AD-A264 990

MENTATION PAGE

Form Approved
OMB No. 0704-0188

1

1 AGENCY USE ONLY *(Leave blank)*	2 REPORT DATE March 1993	3 REPORT TYPE AND DATES COVERED Professional Paper
4 TITLE AND SUBTITLE HAsP—HETEROGENEOUS ASSOCIATIVE PROCESSING		5 FUNDING NUMBERS PR: ZW65 PE: 0601152N WU: DN302054
6 AUTHOR(S) R. F. Freund and J. L. Potter		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Command, Control and Ocean Surveillance Center (NCCOSC) RDT&E Division San Diego, CA 92152-5001		8 PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Chief of Naval Research Independent Exploratory Development Program (IED) OCNR-20T Arlington, VA 22217		10 SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Heterogeneous processing (HP) is a technique intended for Grand Challenge and other high performance computing (HPC) problems and for bridging the gap between the theoretical potential of parallel processing and the current reality. In HP, we aim to match code and algorithms to best-suited architectures, through techniques such as code profiling and analytic benchmarking. Associative computing (AsC) combines ideas from both associative memories and single instruction multiple data (SIMD) computers to look at new ways to use fine-grain parallel processors to achieve results beyond what is normally done by using spinoffs from sequential or multiple instruction multiple data (MIMD) processors. Heterogeneous associative processing (HAsP) is a generalization of the concepts of both HP and AsC. In HAsP, the AsC assumption of linking each datum with its own processor is generalized to assuming that each data file has its own dedicated computer. This paradigm maps onto all levels of granularity and can be easily emulated on most machines. The goal of HAsP is to allow the user to discuss the heterogeneous system at the highest possible level and with the tightest possible synchronism. HAsP offers the potential of combining the simplifying programming approaches and algorithmic efficiencies of AsC with the performance of HP.

424521

Published in *Computing Systems in Engineering*, Vol. 3, Nos. 1–4, pp 25–31.

14. SUBJECT TERMS high performance computing heterogeneous processing superconcurrency			15. NUMBER OF PAGES
			16 PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAME AS REPORT

NSN 7540-01-280-5500

Standard form 298 (FRONT)

21a. NAME OF RESPONSIBLE INDIVIDUAL	21b. TELEPHONE *(Include Area Code)*	21c. OFFICE SYMBOL
R. F. Freund	(619) 553–4071	Code 423

HAsP HETEROGENEOUS ASSOCIATIVE PROCESSING

R. F. Freund† and J. L. Potter‡

†Naval Command, Control and Ocean Surveillance Center, NRaD 423, 271 Catalina Blvd, San Diego, CA 92125-5000, U.S.A.
‡Kent State University, U.S.A.

Abstract—Heterogeneous processing (HP) is a technique intended for Grand Challenge and other high performance computing (HPC) problems and for bridging the gap between the theoretical potential of parallel processing and the current reality. In HP, we aim to match code and algorithms to best-suited architectures, through techniques such as code profiling and analytic benchmarking. Associative computing (AsC) combines ideas from both associative memories and single instruction multiple data (SIMD) computers to look at new ways to use fine-grain parallel processors to achieve results beyond what is normally done by using spinoffs from sequential or multiple instruction multiple data (MIMD) processors. Heterogeneous associative processing (HAsP) is a generalization of the concepts of both HP and AsC. In HAsP, the AsC assumption of linking each datum with its own processor is generalized to assuming that each data file has its own dedicated computer. This paradigm maps onto all levels of granularity and can be easily emulated on most machines. The goal of HAsP is to allow the user to discuss the heterogeneous system at the highest possible level and with the tightest possible synchronism. HAsP offers the potential of combining the simplifying programming approaches and algorithmic efficiencies of AsC with the performance of HP.

1. INTRODUCTION

Distributed heterogeneous high performance computing (DH-HPC) is the "tuned" use of heterogenous suites of sequential and parallel HPC processors to obtain cost-effective HPC and/or metacomputing performance.[1,2] The essence of DH-HPC is the ability to obtain maximal execution by mapping the computational tasks onto the best-suited architecture. The intent is that for problems with diverse computational subtasks, the overall performance cost-effectiveness will be better than any comparable single processor. In addition we aim to reduce the applications programming effort since well-matched code leads to natural implementations. The beginning forms of DH-HPC were cases in which codes were profiled and run on the best-suited machine with the data following along. As discussed below, however, we believe that the ultimate expression of this paradigm is reached in a DH-HPC environment in which the data are profiled and remain resident on the best-suited machine and different instruction streams are passed to it. In other words, HAsP is an HPC form of MISD processing. The long-term objective is to develop a methodology suitable for a heterogeneous suite of supercomputers. The methodology should be effective, expressive, extensible and efficient. By efficient, we mean easy to use and applicable to all types of architecture. By expressive we mean it is usable for all types of problems. Thus, while DH-HPC problems are our primary target, our goal is to also evaluate other compute intensive problems such as dynamic data bases. Extensible means that the methodology must be flexible enough to accommodate new machines and architectures as they are added to the system. To be efficient the methodology must support the existence and addition of high level operators such as sum, convolution, matrix multiply, etc. The short-term objectives for HAsP are the application of DH-HPC and associative computing principles to develop heterogeneous processor suites spanning wide problem sets and to develop new methods for benchmarking, code and data profiling, and the intelligent management of selected DH-HPC suites. A major short-term objective is the extension of DH-HPC paradigms and associative computing principles to heterogeneous suites forming a virtual associative computer.

2. DISTRIBUTED HETEROGENEOUS HPC

Shared memory and message passing are two basic paradigms of computation derived from conventional multi-user concepts that could be used for a heterogeneous supercomputer system. Linda[3] is a shared memory paradigm based on an associative memory concept. Data (tuples or records) to be processed are "put" into a shared memory and idle processors "get" tuples from the memory to work on. Linda assumes equal power processors which treat data like resources.

Actors[4] is a message passing, fine grain, object oriented approach for concurrent computing. It uses three primitives: create; send; and become. Where the get and put primitives of Linda generalize the shared memory model, the create and send primitives of actors generalize the process creation and message passing concepts. Actors allows you to reason about the system; however, reasoning is done at the "actor", i.e. low level message passing level.

BBN's TCL (tool for large grained concurrency) ses high level linguistic constructs and has a virtual ıachine concept for organizing parallelism. The ompiler divides the program into continuations vhich are parceled out by the scheduler for execution ın the most appropriate machine. Implicitly, this is ι message passing or object oriented approach, .e. the data and command must be sent. The virtual nachine primitives range from host language (LISP) primitives to high level commands. TCL maintains ι local view. If there are multiple users, each has heir own TCL scheduler. Thus each works most efficiently if it has application specific information ınd can concurrently interrogate processors from heir current status.

Both the Linda and Acters models were designed 'or and work well on single MIMD computers, but when the concepts are moved to a heterogeneous, distributed network environment, considerable overhead may be incurred. Linda is essentially a data driven approach controlled by the instances of data tuples in shared memory. By definition, this approach requires a considerable amount of data sharing. When implemented (as intended) on a shared memory machine, sharing requires only memory reads and writes. However, if the paradigm is moved to a distributed heterogeneous system, the data must be physically moved resulting in considerable overhead. Actors, being a message passing model, assumes (guarantees) that messages are delivered, therefore it is possible (probable) that large portions of system resources (both hardware and software) are devoted to message passing (i.e. buffering and forwarding data and commands) not computing.

A general problem of most conventional approaches to heterogeneous computing is that they are bottom up. That is, they provide relatively few low level primitives that support a specific MIMD paradigm (Linda shared memory, actors message passing). These primitives are intended to be imbedded in conventional sequential languages (i.e. FORTRAN, C, etc.) or form the foundation of a specifically designed "high level language" and to be executed on a single computer.

The bottom up approach imposes several restrictions on the paradigms. For example, in Linda, tuples are singular, i.e. they represent a single data object or record. By putting pointers in a tuple, arrays and structures can be referenced, but the basic mode of operation is that the normal case is scalar, the exception is parallel and requires more complicated syntax. A more general, more powerful system is achieved if the primitives are composed of entities which include both parallel and scalar as equivalent cases.

In Actors the recommended approach to execute a loop in parallel on a MIMD machine is to break it into individual messages, one for each iteration of the loop. This approach should not be extended to a heterogeneous environment, since it adds considerable overhead to execution and ignores the communication efficiency and natural parallelism of SIMD, vector and other "tightly coupled" architectures. There should be a minimum of communication between computers to determine what to do next. A high level command which encompasses both parallel (i.e. loops) and scalar operations is needed.

Efficient system management demands that the OS communications be at the highest possible level but computation be at the lowest level of parallelism. For example, function/operations should take files as arguments instead of records or tuples, but vector operations should be done at the machine level not at the system level.

Most distributed operating systems are extensions of MIMD paradigms. As a result, there is no natural way to match the proper computer to the computation. For example, in Distributed Linda, both a SIMD and a MIMD could issue a "get" command for the same tupel. The race condition is determined arbitrarily, not on the basis of which computer is better suited for the task. The basic job could be programed so that the "best" architecture issues the "get"; but then the "next best" architecture may be sitting idle while the "best" one is doing all of the work.

Associative computing[5] is a programming paradigm developed explicitly for massively parallel SIMD computers. It uses the concepts that each datum has its own dedicated processor. Thus, in associated computing, commands are broadcast to all components. Those components which recognize that the commands are applicable to them (using associated techniques) execute them. The associative computing paradigm can be combined with DINS to form an efficient comprehensive DH-HPC system.

3. HETEROGENEOUS ASSOCIATIVE PROCESSING (HAsP)

The basic assumption of associative computing is that each datum has its own dedicated processor. In the HAsP environment, this assumption is generalized to assuming that each data file has its own dedicated (possibly parallel) computer. The distinction between an associative memory and an associative computer cannot be over-emphasized. Associative memories require that the selected data be moved to a central processor before they can be processed. An associative computer has a separate processor for every datum wherever it is located and is thus an inherently distributed system. The associative computing paradigm maps well onto all levels of parallelism from low level massively parallel SIMD computers to high level heterogeneous parallelism and can be easily emulated on most machines, both sequential and parallel.

The goal of HAsP is to develop an environment that allows the user to discuss the heterogeneous system at the highest possible level and with the tightest possible synchronism. That is, the primitives should encompass parallelism and heterogeneous computation as the norm and treat "sequential von Neumann computation" as a special case.

The HAsP paradigm makes several assumptions for large heterogeneous computing networks:

(1) there is much more data than code;

(2) data sets typically have one or a few (related) natural organizations that are basic to the various instructions streams that may be directed against them;

(3) the cost of communication is high compared with the cost of computation (or equivalently, the speed of transmission is slow compared with the speed of computation); and

(4) any system can be viewed as a virtual associative computer with a (small) common set of commands and a multiplicity of data tuples each with their own processor.

The conclusion is that the data should be sent to the most appropriate computer initially and the code sent to the data.

In our opinion HAsP can be viewed as a MISD paradigm, i.e. one in which various instructions streams are sent to any given machine (or CPU) which holds those data sets best suited for it.

A layered operating system, a virtual machine organization, automated data conversion, code and data profiling, and a layered metacompiler are all important concepts for HAsP. At first glance, HAsP is similar to message passing systems; however, there are five significant differences. For example:

(1) In HAsP, commands are broadcast to the entire system, not to specific nodes. Nodes select commands based on their data content, not address. A command consists of an: (i) action; and (ii) argument patterns delineated by keywords.

(2) Data movement is minimized. Commands rarely contain data. However, commands to move data may be sent and as a special case, a command may cause a node to send a reply.

(3) When a command is received the argument patterns are used to search the local data base for items that match. In a multi-user environment, part of the pattern may include user id and/or job numbers. Matched items are flagged. The flags are attached to the appropriate keywords and control is passed to the action software. Pattern matching is performed in a mode appropriate for the local computer. On a SIMD machine, the pattern matching is done in parallel. On a MIMD machine, it "could" be done using message passing of the shared memory paradigm. On a sequential machine it would be done sequentially on sorted tuples or by hashing.

(4) It is possible for two or more commands to be issued which match the same data items. In these cases a "state" item would be included to make tuples unique. At the end of an action cycle the state would be updated.

(5) "Load balancing" is done dynamically on initial data load. That is, the HAsP heterogeneous compiler using static code profiling would consult the concordance to determine the best set of computers to use. At run time, dynamic data profiling would be used to refine the decision. Then the data would be input directly to the appropriate computer. Unlike OOP approaches that must (at least conceptually) move data and code from node to node, HAsP emphasizes the movement of code alone. Once the data have been input to a computer, they are rarely moved or copied.

3.1. *Virtual heterogeneous associative machine*

In HAsP, a layered view of heterogeneous processing is advocated. Each layer consists of a virtual heterogeneous associative machine (VHAM). Thus there is a VHAM for large area (nation wide) networks. A VHAM for region wide, statewide, building wide, and local area networks as well as a VHAM for a single heterogeneous computer. Not all HAsP systems need have all levels, indeed three or four levels of VHAMs would probably be most common. Where the code/data is known to be mutually exclusive, multiple associative commands can be issued. At the top most level, the HAsP commands at each level are of a coarse enough granularity that a single physical channel or bus could be divided into several time shared concurrent channels.

VHAM consists of three parts. First, it is a set of "instructions" which defines a virtual heterogeneous associative machine. Second, it contains an execution engine which processes HAsL instruction. Third, it is a system of protocol where by new user defined "instructions" can be added to the system. Thus VHAM is a paradigm with a predefined minimal run time expandable instruction set and execution protocol. In conventional machines, instructions are delivered to a CPU and they are executed without question. In the VHAM, instructions are broadcast to all of the cells, but each cell must determine whether to execute the instruction. This determination is performed as follows: upon receipt of an instruction, a node "unifies" it with its local instruction set of library calls and extended instruction and datafiles. If there is a match, the appropriate routine is called. The node will perform format conversion if necessary. The called "instruction" may in turn issue VHAM instructions. Thus control is distributed even though every level of the HAsC has a designated control node. That is, a "program" starts by issuing a command from the designated control node. If a receiving node receives a command that is in effect a subroutine call, it may become a transporter node. It may first perform some local computations and then start issuing (broadcasting) commands of its own. If the node happens to be a port node, the commands are issued to its subset as well as to its own network. Thus it is possible (even probable) that multiple instructions streams will be broadcast simultaneously.

3.2. *A virtual associative computer*

The virtual machine organization should be as closely coupled as possible. The main question is: what are the virtual commands? The commands should be very high level (i.e. convolve, fit, Gauss) consisting of entire programs or algorithms. The same algorithm may be in two or three different forms; one for each type of machine in the system. A concordance records all of the forms and related parameters. For example, convolve may have four different morphs, one for each machine type or subtype, i.e. hypercube SIMD, grid SIMD or computation model, messsage passing or shared memory. Each concordance entry states its parameters, speed, data format, etc.

Ideally, the layered OS language would be the same as the virtural machine and concordant application languages. The commands from different OS levels may overlap. They would range from basic to complex depending on the level of the OS. Although the OS and application languages would share the same vocabulary, the OS language is real time and interactive. The application language is "compiled" and executed as a background job. There would be a "macro" mode of operation where the user would enter his job inter-actively via the OS language. When he got the correct results, the history of the OS command could be saved in a file, edited if necessary, and compiled and executed as a program.

3.3. *An associative operating system*

The associate operating system would consist of one per level per virtual associative computer. In order to develop a layered OS, all aspects of a conventional operating system needed to be separated, analyzed and then reorganized. For example, currently in a heterogeneous system, a conventional multi-user operating system is at the bottom layer of the system. All operating systems have a data move function from one disk file to another. In a heterogeneous system however, this function needs to be generalized to including moving files from one file server to another and should be put at a higher level in the hierarchy. That is, general file moving is not a primitive but a high level function. Conventional localized file moving is of course a primitive function and is the degenerate case of general file moving.

3.4. *Automatic data conversion*

The most important aspect of this approach is to minimize the amount of data movement in the system. However, when data is moved, it will be automatically converted from one format to another. For example, if data is moved from a word serial MIMD or vector machine environment to a bit serial computer, the data must be "corner turned". Data reformatting would be handled automatically by the OS and application languages just as float to fixed conversion is handled in conventional languages.

3.5. *Metacompilers*

The concept of developing a metacompiler for a heterogeneous group of computers is very enchanting, but very difficult. The efforts from the area of vectorizing compilers might be a first step, but they emphasize transforming code designed for one class of machines and transforming it to execute on another closely related class. Furthermore, current compilers make no effort to attempt to determine the "best" machine for execution and the conventional analysis techniques for converting code to flow graphs is slow and may not be the most effective approach. That is, with the current technology, it should be possible to analyze a code and distribute it among a suite of machines, but a sequential algorithm cannot be analyzed and replaced by a new parallel algorithm.

The automatic detection of parallelism is basically limited to nested loops in the initial code. For example in Linpack, vectorization can be used to optimize the inner most loops, but searching for idioms such as finding the maximum value of a vector and replacing it with a (SIMD) parallel maxval function is much more complex because of the variety of ways in which the function can be expressed. Current technology calls for the use of "patterns". A different pattern must be used for each possible realization of the function. This is an *ad hoc* approach and is not a suitable solution. The traditional analysis techniques may not be applicable. For example, traditional data flow analysis provides reaching definitions, available expressions and loop optimization information. This information may be very useful for planning the top level virtual machine organization; however, in a data parallel language this information is not normally useful. That is, in a conventional sequential language, flow of control is based on the relationship between scalar variables. However, in a (data) parallel language control is determined locally by datum specific logic. Indeed, this kind of control is equivalent to using arrays of variables in a sequential computer. As always, pointers and arrays create situations which are very difficult to handle using traditional analysis techniques.

4. CODE EXECUTION MODELLING

In a heterogeneous supercomputer environment, it is imperative to dynamically assign jobs to computers in such a way as to optimize either throughput or execution speed (or both, ideally). This includes dividing jobs into subtasks which execute optionally on VHAMs at all levels. It is not sufficient to assign tasks on a first come first served basis or some simple priority scheme. Optimal results can only be achieved by code execution modeling. Code execution modeling includes the ability to accurately predict how a code data set combination will execute on a VHAM. It incorporates components of benchmarking, code profiling and data profiling. This section provides background on these topics and describes a prototype system.

4.1. *Benchmarking*

Benchmarks are commonly used to test and evaluate codes, algorithms and machines, and have long been used, especially for HPC. Nevertheless there are fundamental differences in the underlying uses of benchmarks that often lead to semantic misinterpretation and confusion. For example, scientific users of HPC often want results from benchmarks as an indicator of how their existing code will run on new machines. Designers of new algorithms or machines often want to know the future potential, including particularly the result of radical redesign of code and algorithm. The term "peak performance" has often been reserved for this last concept, even though sustained code performance seldom comes close (although intelligent assembley language coding can sometimes lead to sustained performances several times faster than "peak"). One of the more interesting recent approaches was that of Gustafson *et al.*[6] in which they proposed a scalable methodology (SLALOM) in which the amount of work done in a fixed time is the key measure of performance, rather than the amount of time to do fixed work. Furthermore the SLALOM approach emphasizes the need to solve the problem, not run a particular code (which has been written in a style inherently favoring one type of architecture). We propose some refinement of terminology in order to expand on the differing levels of benchmarking by examining several situations:

(a) Consider the case of large physical simulation code, e.g. climate modeling. It may be impractical to make radical changes in the code or algorithms in the near or intermediate future, benchmarks needed.

(b) Let "Benchmark Set" be reserved for codes with little or no "tuning", e.g. the LINPACK test set.

(c) Let "Predictor Set" be reserved for codes in which significant rewriting of code, including assembly language, is permitted, e.g. the PERFECT Club suite. In the case of new vendor products, this would offer the vendors the challenge (and opportunity) to do the best they can, on real problems.

(d) Let "Subproblem Set" be reserved for cases in which we change the algorithm, e.g. moving from one kind of sort to another, as might happen in optimally moving from one type of architecture to another.

(e) Let "Problem Set" be reserved for cases in which the whole approach might be changed, e.g. Potter[5] has clearly demonstrated that in moving from von Neumann machines to SIMD machines, searching, in an associative computing environment, can be more effective than traditional sorting. The specification for a Problem Set might merely consist of work problems that need to be computed in any manner.

A distinction between benchmarking and code profiling also needs to be made. Benchmarking is the process of establishing a suite of codes to model a "typical" workload so that different architectures and machines can be compared. However, most benchmarks have been developed for a traditional sequential machine environment. Vector machines have been developed to optimize code written for this type of environment. On the other hand SIMDs were developed as an independent architecture. In a heterogeneous HPC system, a more rigorous general purpose approach for comparing computers is needed so that computing resources can be assigned dynamically. Freund and Peterson[7] have proposed a formulation for determining the best task assignments in a DH-HPC environment. Dynamic assignment requires that the performance of the currently available computers on waiting jobs can be predicted in such a way that they can be meaningfully compared so that an optimal assignment can be achieved.

Code profiling is the technique of analyzing programs to determine how they may be optimized for execution on any given VHAM. In a heterogeneous supercomputer environment, code profiling can be combined with benchmarking to accomplish code execution modeling.

This section on code execution modeling, defined below, is divided into two subsections: throughput prediction and data profiling. The first proposes an approach for code profiling including a set of atomic commonly used parallel operations which can be easily benchmarked and then combined into more complex formulations to not only predict the time of execution for a piece of code, but to also provide an overall estimate of throughput for an entire DH-HPC system. An important aspect of this work is the ability to predict future performance; and while the approached described below can be laborious, it is intended that the modeling be automated using techniques developed for conventional vectorizing compilers.

4.1.1. *Throughput prediction.* This paper hypothesizes that an important class of codes can be modeled as alternating sequences of scalar and basic parallel operations and that these codes can be meaningfully compared on vector and SIMD machines. These basic sequences can be combined in useful ways to model the operation of the code. The basic sequences in turn are made of component operations which can be combined to produce an estimate of the throughput of a machine for the sequence. Scalar sequences are assumed to consist of unit operations, so that the throughput for a scalar sequence is just the reciprocal of the number of scalar operations times the scalar execution speed.

Vector sequences are assumed to be composed of VECOPS.[8] The VECOPS benchmarks are a set of vector operations which are frequently used by physical scientists in their work. VECOPS can be combined using the equations developed below to produce an estimate of the throughput for the vector sequence. Accordingly, two equations have been generated to predict the throughput of SIMD and vector machines. For SIMD machines, let v be the vector length, n the number of processors, r the quoted (maximum) rate for the arithmetic operation and t the resultant throughput; then

$$t = \frac{vr}{\left\lceil \frac{v}{n} \right\rceil n}. \tag{1}$$

If a compound VECOP operation is being performed (i.e. a vector add and multiply or, SAXPY), the combined rate, r can be calculated by the sum of resistances formula. For example, for two operations r_1 and r_2, the combined throughput, r_t, is:

$$\frac{1}{r_t} = \frac{1}{r_1} + \frac{1}{r_2} \quad \text{or } r_t = \frac{r_1 r_2}{r_1 + r_2}. \tag{2}$$

For n operations, this generalizes to:

$$\frac{1}{r_t} = \sum_{i=1}^{n} \frac{1}{r_i} \quad \text{or } r_t = \frac{\prod_{i=1}^{n} r_i}{\sum_{j=1}^{n} \prod_{i \neq j}^{n} r_i}. \tag{3}$$

The r_t calculated above can be used in the formula for SIMD processors to determine the throughput for the combined VECOP operation.

The peak throughput for vector machines often quoted is calculated by multiplying the basic cycle time by the number of arithmetic units in a processor. However, it is not always possible to make full use of all the units. For example, if a processor has two units, a vector multiplying can only be executed at half the quoted rate because only one of the units is effectively used. On the other hand, a vector multiply and add will execute at the full rate. Another factor is pipeline setup time. This factor must be applied on every reload of the vector registers. For vector machines, let v be the vector length, r the quoted (maximum) rate, u the number of units per processor, u_a the number of units used by the operation, p the pipeline setup time, t the resultant throughput rate, and l the length of the vector registers; then

$$t = \frac{v}{\frac{v}{r\frac{u_a}{u}} + p\left\lceil \frac{v}{l} \right\rceil} = \frac{vru_a}{vu + pru_a\left\lceil \frac{v}{l} \right\rceil} \tag{4}$$

On a vector machine the throughput raises asymptotically to the maximum rate very quickly. On a SIMD machine, the throughput rises linearly until the size of the machine (i.e. the number of PEs) is exceeded then falls to the average rate reflecting the average of the full vector and the nearly empty one. It rises linearly again until the array size is exceeded again and then falls to 2/3, etc. (as SIMD machines have a relatively slow cycle time, the throughput is low when the machine is partially loaded).

These above equations answer the questions: given a data set with vectors of a specific size, which is the better machine for execution? Given that a code, $\mathscr{C}$, can be modeled by an alternating sequence of strings of scalar instructions followed by strings of parallel instructions, then $\mathscr{C}$ can be represented by the following sequence:

$$\mathscr{C} \rightarrow w_{s_1} s_1 w_{p_1} p_1 w_{s_2} s_2 w_{p_2} p_2 \ldots, \tag{5}$$

where w_i is a weight representing the number of operations in each list, s_i is the quoted throughput for scalar operations and p_i is the calculated throughput for the parallel sequence. Then the throughput for the entire sequence can be calculated from the formula:

$$t = \frac{1}{\sum_i (w_{s_i} s_i + w_{p_i} p_i)}. \tag{6}$$

If no MIMD parallelism is present, this reduces to:

$$t = \frac{1}{ws + \sum_i w_{p_i} p_i}. \tag{7}$$

The basic tenants of this model were tested using the CONVEX and DAP computers in the NOSC Superconcurrancy Laboratory. The CONVEX C-210 has a quoted peak rate of 50 mflops and the DAP 510C has a peak rate of 140 mflops for 1024 PEs.

4.1.2. *Data profiling*. An important component of code execution modeling is data profiling. In a general purpose heterogeneous environment, where many machines can perform the same task, such as FFTs, convolution or Gaussian elimination, the question is which machine can do the best job (i.e. execute the fastest) on the specific data set

As an example of data profiling in a HAsP environment, consider the matrix multiply. Let **A** be an $I \times J$ matrix, (a_{ij}), and **B** a $J \times K$, (b_{jk}). The product **C**, is an $I \times K$ matrix, (c_{ik}), i.e.

$$c_{ik} = \sum_{j=1}^{J} a_{ij} b_{jk} \tag{8}$$

In order to compute the $I \times K$ terms, (c_{ik}), we would need a triple loop of the form (assuming appropriate initialization):

```
for $l_1 = 1$ to $L_1$
  for $l_2 = 1$ to $L_2$
    for $l_3 = 1$ to $L_3$
      $c_{ik} = c_{ik} + a_{ij} b_{jk}$
    end
  end
end.
```

The Ls are, of course, the ranges. I, J and K, with the ls being their corresponding indices, i, j and k. Mathematically, it does not matter which of the six possible ways these are computed. However, Dongarra *et al.*[9] have clearly demonstrated that these arrangements can be significantly different compute times when computed on a global memory, vector machine. Since there are several factors that enter into these performance differences, it is very much the concern of the programmer to render the order of the loops optimal depending on the specific circumstances. In the HAsP language, this is not necessary, since the data profiling does this automatically.

To understand how this is done, let us look again at the schematic code above. In the two (of the six cases) in which J (and its associate index, j) form the inner loop, the c_{ik} is a scalar (constant) for the purposes of the inner loop and this is called an SDOT type of operation; the two vectors (the a_{ij} and the b_{jk}) are multiplied component-wise and then each component multiplicand is added to the constant c_{ik}. In the other cases, i.e. where J is not the innermost range, we have the SAXPY family of operations, in which either the a_{ij} or the b_{jk} are a scalar for the purposes of inner loop. In these cases, we have a scalar (either a_{ij} or b_{jk}) multiplied to each component of the other element, a vector in the inner loop, and then each component multiplicand is added, component-wise, to the vector, c_{ik}. As mentioned above, a number of (occasionally conflicting) factors determine the right order to perform the loops. For example, it is generally better (in FORTRAN) to have the innermost loop on the leftmost index so as to avoid non-unit stride through memory and the increased likelihood of bank conflict this brings. It is also usually more efficient to have the innermost loop on the longest range. SAXPY is generally better for short and medium length vectors whereas SDOT is better for longer vectors (at least one reason for this is that SDOT requires summing up the multiplicand terms which usually requires a scalar loop; however, for long vectors this is dwarfed by the reduced operation count by SDOT). It should be noted that for a SIMD machine, these factors are largely irrelevant. In each of the six cases we would store the components of the vectors (as dictated by inner loop) at each node and broadcast the scalar value. From the point of view of a SIMD machine, it matters not whether the multiply is the scalar broadcast value (SAXPY) or not (SDOT). All six cases are essentially the same.

5. CONCLUSION

The HAsP approach outlined here is intended to provide a flexible, comprehensive paradigm for computation on heterogeneous systems of supercomputers. The paradigm is applicable to all levels of computing from single heterogeneous computers to homogeneous local area networks on up to large nation wide heterogeneous networks. The system is designed to minimize data movement and communication overhead and therefore maximize throughput and execution speed. The initial formulas for code execution modeling have been developed and verified. The concept that data as well as code must be profiled was developed and verified by experiment. The next aim is to develop a prototype HAsP system with two or three levels of VHAM in a heterogeneous environment.

Acknowledgements—This work is sponsored by the Office of Naval Technology, the U.S. Navy-ASEE Summer Fellowship Program, Kent State University and the Naval Command, Control and Ocean Surveillance Center.

REFERENCES

1. R. F. Freund, "SuperC or distributed heterogeneous HPC", *Computing Systems in Engineering* **2**(4), 349–355 (1991).
2. R. F. Freund, J. L. Potter and H. J. Siegel, "Avoiding unnatural acts", *Supercomputing Review*, in press.
3. M. Arango, D. Berndt, N. Carriero, D. Gelertner and D. Gilmore, "Adventures with network Linda", *Supercomputing Review*, 42–46, October (1990).
4. G. Agha, *Actors*, The MIT Press, Cambridge, Massachusetts, 1986.
5. J. L. Potter, *Associative Computing*, Plenum Press, New York, 1992.
6. J. Gustafson, D. Rover, S. Elbert and M. Carter, "Slalom update", *Superconductivity Review*, 56-61, March (1991).
7. R. F. Freund and L. J. Peterson, "If the 'network is the computer', then...", *Supercomputing Review*, July (1991).
8. O. M. Lubeck, "Supercomputer performance: the theory, practice, and results", *Advances in Computers* **27**, 309 (1988).
9. J. J. Dongarra, F. G. Gustafson and A. Karp, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine", *SIAM Review* **26**(1), 91–111 (1984).