysis, parallel program design, data distribution and load balancing issues to be addressed. Existing parallelisation systems adopt a range of techniques in an effort to minimise or eliminate the complexity inherent in the fully automated approach. Almost invariably the burden of providing the necessary guidance and expertise lacking in the system falls back on the user. Indeed, existing systems may be classified by the extent to which user interaction is required in the process of code parallelisation. At one end of the spectrum is the purely language based approach in which the user is entirely responsible for determining how parallelism is to be achieved by annotating the code with appropriate compiler directives. At the other end of the spectrum is the goal of a fully automatic parallelisation environment, independent of application domain and requiring no user guidance. Between these extremes lie a number of environments which permit the user to interact with the system during the code development process. These environments may offer differing degrees of guidance and interaction in, for example, selecting appropriate program transformations or deciding on a particular data partitioning scheme.

Recently attention has focused on the use of knowledge bases and expert systems as a means of compensating for lack of user expertise. Such approaches have achieved a degree of success but to date have taken a rather narrow view of the field of knowledge engineering. Alternative methods of extracting and representing knowledge directly from the code itself have not been widely applied. Genetic programming techniques applied to program restructuring have been explored by Ryan et. al. [1] with notable success while neural networks have been variously applied to load balancing and data distribution [2, 3]. It is our belief that if a system is to be developed capable of offering quality strategic guidance for parallelisation then it must be underpinned with an appropriate knowledge model in which expert knowledge and information implicit in the code itself may be captured and synthesised within a coherent framework [4].

The research reported here complements and extends previous work in developing an integrated software development and migration environment for sequential programmers. The result of this work, KATT, (Knowledge Assisted Transformation Tools) has been reported elsewhere [5, 6, 7]. KATT began by employing expert systems only. As various neural network based tools are developed they have been integrated into the environment in line with the underlying knowledge model [8]. This paper reports the development of one of these neural-based components, a transform selector, and reports results obtained from its use with real codes.

2 The KATT Environment

Architecturally, KATT may be considered to consist of three main modules, namely:

- The input handler - responsible for converting input codes to an intermediate, language independent, graphical representation.
- The transformation module - which has access to a suite of correctness preserving graph manipulation routines used to restructure the graphical representation of the code produced by the input handler. The result is a new, functionally equivalent

version of the program that is more amenable to parallelisation, i.e., with dependencies removed or reduced.
- The output handler - responsible for generating actual parallel code, based on the modified graph and evaluating its performance on the target architecture.

KATT employs a source-to-source restructuring model. The explicit knowledge available within the environment is provided by two expert systems; one to aid the user in selecting appropriate code transformations, the other to advise on the best distribution of code and data on the target architecture.

Neural networks however offer a means of accessing an alternative source of knowledge relevant to parallelisation. By extracting domain knowledge implicit in the code itself neural networks can provide an alternative low-level, signal-based, view of the parallelisation problem in contrast to the high-level, symbolic view offered by expert systems. Combining both paradigms within a coherent knowledge model will improve the ability of KATT to offer strategic intelligent guidance to the user through access to a broader and deeper knowledge model.

3 Development of the Transform Selector

We use an SPMD model that concentrates on detecting and realising potential parallelism in computationally intensive sequential code loops. To do this requires that any loop carried dependencies, the principal inhibitors to parallelisation, can be detected and removed by a suitable transformation. What is required is a tool capable of taking an input code loop and recommending an appropriate transformation, or transformation sequence, that will reduce or eliminate any dependencies in the input code.

3.1 Choice of the Neural Paradigm

A number of network architectures were considered as potential solutions. One immediate problem that must be addressed is the nature of the data available to train the network. While there is an abundance of code loops available for input data there is no corresponding source of output data where suitable transformations, or transformation sequences, have been identified for each input code loop. In general, it requires an expert to specify the most appropriate transformations for a given input code. Casting the problem as one requiring supervised learning (e.g., a multilayer perceptron) would inevitably result in a network which encapsulated the opinion of a given expert. As agreement among experts is rare in the field of parallelisation the value of such a trained network would be questionable.

Selecting the "best" sequence of loop transformations can also be viewed as an optimisation problem. Here the problem is formulated as one of finding the optimum sequence of transformations that satisfy the dependence constraints of the code loop under consideration while minimising execution time (e.g., a Hopfield network or a Boltzmann machine). However, mapping the dependencies to network weights and interpreting the eventual solution makes this approach difficult from the representational point of view.

The Kohonen self-organising network was eventually identified as the most appropriate neural component for the transform selector. The principal reason for this choice is that the Kohonen network employs an unsupervised learning algorithm in which it is not necessary to know in advance the 'correct' output for a given input. Once trained the organised network topology reflects the statistical regularities of the input data. This information is useful for exploratory data analysis and visualisation of high dimensional data.

3.2 Data Sources

The training data used to develop the Kohonen network is derived from a suite of standard Fortran-77 benchmarking codes. The sources include the Livermore loops [9] and Dongarra's parallel loops [10]. Loops from these sources represent examples of real code; an important consideration if the eventual transform selector is to be capable of dealing with the intricacies of real input codes. A selection of Banerjee's loops [11] were also included. By comparison with the other sources, these are not 'real' codes but were included for their rich data dependence information. The eventual loop corpus contains 110 loops with a total of almost 16,000 dependencies.

3.3 Code Characterisation

Code characterisation is a necessary pre-processing stage in which a set of feature vectors is generated for each loop in the code set. It is important that the characterisation scheme employed should capture information influential in the selection of loop transformations, particularly information on the nature of any dependencies present in the loop. The characterisation scheme used encodes 12 features for each loop in a 20 component feature vector. Details of the characterisation scheme are shown in Table 1.

It is important that the characterisation scheme captures the complexities or real codes. In particular, the scheme must deal with symbolic dependencies where symbolic terms occur in loop bounds or array subscript expressions. Such symbolic terms impede dependence analysis yet empirical studies have shown that over 50% of array references and 95% of loop bounds in real programs contain non-linear, symbolic, expressions [10, 12]. Where precise dependence information is unobtainable the characterisation scheme encodes one of five possible reasons (Cat in Table 1).

A tool has been developed to automatically generate the set of feature vectors from Fortran-77 input code where each feature vector corresponds to a single loop carried dependence. As such, a given loop may give rise to a number of feature vectors, one for each dependence. The characterisation scheme also provides contextual information through a strength rating representing the proportion of each particular dependence type in a loop. This characteristic provides a measure of the complexity of the loop in terms of data dependencies.

While the capability to deal with real codes is essential it is also necessary to limit the complexity of the problem in order to limit the dimensionality of the feature vectors. To this end all loops in the loop corpus contain assignment statements only in the loop body, are normalised and either single or perfectly double nested.

Table 1. Characterisation Scheme

Feature	Mnemonic	Vector Components
Type of dependence – flow, anti, output or input	Type	3
Direction vector	Direction	2
Category when precise information is unobtainable	Cat	5
Flow dependence strength in loop	δ_f	1
Anti dependence strength in loop	δ_a	1
Output dependence strength in loop	δ_o	1
Input dependence strength in loop	δ_i	1
Lower bound of outer loop	LBI	1
Upper bound of outer loop	UBI	1
Lower bound of inner loop	LBJ	2
Upper bound of inner loop	UBJ	2

3.4 Network Training

The feature vectors generated from the Fortran-77 loop corpus are used as input to train the network. Since the Kohonen network uses an unsupervised learning algorithm it is not necessary to know *a priori* which transformations are appropriate for a given loop. During training the Kohonen layer undergoes a self-organising process in which a two-dimensional map is produced representing the higher dimensional input space. An essential feature of the map produced is that it preserves the topology of the input space in that inputs which are 'close together' in input space are mapped to points 'close together' on the Kohonen layer. In effect, points on the Kohonen map represent prototypes, or cluster centres, for the features vectors used during training. Thus, a feature vector input to the trained network will be represented by a single prototype on the mapping layer.

It is a fundamental assumption in this work that input codes which map to the same prototype on the Kohonen mapping layer, and are therefore 'close together' in input space, are amenable to the same transformation. The essence of the transformation framework then is to establish which transformation(s) are most appropriate for each prototype represented in the Kohonen layer.

3.5 Labelling the Map

A reverse engineering approach was adopted in labelling each prototype with the corresponding transformation(s). Firstly, five of the most useful and widely used transformations were identified. The transformations chosen were:

- D Loop Distribution
- I Loop Interchange
- S Loop Skewing
- E Scalar Expansion and
- R Statement Reordering

For each transformation a data set of representative codes was established where it is known that the code is amenable to the transformation. Each code is then characterised and the resultant feature vector presented to the trained network. In each case one prototype in the mapping layer is identified which best represents the input and that prototype labelled with the associated transformation. The result is a transformation framework; a labelled map that may be used to suggest transformations that should be applied to an input code in an attempt to reduce or remove loop carried dependencies.

4 Results

As an illustrative example of how the transformation framework operates, consider the following loop:

```
DO I = 4,200
  A(I) = B(I) + C(I)
  B(I+2) = A(I-1) + A(I-3) + C(I-1)
  A(I+1) = B(2*I+3) + 1
CONTINUE
```

This loop is input to the characterisation tool and four feature vectors, X_1 to X_4, returned - one for each loop carried dependency identified. Each feature vector is presented in turn to the Kohonen network and the position of the corresponding prototypes noted. These are shown at grid co-ordinates (7,11), (2,8), (4,1) and (5,11) on the labelled map in figure 1.

Notice that the inputs do not all fall directly onto labelled prototypes. Table 2 shows the distances from inputs to the nearest labelled prototypes.

The best choice of transformation appears to be loop skewing because it is the closest labelled prototype to a number of inputs and should remove most dependencies. However loop skewing is illegal, i.e., it does not apply to 1D loops. The next best transformation is loop distribution, which is legal. Applying loop distribution results in two separate loops with fewer dependencies in each.

```
DO I = 4,200
  S1: A(I) = B(I) + C(I)
  S2: B(I+2) = A(I-1) + A(I-3) + C(I-1)
CONTINUE

DO I = 4,200
  S3: A(I+1) = B(2*I+3) + 1
CONTINUE
```

A second iteration of the procedure is now performed. After loop distribution the second loop is parallel, a fact detected by the characterisation tool resulting in no

further transformations being performed on this loop. The first loop has now dependencies at S1 δ_f S2 by A, S2 δ_f S1 by B, and S2 δ_i S2 with A. Repeating the same process as before results in two vectors, Y_1 and Y_2, shown plotted at their corresponding prototypes in figure 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1		IE R			S	D		S								
2				D		Y_1	D	X_3								
3								S							S	
4	X_4					D S					D		D		S	DI
5									D		S X_2	Y_2				
6										D					D	
7				D ER							X_1					
8							E									
9																
10	IS															
11																
12																

Fig. 1. Labeled map showing inputs X_1, X_2, X_3 and X_4 from the first iteration and Y_1, Y_2 from the second iteration

Table 2. Distance from input vectors to nearest labelled prototypes

Input	Nearest Prototype(s)	Transform	Distance
X_1	(6,10)	Distribution	1.414
X_2	(1,8), (2,7), (3,8)	Skewing Distribution	1.000 1.000
X_3	(1,2)	Interchange or Expansion or Reordering	3.162
X_4	(5,11)	Skewing	0.000

It can be seen from the map that Y_1 is closest to a prototype representing loop distribution. Input Y_2 is nearest loop skewing but very near two prototypes also representing loop distribution. Therefore, loop distribution is the recommended transformation. Application of this transformation gives the following loops:

```
DO I = 4,200
  S1: A(I) = B(I) + C(I)
CONTINUE

DO I = 4,200
  S2: B(I+2) = A(I-1) + A(I-3) + C(I-1)
CONTINUE
```

Both these loops may now be executed in parallel; hence there is no requirement for further transformation.

This example demonstrates an obvious example of when to stop applying transformations. In other cases, determining when to stop iteratively applying the process is not so straightforward. The following stopping criteria are used in order of preference, namely:

- all the dependencies are removed,
- the user decides to apply no further transformations,
- applying the recommended transformation does not reduce the number of dependencies – in this case the user may decide to proceed with the code after transformation or roll back to a previous state, and
- the distance on the map between inputs and labelled prototypes is large – typically for any distance greater than 3 the network cannot confidently make a recommendation.

The transformation framework presented here has been successful in recommending appropriate transformations for over 90% of the codes on which it has been tested.

5 Conclusions

A hitherto untried neural network based technique offering strategic guidance on the selection of appropriate transformation sequences has been developed and implemented. The transform selector developed has been tested and has produced excellent results. Research is continuing into the possibility and desirability of extending the existing tool. Extensions may include using more than the five transformations originally chosen or labelling more prototypes by using more labelling codes.

The next stage of the work is to integrate the neural-based transform selector and the existing expert system to form a hybrid transformation framework within the KATT environment. As a first stage in this integration a comparative study has been undertaken to compare the advice given by both systems. An essential part of the integration will involve extending the expert system with the new knowledge gained as a result of analysing the Kohonen network at the heart of the new transformation framework.

References

1 C. Ryan: Automatic Re-engineering of Software Using Genetic Programming. Kluwer Academic Publishers, 1999.

2 H. Heiss and M. Dormanns: Partitioning and mapping of Parallel Programs by Self-Organisation. Concurrency: Practice and Experience, Vol. 8(9), 1996, pp.685-706.

3 B. McCollum: The Application of AI Methodologies to Code Partitioning and Distribution. Internal Report, School of Computer Science, The Queen's University of Belfast, N. Ireland, 1999.

4 V. Purnell, P. H. Corr and P. Milligan: A Novel Approach to Loop Characterization. IEEE Computer Society Press, 1997, pp.272-277, ISBN 0-8186-8215-9.

5 P. Milligan, P. P. Sage, P. J. P. McMullan and P. H. Corr: A Knowledge Based Approach to Parallel Software Engineering. In: Software Engineering for Parallel and Distributed Systems. Chapman and Hall, 1996, pp.297-302, ISBN 0-412-75640-0.

6 P. J. P. McMullan, P. Milligan, P. P. Sage and P. H. Corr: A Knowledge Based Approach to the Parallelisation, Generation and Evaluation of Code for Execution on Parallel Architectures. IEEE Computer Society Press, 1997, pp.58-63, ISBN 0-8186-7703-1.

7 B. McCollum, V. Purnell, P. H. Corr and P. Milligan: The Improvement of a Software Design Methodology by Encapsulating Knowledge from Code. IEEE Computer Society Press, 1998, pp.913-917, ISSN 1089-6503.

8 B. McCollum, P. Milligan and P. H. Corr: The Structure and Exploitation of Available Knowledge for Enhanced Data Distribution in a Parallel Environment. In: N. E. Mastorakis (ed): Software and Hardware Engineering for the 21st Century. World Scientific and Engineering Society Press, 1999, pp.139-145, ISBN 960-8052-06-8.

9 F. McMahon: The Livermore FORTRAN Kernals: A Computer Test of Numerical Performance Range. TR UCRL-55745, 1986.

10 J. Dongarra: A Test Suite for Parallelising Compilers: Description and Example Results. Parallel Computing, Vol. 17, 1991, pp.1247-1255.

11 U. Banerjee: Loop Transformations for Restructuring Compilers. Macmillan College Publishing Company, 1992.

12 S. Zhiyu: An Empirical Study of Fortran Programs for Parallelizing Compilers. Technical report 983, Center for Supercomputing Research and Development.

Multiple Device Implementation of WMPI[1]

Hernâni Pedroso and João Gabriel Silva

Dependable Systems Group/CISUC
Dept. Engenharia Informática – Univ. Coimbra
Portugal
{hernani,jgabriel}@dei.uc.pt

Abstract. WMPI is an implementation of the Message Passing Interface (MPI) standard for Win32 platforms. It was the first implementation and is one of the most used worldwide for this operating system's family. In this paper, we describe a new version of WMPI (1.5), which can use several communication devices simultaneously in the same computation. This new version of WMPI can also use more than one network interface card on each machine to communicate with the others. The new library is also thread safe, which enables to better use the processor by using more than one thread per process. The possibility of passing different arguments to each process was also introduced. The communication devices are independent from the core library and can be developed by anyone; this will enable the product vendors to easily make a WMPI version running on their communication medium or protocol.

1 Introduction

The computational capabilities of Personal Computers (PCs) had a tremendous growth in the last few years (and they seem to keep this pace for some time). The cost/performance ratio of this type of machines is becoming very appealing when compared with traditional workstations or MPPs. Following the increase of capabilities in computation, the interconnection between processors is also getting better. PC boxes with 2 or 4 processors are now common and the number of processes tends to increase up to 32 in the near future. This represents a high bandwidth with low latency communication between processing elements, since they communicate through shared memory. For larger clusters, new network technologies are emerging, which present a much better performance than in the past years. All these factors are leading the PC clusters to a very widely used platform for cost effective high performance computing.

WMPI (Windows Message Passing Interface) [9,12] was the first full implementation of the MPI standard for Windows operating systems. It was originally based on MPICH [6] and ever since has received several optimizations to improve its

[1] This work was partially supported by the Portuguese Ministry of Science and Technology through the R&D Unit 326/94 (CISUC) and the project PRAXIS XXI 2/2.1/TIT/1625/95 named ParQuantum.

performance. Results presented in [1] show that WMPI is the fastest implementation freely available for Win32 platforms and competes head to head with commercial products. The work presented here is one more step to improve the usability of the library and is a result of several requests from users worldwide.

2 Motivation

One of the drawbacks of constructing a PC cluster for high performance computing was the lack of interconnection networks that could present high bandwidth and low latency between nodes. Most of the PC clusters were using TCP over Ethernet and Fast Ethernet networks, which are not optimized for message exchange performance but to reliability and cost. As the computational power of the PCs grew the network became the bottleneck of the cluster. Aware of this fact, the hardware vendors have started to create new technologies that improve the message passing performance between the computer nodes. VIA [11] is the most recent effort, which is presenting a wide acceptance, although Myrinet [2], Gigabit Ethernet and SCI [7,8] are also available. The MPI library must be able to follow these improvements of the underlying systems. It is thus necessary to create specific devices for each technology in order to use the maximum performance that it can offer.

Some clusters may have a heterogeneous communication system. A cluster may have a small set of computers that are interconnected with a high performance network, but more expensive. This situation may normally occur with the growing of the number of machines along with time and the acquisition of new technology. Sometimes also happens that some machines are added momentarily to boost a computation by increasing the number of processors. These new machines generally do not have the same communication medium has the existing cluster.

To improve the communication speed, some PCs have more than one network interface card (NIC). Each of these cards has a different network address. The previous version of WMPI could not use more than one address per machine. This new version allows the user to choose which NIC (or network address) should be used to communicate with each other computer.

3 Implementation Difficulties

As is extensively presented in [9], WMPI has the same layered implementation as MPICH. The upper layer implements all the MPI functions and uses an abstract device interface, the ADI [4], to access to more hardware dependent communication subsystems. Depending on the environment the latter can be a native subsystem or another message passing system. The WMPI implementation uses the ADI channel interface implementation [5] over p4 [3], a third generation parallel programming library.

Although the structure inherited from MPICH is excellent for portability and performance, its support for simultaneous multiple device computation is almost absent. The code present in the MPICH ADI2 implementation to work with simultaneous devices is not used in the current MPICH version. Moreover, it seems that the code is not complete. From the analysis of the existing code, it was verified that MPICH uses a very restricted structure where the user cannot really adapt the computation to its cluster. Also the ADI channel layer does a lot of polling over the all devices involved in the computation. Although this polling might improve the performance of the computation, with the increase number of devices it can produce the opposite result.

The ADI delegates all the responsibility of process management on the devices. Within the former version of WMPI (version 1.3), the p4 library is responsible for creating the processes, manage them and their communications and to kill all of them at the end of the computation. This makes the devices much more complex and obligates them to do work beyond the simple message exchange. Moreover, the fact that the lowest layers manage all the processes means that the coexistence of more than one independent device in the same computation is very difficult due to the need of co-operation and awareness between the devices.

4 Multiple Device Implementation

The new WMPI implementation (version 1.5) has a lot of differences when compared with the previous one. The old architecture was too restrictive and not flexible, so we had to create a new one. The process management work is now done, not in the devices but in an upper layer. This helps to reduce the complexity of the devices and permits a considerable independence between the devices and the WMPI core. However, the act of creating processes is device dependent because when creating a remote process is necessary to use some kind of communication medium to access to the remote machine. Hence the devices are responsible for creating the processes but, in opposition to the previous version, the devices are just a medium to create the processes, an upper layer conducts all the management.

Devices are now independent DLLs that are loaded at the startup of each process according to the cluster configuration. This approach introduces some problems for users since they have to manage more DLLs. However, this small drawback cannot supersede the incontestable advantages:

- New devices can be used without having to get a new version of WMPI. In fact, this reduces the DLLs management problem, because only one WMPI DLL is used for every device.
- The user can now test several device implementations (even for the same communication medium) to find which one has the best performance on her/his cluster/application.
- The development of new devices is totally independent from the WMPI core code. Having separated DLLs also helps the debugging of the new devices.

- Devices can be developed by the communication medium vendors. This will allow WMPI to be available for those communication mediums faster and probably better (since they are developed with total knowledge of the underlying medium).
- When a new version of WMPI is released, the users just have to update the WMPI DLL and can keep using their communication devices.

Since there are several devices through which the machines of the cluster can communicate, the user is responsible for defining how they communicate during the WMPI computation. The user has to create a cluster configuration file. Figure 2 presents an example of such a file. Each machine is identified by its Windows machine identification. For each machine, the user has to indicate which devices the processes running on that machine can communicate with and the identification of the machine using that device. This allows the user to specify more than one device, of the same type, in the same machine (with different addresses). It also solves the problem of identification of the machines in the Process Group file (PG) when a machine has more than one device.

With this configuration file, the user can also define communication paths and cluster communication structures. For example, the configuration of the Figure 2 represents the cluster of Figure 1, where `machine1` and `machine2` have a NIC to communicate with each other and a NIC to communicate with the other machines. This way, `machine1` and `machine2` have a direct path between them; this avoids the collisions with the traffic to the machine3. This type of configuration improves the communication performance between the two nodes; this is becoming more common as the price of the network cards lowers.

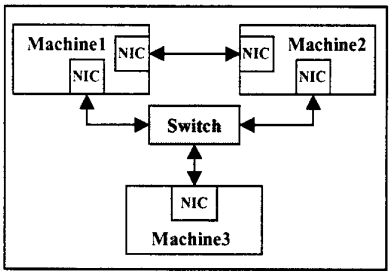


Fig. 1. Cluster defined by the configuration file.

```
/Machines
machine1
tcp 193.145.134.21
other 193.145.134.22
shmem machine1
machine2
tcp 193.145.134.11
tcp:1 193.145.134.12
shmem machine2
machine3
tcp 193.145.134.31
shmem machine3
/Connections
default internal shmem
default external tcp
machine1 tcp:1 machine2 tcp:1
```

Fig. 2. Example of a Cluster Configuration File

The definition of the devices used to communicate between the several machines is placed after the definition of all the machines on the cluster. The user has the possibility of specifying the default device for inter-process communication whether they run within the same machine (internal) or when in different ones (external). The user can also specify special connections, which do not follow the default. In the

above example, `machine1` uses the second IP address to communicate with the `machine2`, which also uses its second IP address.

The PG file has some changes to. In the previous version the machine where the first process (the Big Master) was started had to be specified as *Local*. Also the other machines had to be specified using their IP address. The new PG file (Figure 3) uses the Windows machine name to specify which machine is used to run the process.

```
Machine1 1 c:\wmpi\test.exe 1 1
Machine1 2 c:\wmpi\test2.exe 1 2
Machine2 1 c:\wmpi\test.exe 2 1
Machine3 2 c:\wmpi\test2.exe 3 1
```

Fig. 3. Example of a Process Group file.

In the former version of WMPI even the arguments that were passed to the Big Master were not passed along to the Worker processes. The user had to pass the information to the Workers through messages. This new version avoids all these problems by allowing the user to specify different arguments for each processes of the computation in the PG file. The user can also give different arguments to different processes on the same machine (whether they use the same executable or not).

Devices for Shared Memory and TCP/IP communication are being implemented. The results gathered this far show that the new architecture fulfills the requirements. Performance results of these new devices will be presented in the final paper version.

5 Data Visualization and GUI Applications

Libraries for parallel computation are a valuable aid to execute computing demanding algorithms. They allow the user to get its results much faster. However if the results are not presented in an intuitive way, it may be very difficult to interpret them. Most of the time it is much easier to understand the information in a graphical representation than in a matrix with several thousands of values.

It is normally considered that the presentation of the results is out of the scope of the parallel libraries and few might have any API function to produce some graphical outcome of the results. The users have to rely on the functionality provided by the operating system they are working on or to use some external libraries.

The users of the Windows operating system expect to have graphical user interfaces (GUI) to control their applications and to visualize the results in a graphical mode. The WMPI does contain any function to produce graphical results, but can be easily used in conjunction with the Microsoft Foundation Classes (MFC) [13]. This way is possible to produce GUI applications that calculate the results in parallel and present them in a graphical manner.

The WMPI does not require starting at the beginning of the process and does not use the contents of the `argc` and `argv` parameters (in fact they can be passed as `NULL`

into the `MPI_Init` call). Hence it can be started by any GUI application whenever its necessary to make the computation intensive part.

Figure 4 presents an application that calculates the Mandelbrot set [14] and displays it. The WMPI is started when the application initializes and remains running until the user ends the application. The WMPI could be started every time that a calculation is requested, however the penalty time for starting the environment is quite high for such small applications.

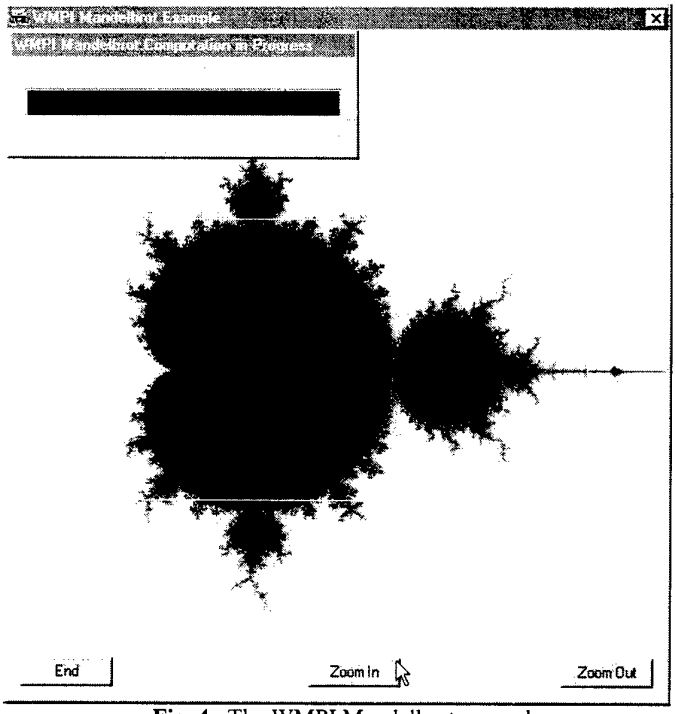


Fig. 4. The WMPI Mandelbrot example.

This application is constituted by tow separated executables. One interacts with the user, presents the results and distributes the data to be computed. The second executable is a simple C program without any graphical interface and is used just to compute the data. In fact it is a Master/Worker application. The Master spreads the data around and gathers the results from the Workers.

In this example, the user can select a region of the window to zoom in and see the Mandelbrot set with more detail. The Master (GUI executable) splits the computation in small jobs and scatter them by the Workers. After receiving the results, it presents them to the user. The WMPI application is completely controlled by a graphical user interface.

6 Multiple Threads in a WMPI Process

One of the most expected features of the new WMPI version was thread safety. While creating the new structure it was taken as a demand. The usage of multiple devices simultaneously implied that the library had to be partially thread safe, since each device can have one or more threads that access concurrently to the WMPI internal structures. A deep study was made to identify every structure that could be corrupted by simultaneous accesses and a protection mechanism was introduced. The synchronization had to be as light as possible to reduce the impact in the latency of the library.

The new WMPI library offers the highest level of thread safeness (`MPI_THREAD_MULTIPLE`) described in the MPI 2.0 standard (Chapter 8 – Section 7). This thread safety level allows the users to make simultaneous calls to MPI functions.

By using multiple threads in the same process, the user can better exploit the CPU usage by reducing the waiting times. For example the Mandelbrot example uses two threads in the GUI application. One acts as a Worker and the other as a Master. This way is possible to receive a result from any Worker and send it another job without having to wait for the Master to calculate its job until the end.

Master/Worker applications can easily take advantage of multiple threads, however many applications can be programmed to better use the CPU capacity by using more than one thread per process.

7 Performance Results

Although the performance of the WMPI library was among the best existing libraries for Windows NT systems [1], during the evolution of the library new strategies were introduced to reduce the communication latency. The new architecture reduces the synchronization and uses functionality specific from the operating system. Since the former version of WMPI was based in a portable implementation, it used generic functionality. This version does not intend to be portable, hence specific operating system functions were welcome whenever they improved the latency.

Table 1 shows a time comparison of the one-way latency between two processes running on a dual PIII 550Mhz through shared memory communication. It can be seen that the latency was reduced from almost 29 microseconds to approximately 17 microseconds in a zero length message. The Figure 5 shows the percentage of improvement that the new version has when compared with the former one. Up to 8 Kbytes, the new version is 40 to 50% better (half of the latency). The improvement is reduced for larger messages because the time spent in memory copies is much higher than the latency time and the later is diluted in the outcome.

Table 1. One-way latency for the shared memory device in a dual PIII 550Mhz (time in microseconds).

Size	WMPI 1.3	WMPI 1.5
0	28.95	16.87
1	30.75	17.8
2	30.62	17.48
4	30.77	17.71
8	31.53	17.28
16	31.25	17.84
32	32.08	17.71
64	33.6	18.82
128	35.35	19.11
256	38.12	20.78
512	40.4	22.49
1K	51.65	25.98
2K	64.33	32.17
4K	97.87	48.13
8K	165.57	86.77
16K	165.2	141.24
32K	285.55	258.76
65K	532.72	503.53
128K	1123.19	1070.61

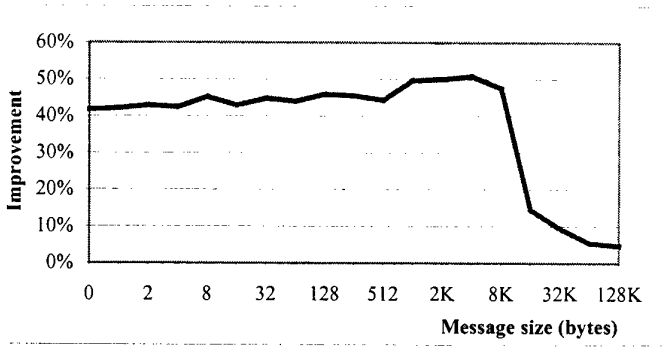


Fig. 5. Improvement in latency between the two WMPI version (1.3 and 1.5)

The next experiment uses the TCP device to communicate between two processes running on two dual Pentium-Pro machines connected through a Fast Ethernet network. The Figure 6 shows that WMPI uses practically all the available bandwidth for larger messages. For smaller messages, the result is conditioned by the latency that the library introduces in the communication. But when the message size starts to increase and the latency introduced gets a much smaller weight in the final result than the time to send the message through the network, the library is able to use practically all the bandwidth that the system offers. The improvement starts at 256 bytes and at 8Kbytes already uses 93% of the available bandwidth.

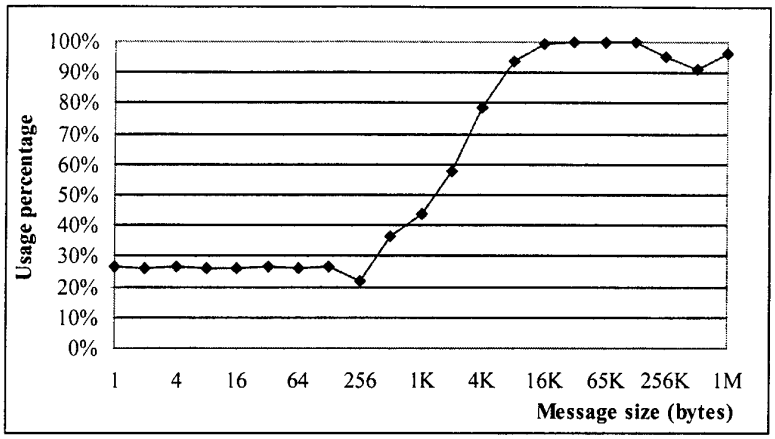


Fig. 6. Bandwidth utilization in a Fast Ethernet network.

Figure 7 presents the speed up obtained using two different applications. The Mandelbrot set calculation and the PI calculation examples. The two applications have quite different results using WMPI. This is due to the fact that it is impossible to make the construction and presentation of the bitmap figure of the Mandelbrot set in parallel. Since this action is centralized in the GUI application, it reduces the parallelism of the application and hence the speed up.

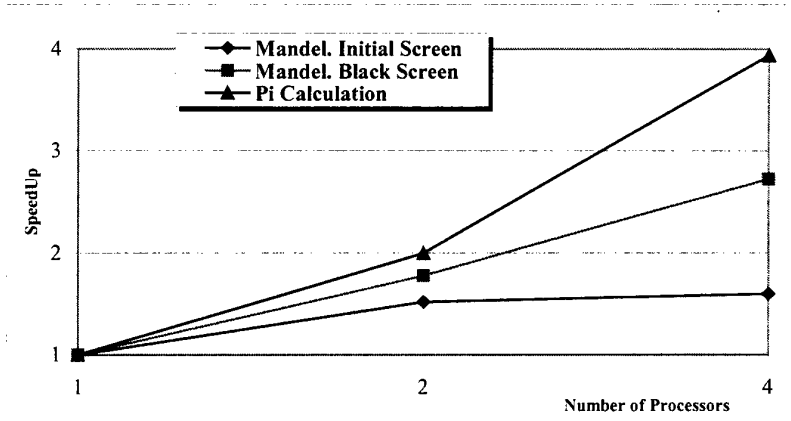


Fig. 7 Speedup obtained using the PI calculation and the Mandelbrot set calculation.

Two measurements were gathered for the Mandelbrot application. One contains the speed for the calculation of the initial screen (Figure 4), the other were gathered when processes have to calculate a completely black region. In the second experiment the results are better because the black region is more computation intensive (the values

are tested until they are considered inside the set) and more time is spent in computing (which reduces the weight of the presentation in the overall time).

The PI calculation shows that WMPI is able to offer a practically linear speedup when the application permits.

8 Conclusions

The previous implementation of WMPI forced the user to have a homogeneous communication system on the cluster. The changes introduced in WMPI allow the users to adjust the computation to fully use the capabilities of their cluster.

The way that the user configures the parallel execution through the PG file is now much more complete and versatile. The name of the machines is unambiguous and arguments can be passed to every process.

The library is now thread safe, which allows the user to fully exploit the CPU usage per process by reducing the waiting times. The usage of multiple thread also aids to develop applications where processes running on one machine have different functionality, in these cases instead of using processes the user can use threads and reduce the penalty time for context switching.

Since the communication devices are now independent of the WMPI core, it is possible to implement devices for the new technologies that are emerging without having to change the core code. This also avoids the release of several versions of WMPI, one for each communication subsystem.

This new architecture allows the communication medium vendors to implement the device drivers for their products. The knowledge they have on their product allows them to build a device with the best performance. We hope that this strategy also enables the devices to be available to the users right after the release of a new technology. Since it can be developed by anyone, even by WMPI users.

The new structure of the WMPI core is also driven by the need of make deep changes for the MPI-2 Dynamic Process Creation. A full MPI-2 implementation (WMPI 2.0) is under development and will be released in the near future.

References

1. Baker, M: MPI on NT: The Current Status and Performance of the Available Environments. Proc. of 5th European PVM/MPI User's Group Meeting, pp.63-73 (September 1998).
2. Boden, N., Cohen, D., Felderman, R., Kulawik, A., Seitz, C., Seizovic, J. and Su, W.: Myrinet: A Gigabit-per-second Local Area Network. IEEE Micro, pp. 29-36 (February 1996).
3. Butler, R. and Lusk, E.: Monitors, messages and clusters: The p4 parallel programming system. Parallel Computing, 20:547-564 (April 1994).

4. Gropp, W. and Lusk, E.: An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Preprint MCS-P342-1193, Argonne National Laboratory (1994).
5. Gropp, W. and Lusk, E.: MPICH working note: Creating a new MPICH device using the channel interface. Technical Report ANL/MCS-TM-213, Argonne National Laboratory (1995).
6. Gropp, W., Lusk, E., Doss, N. and Skejellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Pre-print MCS-P567-0296 (July 1996).
7. Gustvason, D. and Li, Q.: The Scalable Coherent Interface (SCI). IEEE Communications Magazine, pp. 52-63 (August 1996).
8. IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface (SCI). (August 1993).
9. Marinho, J. and Silva, J.G.: WMPI – Message Passing Interface for Win32 Clusters. Proc. of 5th European PVM/MPI User's Group Meeting, pp.113-120 (September 1998).
10. Message Passing Interface Forum: MPI: A message-passing interface standard. International Journal of Supercomputer Applications, 8(3/4):165-414 (1994).
11. Virtual Interface Architecture Interface Specification: Version 1.0 http://www.viarch.org (December 1997).
12. WMPI Homepage – http://dsg.dei.uc.pt/wmpi
13. Kruglinski, D., Shepherd, G. and Wingo, S.: Programming Microsoft Visual C++, Fifth Edition. Microsoft Press, ISBN 1-57231-857-0, 1998
14. Mandelbrot, B.: Fractal Geometry of Nature. W. H. Freeman, ISBN 07-167-11869, 1998

Optimization with Parallel Computing

Sourav Kundu

Emerging Technologies Research Laboratory (ETRL), Interlogic Japan Inc.
Pure Toranomon Bldg. 2F. 3-16-7 Toranomon, Minato-ku,
Tokyo 105-0001, Japan
sourav@ilogic.net

Abstract - Complexity Engineering deals with harnessing the power of Cellular Automata (CA) like simple models to solve real life difficult and complex engineering problems, dealing with systems that have very simple components that collectively exhibit complex behaviors. Cellular Automata (CA) are examples of dynamical systems which may instead exhibit "self organizing" behavior with increasing time. CAs are commonly used in modeling modular systems. An important aspect of modularity in engineering systems is the abstraction it makes possible. Once the construction of a particular module has been completed, the module can be treated as a single object, and only its behavior need be considered, wherever the module appears. One such application of modularity is a described in this paper where a structural plate is considered as composed of smaller "structural modules" which are considered as cells in a lattice of sites in a CA and have discrete values updated in discrete time steps according to local rules. These local rules are generally fixed in a CA, but we consider these rules as evolvable. To evolve the *local rules*, we use the Genetic Algorithm (GA) model. Though the application described here is simple, but it will serve to demonstrate that the GA *can* discover CA rules that give rise to emergent computational strategies by self-organization, to exhibit globally coordinated tasks in optimization by simple local interactions only.

1 Introduction

In conventional engineering, systems are built to achieve very specific goals by exhibiting specific "global" behavior. Even the behavior of each of their component "local" parts are strictly designed and have only specific reasons for their existence. The overall behavior of these systems must be simple enough so that complete prediction and often also analysis, is possible. Thus for example motion in conventional mechanical engineering devices is usually constrained to be periodic. Of course more complex behavior could be realized or expected from the basic components of a mechanical engineering device but principles necessary to make use of such behaviors and theory necessary to

analyze such behaviors, is not yet known. On the contrary, nature provides many examples of systems whose basic components are simple, but whose overall behavior is extremely complex. Mathematical models such as Cellular Automata (CA) capture the essential features of such "bottom-up" complex systems. Complexity Engineering deals with harnessing the power of CA like simple models to solve real life difficult engineering problems, dealing with systems that have very simple components that collectively exhibit complex behaviors. Generally speaking, discrete dynamical systems that follow the second law of thermodynamics evolve with time to maximal entropy and complete disorder. But Cellular Automata are examples of dynamical systems which may instead exhibit "self organizing" behavior with increasing time. Even starting from complete disorder, their irreversible evolution can spontaneously generate ordered structures. Sometimes even decrease of entropy with time is noticed as a result of self-organization. This paper introduces the idea of using genetic (GA based) learning of a Cellular Automata (CA) that takes a disordered structural layout to an ordered one, satisfying several conflicting design criteria and finally producing an optimal or acceptable structure/design. To evolve the Cellular Automata (its rules) we use a Genetic Algorithm (GA) which is a widely accepted computational framework for evolution. The GA encodes the CA rules and evolves them progressively with time to more efficient ones (culling the less efficient ones in the evolution process). Detail computer simulation studies have been performed and results are presented in this paper. Though the application described here is simple but it will serve to demonstrate that the GAs can "discover" CAs that give rise to emergent computational strategies and exhibit global coordination tasks in structural optimization of complex engineering systems by simple local interactions only.

2 Self Organization

The second law of thermodynamics implies that isolated microscopically reversible physical systems tend with time to states of

maximal entropy and maximal "disorder". However "dissipative" systems involving microscopic irreversibility may evolve from "disordered" to more "ordered" states. This phenomenon of evolving from "disordered" to more "ordered" states can be seen in dynamically stable systems like Cellular Automata (CA)[1]. An elementary CA is a single array of "cells" capable of transforming themselves from one discrete "state" to some other. A certain definite set of interaction rules, governing how cells change their states in accordance to the value of the state of the neighboring cells, is given to each cell. With simple initial configurations, a CA either tend to homogeneous states or generate self-similar patterns with fractal dimensions $\cong 1.59$ or $\cong 1.69$. With random initial configurations, the irreversible character of the cellular automaton evolution leads to several self-organization phenomena. Statistical properties of the structure generated are found to lie in two universality classes, independent of the details of the initial state or the CA rules. This paper shows how the CA model can be used for its self-organizing capabilities for design optimization of structural plates and shells. The design goal is to find a minimum weight structure. Elements of this structure "adapts" to a minimum weight design, using the rules of the CA to transform its states (cell thickness). Rules are then subjected to genetic evolution by using a GA. A complex system is simulated with the interacting structural sub-elements being encoded as the *cells* of the CA. Sequential gathering of information with the interaction of structural sub-elements progressively modifies the different elements (variables) of the structure and the overall system evolves with time. This means that the rules of the CA (used for interaction between the structural sub-elements), are progressively modified by the genetic recombination (crossover and mutation), as these rules are directly encoded in the genotype string of the GA. Complex adaptive systems theory has nexus with structural optimization, both large and small scale, because both exhibit global behavior as a result of local action-reaction patterns, but this has not so far been exhaustively studied or experimented with. However, there has been some preliminary research done in the area of evolution of simple CA models [2].

3 Structural optimization by evolving a cellular automata

An evolutionary CA model has been applied here to structural optimization by combining the strengths of CA and a Genetic Algorithm (GA)[3], to find local rules of a CA which can minimize the weight of a plate structure. The plate is subjected to an external load, which may be distributed or at a point. The plate is 50mm X 50mm square with fixed left edge and concentrated load on the right edge. This plate is divided into 25 unit square elements and a discrete set of plate thickness is defined for each of these elements. The Cellular Automata encodes the configuration of this plate, with the state of the cells in the CA representing the discrete set of plate thickness, changing from one thickness to any other as a result of interaction with its neighboring elements as defined by the exhaustive set of CA rules. These rules are then subjected to evolutionary improvement by undergoing the Genetic Algorithm iterative cycle. The CA lattice starts out with an Initial Configuration (IC) of cell states (0s and 1s) and this configuration changes in discrete time steps in which all cells are updated simultaneously according to the CA rules. A table of these rules is encoded in the GA chromosome string and they evolve over time to give a rich set of rules that can take any starting random IC of a CA to a desired final configuration.

3.1 Cellular Automata

The structure of a system need not be complicated for its behavior to be highly complex, corresponding to a complicated computation. Computational irreducibility may thus be found even among systems with simple construction. Cellular Automata (CA) provide such an example [1]. A CA consists of a lattice of cells, each with k possible values, and each updated in time steps by a deterministic rule, depending on the neighborhood of R sites. Cellular automata are thus mathematical idealizations of physical systems in which space and time are discrete, and physical quantities take on a finite set of discrete values. A cellular automaton typically consists of a regular uniform

array of cells. The state of the cellular automaton is completely specified by the values of the variables at each of these cells. The cellular automaton evolves in discrete time steps, with the value of the variable at one cell being affected by the values of the variables at sites in its "neighborhood" on the previous time step. The neighborhood of a cell is typically taken to be the cell itself and all immediately adjacent cells. The variables at each cell are updated simultaneously ("synchronously"), based on the values of the variables in their neighborhood at the preceding time step, and according to a definite set of local rules [1].

A one-dimensional cellular automaton is a lattice of N two-state machines ("cells"), each of which changes its state as a function only of the current states in a local neighborhood. The lattice starts out with an initial configuration (IC) of cell states (0s and 1s) and this configuration changes in discrete time steps in which all cells are updated simultaneously according to the CA "rule" . Here the term "state" is used to refer to the value of a single cell. The term "configuration" is used to refer to the collection of local states over the entire lattice. A CA's rule can be expressed as a lookup table ("rule table") that lists, for each local neighborhood, the state which is taken on by the neighborhood's central cell at the next time step. For a binary-state CA, these update states are referred to as the "output bits" of the rule table. In a one-dimensional CA, a neighborhood consists of a cell and its r ("radius") neighbors on either side. The CA implemented in our model does not have periodic boundary conditions where the lattice is viewed as being circular, instead special rules are described for the edges of the lattice (boundaries).

Cellular automata have been studied extensively as mathematical objects, as models of natural systems, and as architectures for fast, reliable parallel computation. However, the difficulty of understanding the emergent behavior of CAs or of designing CAs to have desired behavior has up to now severely limited their use in science and engineering and for general computation and ofcourse for optimization. The work described here is on using genetic algorithms to obtain certain optimal CAs to perform computations for optimization tasks. Typically, a CA performing a computation means that the input to the computation is encoded as the Initial Configuration (IC), the output is decoded from the configuration reached at some later time-step, and the

intermediate steps that transform the input to the output are taken as the steps in the computation. The computation emerges from the CA rule being obeyed by each cell.

To produce CAs that can perform sophisticated parallel computations, the Genetic Algorithm (GA) must search for CAs in which the actions of the cells, taken together, is coordinated so as to produce the desired behavior. This coordination must, of course, happen in the absence of any central processor or central memory directing the coordination. Some early work on evolving CAs with GAs was done by Packard [4]. Koza[5] also applied genetic programming to evolve CAs for simple random-number generation. In this work, we have used a form of the GA to evolve one-dimensional, binary-state $r = 2$ CAs to perform a structural optimization task.

3.2 Genetic Algorithms

Genetic Algorithms[3] are distinguished by their parallel investigation of several areas of a search space simultaneously by manipulating a population, members of which are coded problem solutions. The task environment for these applications, is modeled as an exclusive evaluation function which, in most cases is called a fitness function that maps an individual of the population into a real scalar. The motivational idea behind GA is natural selection. Genetic operators like selection, crossover and mutation are implemented to emulate the process of natural evolution. A population of "organisms" (usually represented as bit strings) is modified by the probabilistic application of the genetic operators from one generation to the next. GAs also has a potential for multi-dimensional optimization as they work with population of solutions rather than a single solution. A detailed explanation of the theory and working of the GA can be found in numerous existing literatures on the subject, for example in Goldberg [3].

4 GA Encoding Of The CA Rules

The principal difficulty in this model is to find a suitable encoding technique for each GA genotype string (chromosome or bit string) which has to represent a candidate rule set (or a rule table). We propose a new encoding technique here. A CA's rule table Π can be expressed as a rule-table that lists, for each local neighborhood, the state which is taken on by the neighborhood's central cell at the next time step. This is illustrated in Figure 1. For a binary-state CA, these update states are referred to as the "output-bits" of the rule table as shown in Figure 2. For each of, all the possible permutations of neighborhood values (r = 1), we first define the output bit.

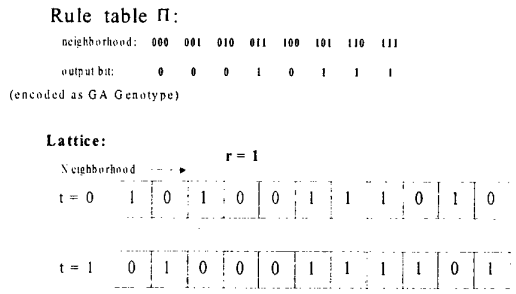


Fig 1: Rule Table (Π) representation technique in one-dimensional, binary state, nearest neighbor (r =1) cellular automaton with N=11.

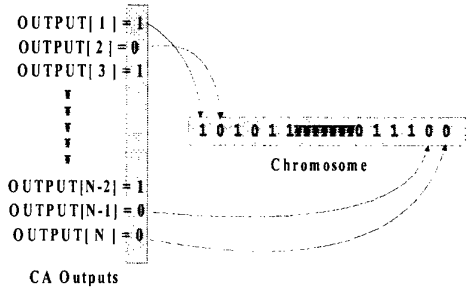


Fig 2: Output bit of each rules is encoded as

the "genes" in the GA genotype.

Each GA chromosome (bit string) consists of the output bit and the whole set of local rules (rule table) is encoded as the chromosome. Thus each population member (chromosome) of the GA represents a candidate rule table in our model. We consider unit length neighborhood of cells in four cardinal directions, which affect the state of each cell, as shown in Figure 3. Thus we have four cells that affect each cell plus the cell itself. These five cells have 2 state each making it 32 states in all (2^5=32), consequently making it 32 basic rules in all. We then have special rules as: *Case 1* - Referring to Figure 4 we have one generalized rule for corner elements (1,2,3,4) and one generalized rule for each group of edge elements (5,6,7,8) and 32 rules for elements marked 9. This makes 34 rules in all that are evolved by the GA. *Case 2*. - Referring to Figure 4 we have four special rules for each of the corner elements (1,2,3,4) and four special rules for each group of the edge elements (5,6,7,8) and 32 rules for elements marked 9. This makes 40 rules in all that are evolved by the GA.

5 Numerical application

The CA evolves through a series of transformation of its states which change in discrete time, given the local rule set for each of *Case 1*, and *Case 2* described above. We limit the CA transformation to 25 and then analyze the plate structure, (with variable material distribution through out its 25 elements) for stresses, by a Finite Element Analysis (FEA) program. The *Mises* stresses of each elements are calculated by FEA by employing decomposition of each square element into the upper and lower triangular element and determining the constraint violation conditions of these triangular element. If there are any stresses which violate the given stress constraint and allowances, we add an appropriate penalty on its weight, which proportionately reduces the *fitness* returned to the GA for that particular table of local rules (GA chromosome). Each set of local rule is subjected to 100 different ICs of

the CA and we use the average fitness for all these 100 ICs as the fitness that is returned to the GA to evaluate the goodness or utility of each set of local rules that are generated by each GA iteration. Thus, we progressively modify the set of local rules which helps us to achieve the minimum weight design in the smallest number of CA state transformation cycles. The overall system architecture is illustrated in Figure 5.

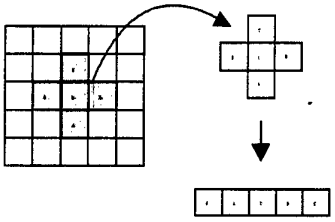


Fig 3: Neighborhood of each cell in four directions that are considered while encoding the CA computations in the GA genotype.

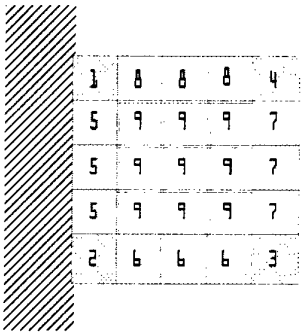


Fig 4: Special CA rules of edges and for corners of the structural plate.

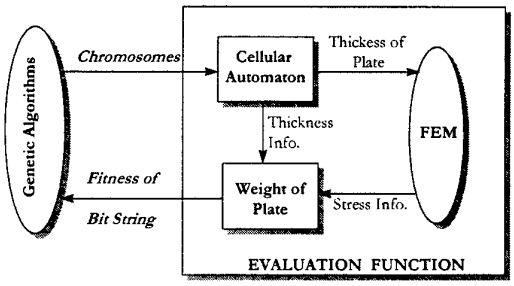


Fig 5: The system architecture for the evolving CA rules for structural optimization.

Table A. The 32 CA rules that produced the structural plate design shown in Figure 10.

00000	00001	00010	00011	00100	00101	00110	00111	01000	01001
0	1	1	1	0	1	1	1	1	0
01010	01011	01100	01101	01110	01111	10000	10001	10010	10011
0	0	1	0	1	0	0	0	0	0
10100	10101	10110	10111	11000	11001	11010	11011	11100	11101
0	0	0	1	0	1	0	0	1	1
11110	11111								
0	0								

5.1 Optimization Results

Figures 6 to 10 show results of preliminary experimentation and computer simulation. These results also confirm that a self-organizing approach can be used under the limitation of computational abilities, to find (or learn) the best set of local rules for optimization of plate like structures with various distributed plate thicknesses. We believe that the figures of optimization results will be comprehended better by the reader, rather than the final rules and so we present the figures of final results obtained. Nevertheless as an example, some final evolved rules are shown in Table A. The 32 CA local rules shown in Table A, along with the 8 edge rules (Case 2 : ref. Section 3.1) produce the final design presented in Figure 10. The top lines of Table A shows the neighborhood of the cells and the bottom lines, the output bit (ref. Figure 1).

6 Conclusions

The discovery of rules that produce global optimization of structural plates show instances of GA's producing sophisticated emergent computation in decentralized, distributed systems such as CAs. The rule

discoveries made by a GA are encouraging for the prospect of using GAs to automatically evolve computation for more complex tasks and in more complex optimization systems. Moreover, evolving CAs with GAs also gives us a tractable framework in which to study the mechanisms by which an evolutionary process might create complex coordinated behavior in natural decentralized distributed systems.

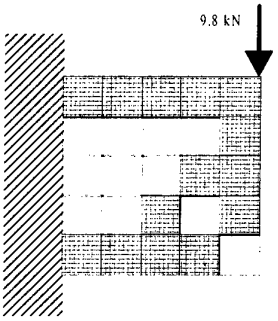


Figure6.

Number of rules = 40
GA Population = 500
Crossover rate = 35%
Mutation rate = 1%
Generation = 90

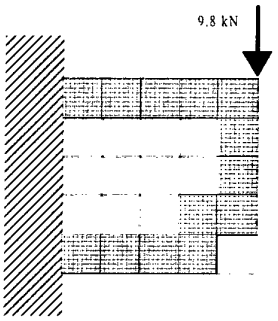


Figure7.

Number of rules = 40
GA Population = 300
Crossover rate = 30%
Mutation rate = 3%
Generation = 95

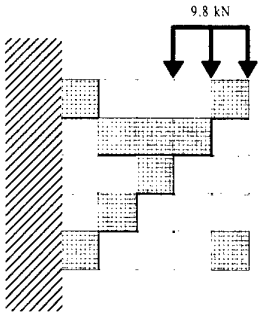


Figure8.

Number of rules = 34
GA Population = 300
Crossover rate = 30%
Mutation rate = 1%
Generation = 23

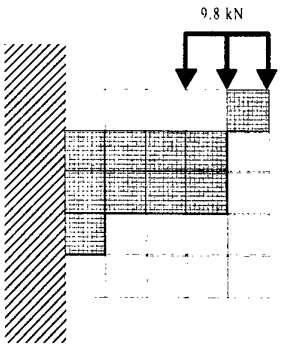


Figure9.

Number of rules = 40
GA Population = 300
Crossover rate = 30%
Mutation rate = 3%
Generation = 30

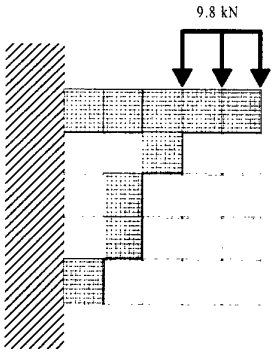


Figure10.

Number of rules = 40
GA Population = 150
Crossover rate = 40%
Mutation rate = 1%
Generation = 66

t = 1.0mm

t = 10.0mm

References

1. Wolfram, S., (1994). *Cellular Automata and Complexity, Collected Papers*. Addison-Wesley, Reading, Massachusetts.
2. Mitchell, M.; Crutchfield, J. P. and Das, R., (1996). "Evolving cellular automata with genetic algorithms: A review of recent works.", Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96).
3. Goldberg, D. E., (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Massachusetts.
4. Packard, N. H., (1988). "Adaptation toward the edge of chaos". In J. A. S. Kelso, A. J. Mandell, M. F. Shlesinger, eds., *Dynamic Patterns in Complex Systems*, 293-301. Singapore: World Scientific.
5. Koza, J. R., (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

Power System Reliability by Sequential Monte Carlo Simulation on Multicomputer Platforms

Carmen L.T. Borges Djalma M. Falcão

Federal University of Rio de Janeiro
P.O. Box 68504, 21945-970, Rio de Janeiro - RJ
Brazil
carmen@dee.ufrj.br, falcao@coep.ufrj.br

Abstract. A powerful technique used for power system composite reliability evaluation is Monte Carlo Simulation (MCS). There are two approaches to MCS in this context: non-sequential MCS, in which the system states are randomly sampled, and sequential MCS, in which the chronological behaviour of the system is simulated by sampling sequences of system states for several time periods. The sequential MCS approach can provide information that the non-sequential can not, but requires higher computational effort and is more sequentially constrained. This paper presents a parallel methodology for composite reliability evaluation using sequential MCS on three different computer platforms: a scalable distributed memory parallel computer IBM RS/6000 SP with 10 processors, a network of workstations (NOW) composed of 8 IBM RS/6000 43P workstations and a cluster of PCs composed of 8 Pentium III 500MHz personal microcomputers. The results obtained in tests with actual power system models show considerable reduction of the simulation time, with high speedup and good efficiency.

1 Introduction

The primary function of electric power systems is to satisfy the consumers' demand in the most economic way and with an acceptable degree of continuity, quality and security. The ideal situation would be that the energy supply was uninterrupted. However, the occurrence of failures of some components of the system can produce disturbances capable of leading to the interruption of the electric energy supply. In order to reduce the probability, frequency and duration of these failure events and their effects, it is necessary to accomplish financial investments in order to increase the reliability of the system. It is evident that the economic and the reliability requirements can conflict and make it difficult to take the right decisions.

The new competitive environment of the electric energy market makes the evaluation of energy supply reliability of fundamental importance when closing contracts between utilities companies and great consumers. In this context, the definition of the costs associated with the supply interruption deserves special attention, since engineers must now evaluate how much it is interesting to invest

in the system reliability, as a function of the cost of the investment itself and the cost of the interruption for the consumer and for the energy vendors. This new environment also requests the reliability evaluation of larger parts of the interconnected system and it can demand, in some cases, nation-wide systems modelling. For this purpose, it becomes a necessity to develop computational tools capable of modelling and analysing power systems of very high dimensions.

One of the most important methods used for reliability evaluation of power systems composed of generation and transmission sub-systems is Monte Carlo Simulation (MCS). MCS allows accurate modelling of the power system components and operating conditions, provides the probability distributions of variables of interest, and is able to handle complex phenomena and a great number of severe events [1, 2].

There are two different approaches for Monte Carlo simulation when used for composite system reliability evaluation: Non-Sequential MCS and Sequential MCS. In Non-Sequential MCS, the state sampling approach is used, in which case the state space is randomly sampled without concerning the system operation process chronology. This implies in disregarding the transitions between system states. In Sequential MCS, the chronological representation is adopted, in which case the system states are sequentially sampled for several periods, usually years, simulating a realisation of the stochastic process of system operation. The expected values of the main reliability indices can be calculated by both approaches. However, estimates of specific energy supply interruption duration and the probability distribution of duration related indices can only be obtained by sequential MCS [3]. In applications related to production cost evaluation, only the sequential approach can be used. On the other hand, sequential MCS demands higher computational effort than non-sequential MCS. Depending on the system size and modelling level, the sequential MCS computer requirements in conventional computer platforms may become unacceptable [4].

In both MCS approaches, the reliability evaluation demands the adequacy analysis of a very large amount of system operating states, with different topological configurations and load levels. Each one of these analyses simulates the operation of the system at that particular sampled state, in order to determine if the energy demand can be attended without operating restrictions and security violations. The main difference between the two approaches is concerned with the way the system states are sampled, which is randomly done in the non-sequential MCS and is sequentially sampled in time in sequential MCS. Each new sampled state in sequential MCS is dependent of the configuration and duration of the previously sampled one.

This paper describes results obtained by a parallel methodology for composite reliability evaluation using sequential MCS on three different multicomputer platforms: a scalable distributed memory parallel computer IBM RS/6000 SP with 10 processors, a network of workstations (NOW) composed of 8 IBM RS/6000 43P workstations and a cluster of PCs composed of 8 Pentium III 500MHz personal microcomputers. In a previous paper [5], a methodology for parallelisation of composite reliability evaluation using non-sequential MCS was

presented. Tests performed on three electric systems showed results of almost linear speedup and very good efficiency obtained on a 4 nodes IBM RS/6000 SP parallel computer. As a continuation of that work, this paper now deals with sequential MCS on three different multicomputer platforms. The chronological dependency between consecutive sampled states that exists in the sequential MCS approach introduces much more complexity in developing a parallel algorithm to solve the problem than there was in non-sequential MCS. Although MCS techniques are used to sample the system state configuration, each of them is now coupled in time with the next one, and this introduces significant sequential constraints in the parallelisation process.

The methodology presented in this paper is based on coarse grain asynchronous parallelism, where the adequacy analysis of the system operating states within each simulated year is performed in parallel on different processors and the convergence is checked on one processor at the end of each simulated year. Some actual power system models are used for evaluating the performance of the methodology and the scalability correlation with the network architecture and bandwidth.

2 Sequential Monte Carlo Simulation

The power system reliability evaluation consists of the calculation of several indices, which are indicators of the system adequacy to the energy demand, taking into consideration the possibility of occurrence of failures of the components. In particular, the composite reliability evaluation considers the possibility of failures at both the generation and the transmission sub-systems. A powerful technique used for power system composite reliability evaluation is MCS [1]. One possible approach is the chronological representation of the system operation stochastic process, in which case the system states are sequentially sampled in time. One implementation of the chronological representation is the use of sequential MCS. In sequential MCS, the system operation is simulated by sampling sequences of operating states based on the probability distribution of the components states duration. These sequences are sampled for several periods, usually years, and are called yearly synthetic sequences. The duration of the states and the transitions between consecutive system states are represented in these synthetic sequences.

The reliability indices calculation using sequential MCS may be represented by the evaluation of the following expression:

$$\bar{E}(F) = \frac{1}{N} \sum_{k=1}^{N} F(y_k) \tag{1}$$

where

N: number of simulated years

y_k: yearly synthetic sequence composed of the sampled system states within year k

F: adequacy evaluation function to calculate yearly reliability indices over the sequence y_k
$\bar{E}(F)$: estimate of the expected value of the adequacy evaluation function

The reliability indices correspond to estimates of the expected value of different adequacy evaluation functions F for a sample composed of N simulated years. For calculation of the values of F associated with the various indices, it is necessary to simulate the operating condition of all system states within a year. Each simulation requires the solution of a static contingency analysis problem and, in some cases, the application of a remedial actions scheme to determine the generation re-scheduling and the minimum load shedding. Most of the computational effort demanded by the algorithm is concentrated is this step.

The convergence of the evaluation process is controlled by the accuracy of MCS estimation by the coefficient of variation α, which is a measure of the uncertainty around the estimates, and is defined as:

$$\alpha = \frac{\sqrt{V(\bar{E}(F))}}{\bar{E}(F)} \tag{2}$$

where $V(\bar{E}(F))$ is the variance of the estimator.

A conceptual algorithm for composite reliability evaluation using sequential MCS is described next:

1. *Generate a yearly synthetic sequence of system states y_k;*
2. *Chronologically evaluate the adequacy of all system states within the sequence y_k and accumulate these results;*
3. *Calculate the yearly reliability indices $F(y_k)$ based on the values calculated in step (2).*
4. *Update the expected values of the process reliability indices $\bar{E}(F)$ based on indices calculated in step (3);*
5. *If the accuracy of the estimates of the process indices is acceptable, terminate the process. Otherwise, return to step (1).*

The yearly synthetic sequence is generated by combining the components states transition processes and the chronological load model variation in the same time basis. The component states transition process is obtained by sequentially sampling the probability distribution of the component states duration, which may follows an exponential or any other distribution. This technique is called State Duration Sampling Approach.

2.1 State Duration Sampling Approach

This sampling process is based on the probability distribution of the component states duration. The chronological component state transition processes for all components are first simulated by sampling. The chronological system state transition process is then created by combination of the chronological component state transition process [1]. This approach is detailed in the algorithm below for the case of exponential probability distribution for the component state duration:

1. *Specify the initial state of the system by the combination of the initial state of all components;*
2. *Sample the duration of each component residing in its present state (t_i) by:*

$$t_i = -\frac{1}{\lambda_i} \ln U_i \tag{3}$$

   where λ_i is the transition rate of the component and U_i is a uniformly distributed random number between [0,1].
3. *Repeat step (2) for the whole period (year) and record sampling values of each state duration for all components.*
4. *Create the chronological system state transition process by combining the chronological component state transition processes obtained in step (3) for all components. This combination is done considering that a new system state is reached when at least one component changes its state.*

This approach is illustrated in Fig. 1 for a system composed of two components represented by two-state stochastic model.

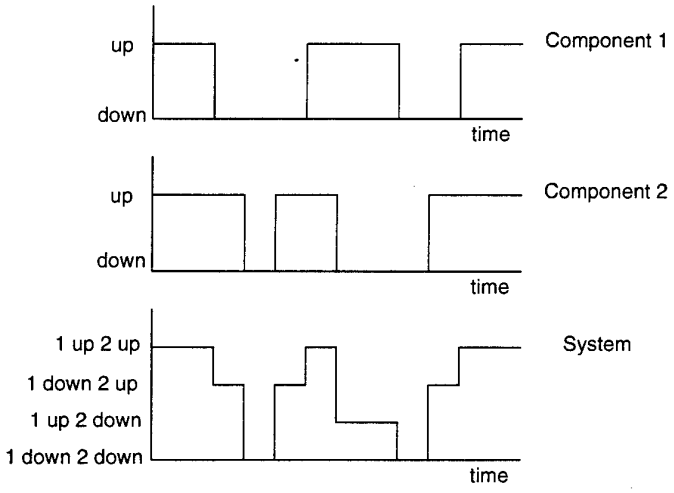


Fig. 1. State Duration Sampling

3 Parallel Methodology

One possible approach to parallelise the problem described above is to have a complete year analysed on a single processor and the many years necessary to

converge the process analysed in parallel on different processors. This implies that the parallel processing grain is one year simulation. However, this approach does not scale well with the number of processors and the number of simulated year for convergence [6].

A more scalable approach to parallelise the problem is to analyse each year in parallel by allocating parts of one year simulation to different processors. Since within a yearly synthetic sequence the adequacy analysis of the system operating states should be chronologically performed, this parallelisation strategy requires a careful analysis of the problem and more a complex solution.

The generation of the yearly synthetic sequence is a strictly sequential process, since each state depends on the previous one. However, most of the computation time is not spend in the sequence generation itself, but in the simulation of the system adequacy at each state that compounds the sequence. In that sense, if all processors sample the same synthetic sequence, the adequacy analysis of parts of this sequence can be allocated to different processors and performed in parallel. Of course some extra care must be taken in order to group the partial results of these sub-sequences and calculate the yearly reliability indices.

In the methodology used in this paper, the whole synthetic sequence is divided in as many sub-sequences as the number of scheduled processors, the last processor getting the remainder if the division is not exact. Each processor is then responsible for analysing the states within a particular sub-sequence. In a master-slave model, at the end of each sub-sequence analysis, the slaves send to the master their partial results and start the simulation of another sub-sequence in the next year. The master is responsible for combining all these sub-sequence results sequentially in time and compounding a complete year simulation. Since this methodology is asynchronous, the master has to keep track of which year does a sub-sequence result it receives is related to and accumulate the results at the right year. Each time it detects that a year has been completely analysed, it calculates the yearly reliability indices and verifies the convergence of the process. When convergence is achieved, the master sends a message to all slaves to stop the simulation, calculates the process reliability indices, generates reports and terminates execution.

The methodology precedence graph is shown in Fig. 2. Each processor has a rank in the parallel computation which varies from 0 to (p-1), 0 referring to the master process. The basic tasks involved are: **I** - **I**nitialisation, **A** - Sub-sequence States **A**nalysis, **R** - **R**eception and Control of Sub-sequences, **C** - **C**onvergence Control, **S** - Individual **S**tates Analysis and **F** - **F**inalisation. A superindex k,i associated with a task means it is relative to the i-th sub-sequence within year k and a superindex k,i,j means it is relative to the j-th state within sub-sequence i in year k.

Since sequential MCS simulates a chronological evolutionary process, a system state configuration is dependent on the topological evolution of the previous states. The adequacy analysis of a state will determine if it is an up or down state and adequate procedures must be taken, depending on the kind of transition that lead to that state (up-up, up-down, down-up, down-down). In order

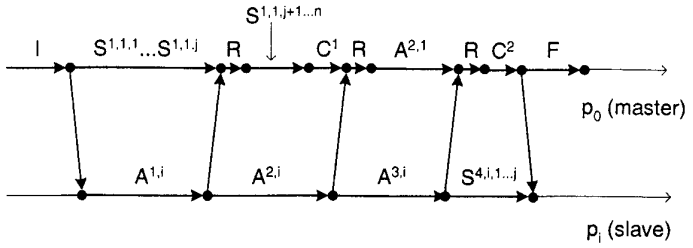


Fig. 2. Precedence Graph

to identify the transition that occurs between sub-sequences analysed in parallel on different processors, the last state of each sub-sequence is analysed on two processors: the one responsible for that sub-sequence and the one responsible for the next sub-sequence. This allows the knowledge if the first state of a sub-sequence comes from an up or down state, permitting that the proper action be taken. This solution is illustrated in Fig. 3 for 4 processors.

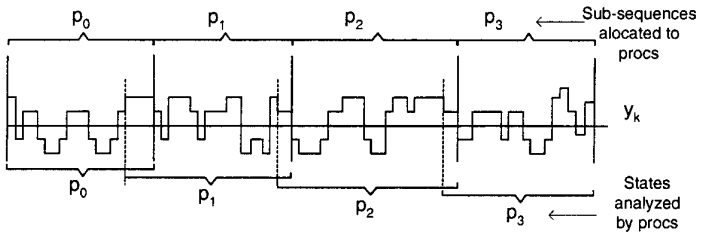


Fig. 3. Consecutive Sub-Sequences Transition

A very important problem that must be treated in the division of a synthetic sequence in sub-sequences to be analysed in parallel is if this border coincides with a failure sub-sequence of the whole synthetic sequence. A failure sub-sequence is a sequence of failure states, which corresponds to an energy supply interruption of duration equal to the sum of the individual failure state duration. If this coincidence occurs and is not properly treated, the duration related indices and their distribution along the months are wrongly evaluated because the failure sub-sequence is not completely detected and evaluated at the same processor. To solve this problem, the methodology makes a failure sub-sequence to be completely evaluated at the processor on which the first failure state of the sub-sequence occurs, as illustrated in Fig. 4.

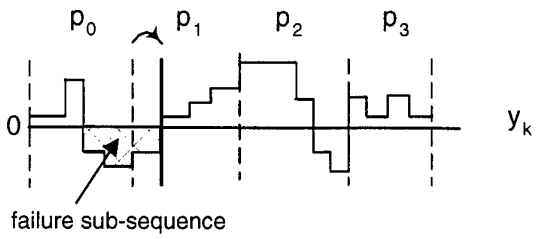


Fig. 4. Failure Sub-sequences Treatment

If the last state of a processor allocated sub-sequence is a failure state, what means that the simulation is within a failure sub-sequence, that processor carries on analysing the next states until a success state is reached, ensuring that the whole failure was analysed on it. In a similar way, if the state before the first of a sub-sequence allocated to a processor is a failure state, that processor skips all states until a success state is reached and starts to accumulate the results from this state on. This guarantees the correctness of the duration related indices evaluation and distribution but adds some computation overhead cost to the overall simulation. However, some extra costs must usually be added when parallelising any sequentially constrained application.

4 Results

This work was implemented on three different multicomputer platforms.

1. A scalable distributed memory parallel computer IBM RS/6000 SP composed of 10 POWER2 processors interconnected by a high performance switch of 40 MBps full-duplex bandwidth and 50 μsec latency;
2. A network of workstations (NOW) composed of 8 IBM RS/6000 43P workstations interconnected by an Ethernet (10Base-T) network. The peak bandwidth of this network is 10 Mbps unidirectional;
3. A PC cluster composed of 8 Pentium III 500MHz personal microcomputers interconnected by a Fast-Ethernet (100 Base-T) network via a 12 ports 100 Mbps switch. Each PC has 128 MB RAM and 6.0 GB IDE UDMA hard disk. The peak bandwidth and latency of the network is 100 Mbps unidirectional and 500 μsec, respectively. The operating system running over the network is Windows NT 4.0.

The message passing system used on the first and second platforms is the MPI implementation developed by IBM for the AIX operating system. On the PC cluster it is used the WMPI v1.2 [7], which is a freeware MPI implementation developed at the Coimbra University, Portugal, for Win32 platforms. It is

based on MPICH 1.1.2 and uses the ch-p4 device developed at Argonne National Laboratory (ANL) [8]. The implementations used on the three platforms comply with MPI standard version 1.1 [9].

Three different electric systems were used as test to verify the performance and scalability of the parallel implementations. The first one is a representation of the New Brunswick power system (NBS) proposed by CIGRÉ as a standard for reliability evaluations [10]. This system has 89 nodes, 126 circuits and 4 control areas. The second and third systems are representations of the Brazilian power system, with actual electric characteristics and dimensions, for Southern region (BSO) and Southeastern region (BSE), respectively. These systems have 660 buses, 1072 circuits and 78 generators and 1389 buses, 2295 circuits and 259 generators, respectively. A convergence tolerance of 5% in the coefficient of variation of the EPNS index was adopted in all simulations.

The parallel efficiency obtained on 4, 6 and 10 processors of the IBM RS/6000 SP parallel computer for the test systems, together with the CPU time of the mono-processor execution, are summarised on Table 1.

Table 1. RS/6000 SP Results

System	CPU time	Efficiency (%)		
	p=1	p=4	p=6	p=10
NBS	30.17 min	97.32	94.64	84.62
BSO	13.02 min	93.57	93.27	82.23
BSE	15.30 hour	97.81	97.41	91.20

The application of the parallel methodology produces significant reduction of the simulation time required for reliability evaluation using sequential MCS. The efficiencies are very good staying beyond 82% for all test systems on 10 nodes. The methodology is scalable with the number of simulated years required for convergence, which varies from 13 years for the BSO system to 356 for the NBS system, and also with the number of allocated processors.

The parallel efficiency obtained on 4, 6 and 8 workstations of the NOW and the CPU execution time on one workstation are summarised on Table 2.

Table 2. NOW Results

System	CPU time	Efficiency (%)		
	p=1	p=4	p=6	p=8
NBS	14.43 min	87.96	80.21	73.11
BSO	6.18 min	87.26	72.29	65.36
BSE	7.08 hour	92.90	91.57	90.60

The analysis of the efficiency achieved on the NOW shows the higher communication cost of an Ethernet network in comparison with the high performance switch of the RS/6000 SP. Most of the communication time at this platform is spent by the initial broadcast of the problem data and this is a consequence of two characteristics: first, the smaller bandwidth of this network and second, the fact that the MPI broadcast is a blocking directive, implemented as a sequence of point-to-point communications. In an Ethernet bar network topology this costs more than in a multinode interconnected switch. The scalability can also be considered good especially for the larger system.

The parallel efficiency obtained on 4 and 8 PCs of the cluster and the CPU execution time on one single PC are summarised on Table 3.

Table 3. PC Cluster Results

System	CPU time	Efficiency (%)		
	p=1	p=4	p=6	p=8
NBS	5.37 min	88.79	85.33	81.80
BSO	2.06 min	92.37	85.22	75.28
BSE	2.28 hour	98.64	98.11	96.25

The results achieved on this platform can be considered excellent, specially if taken into consideration the low cost, easiness of use and high availability of the computing environment. The sequential simulation time is already smaller than in the other platforms as a consequence of the more modern and powerful processor used. The efficiency of the parallel solution can be considered very good reaching more then 96% for the larger and more time consuming test system on 8 PCs. The parallel results show higher efficiency than the NOW ones mostly due to the higher bandwidth of the network and the use of a 100 Mbps switch in a star topology. As a consequence, the scalability of the methodology is less affected by the increase in the number of processor.

The speedup curves of the parallel methodology are shown on Figures 5, 6 and 7 for the RS/6000 SP, NOW and PC Cluster platforms, respectively.

5 Conclusions

The power system composite reliability evaluation using the sequential MCS approach simulates a realisation of the stochastic process of the system operation. Power supply interruption duration and probability distribution of duration related indices can be calculated, what is not possible using the non-sequential MCS approach. These issues are fundamental in production cost studies that are receiving more and more attention on the new competitive environment of power system markets. However, the major drawback of sequential MCS application is the high elapsed computation time required on conventional platforms

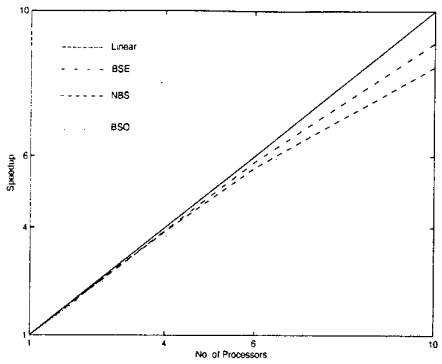


Fig. 5. RS/6000 SP Speedup Curve

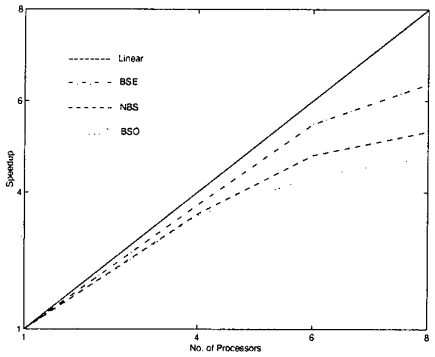


Fig. 6. NOW Speedup Curve

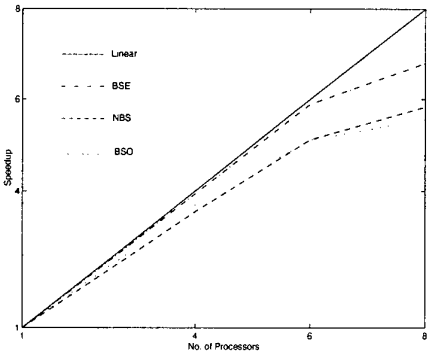


Fig. 7. PC Cluster Speedup Curve

for elevated dimension system models. This paper presented a parallel methodology for solving this problem, implemented on three different multicomputer platforms. The good results obtained in this work show that the computational cost paid by the parallelisation of a sequentially constrained problem are fairly compensated by the overall reduction of the simulation time. Even in the cases where the efficiency are not that high, the engineer production time saved by the use of the parallel methodology justifies it use. Moreover, the use of low cost platforms like a NOW or a cluster of PCs, which are usually available at any scientific institution and at most electric utilities, aggregate a new appeal to the adoption of parallel processing as a reliable and economic computing environment.

References

1. R. Billinton and W. Li, *Reliability Assessment of Electric Power Systems Using Monte Carlo Methods*, Plenum Press, New York, 1994.
2. M.V.F. Pereira and N.J. Balu, "Composite Generation / Transmission Reliability Evaluation", *Proceedings of the IEEE*, vol. 80, no. 4, pp. 470-491, April 1992.
3. R. Billinton, A. Jonnavithula, "Application of Sequential Monte Carlo Simulation to Evaluation of Distributions of Composite System Indices", *IEE Proceedings - Generation, Transmission and Distribution*, vol. 144, no. 2, pp. 87-90, March 1997.
4. D.M. Falcão, "High Performance Computing in Power System Applications", *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1215, pp. 1-23, February 1997.
5. C.L.T. Borges and D.M. Falcão, "A Parallelisation Strategy for Power Systems Composite Reliability Evaluation (Best Student Paper Award: Honourable Mention)", *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1573, pp. 640-651, 1999.
6. C.L.T. Borges, "Power Systems Composite Reliability Evaluation on Parallel and Distributed Processing Environments", *PhD Thesis*, in portuguese, COPPE/UFRJ, Brazil, December 1998.
7. J.M. Marinho, "WMPI v1.2", *http://dsg.dei.uc.pt/wmpi*, Coimbra University, Portugal.
8. R. Butler, E. Lusk, "User's Guide to the p4 Parallel Programming System", *ANL-92/17, Mathematics and Computer Science Division*, Argonne National Laboratory, October 1992.
9. M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra,"*MPI: The Complete Reference*", The MIT Press, Cambrige, Massachusetts, 1996.
10. CIGRÉ Task Force 38-03-10, "*Power System Reliability Analysis - Volume 2 - Composite Power Reliability Evaluation*", 1992.