New Jersey Institute of Technology
Digital Commons @ NJIT

Theses | Theses and Dissertations

Fall 1993

Concurrent use of two programming tools for heterogeneous supercomputers

Javier G. Vasquez
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Computer Sciences Commons

Recommended Citation
Vasquez, Javier G., "Concurrent use of two programming tools for heterogeneous supercomputers" (1993). *Theses*. 1211.
https://digitalcommons.njit.edu/theses/1211

This Thesis is brought to you for free and open access by the Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science &
Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

CONCURRENT USE OF TWO PROGRAMMING TOOLS FOR HETEROGENEOUS SUPERCOMPUTERS

by
Javier G. Vasquez

In this thesis, a demostration of the heterogeneous use of two programming paradigms for heterogeneous computing called Cluster-M and HAsC is presented. Both paradigms can efficiently support heterogeneous networks by preserving a level of abstraction which does not include any architecture mapping details. Furthermore, they are both machine independent and hence are scalable. Unlike, almost all existing heterogeneous orchestration tools which are MIMD based, HAsC is based on the fundamental concepts of SIMD associative computing. HAsC models a heterogeneous network as a coarse grained associative computer and is designed to optimize the execution of problems with large ratios of computations to instructions. Ease of programming and execution speed, not the utilization of idle resources are the primary goals of HAsC. On the other hand, Cluster-M is a generic technique that can be applied to both coarse grained as well as fine grained networks. Cluster-M provides an environment for porting various tasks onto the machines in a heterogeneous suite such that resources utilization is maximized and the overall execution time is minimized. An illustration of how these two paradigms can be used together to provide an efficient medium for heterogeneous programming is included. Finally, their scalability is discussed.

CONCURRENT USE OF TWO PROGRAMMING TOOLS FOR HETEROGENEOUS SUPERCOMPUTERS

by
Javier G. Vasquez

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer and Information Science

January 1994

Blank Page

APPROVAL PAGE

CONCURRENT USE OF TWO PROGRAMMING TOOLS
FOR HETEROGENEOUS SUPERCOMPUTERS

Javier G. Vasquez

Dr. Mary M. Eshaghian, Thesis Advisor | Date
Assistant Professor of Computer and Information Science, NJIT

Dr. Daniel Y. Chao, Committee Member | Date
Assistant Professor of Computer and Information Science, NJIT

Dr. David Wang, Committee Member | Date
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Javier G. Vasquez

Degree: Master of Science in Computer Science

Date: January 1994

Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1994

- Bachelor of Science in Computer Science
  Jersey City State College, Jersey City, New Jersey , 1989

Major: Computer Science

This thesis is dedicated to my parents and my family

ACKNOWLEDGMENT

The author wishes to express his sincere gratitude to his supervisor, Dr. Mary M. Eshaghian for her guidance, friendship, and moral support throughout this research.

Special thanks to Dr. Daniel Chao and Dr. David Wang for serving as members of the committee. The author is grateful to the Department of Computer and Information Science for partially funding this research.

The author appreciates the timely help and suggestions from the project group team members Phil Chen, Ajitha Gadangi and Ying-Chieh Jay Wu.

The author is very much grateful to Lisa A. Ryan and her family for their moral support throughout the graduate studies time. The author also wishes to thank Annette Damiano for her professional comments in finalizing this thesis.

TABLE OF CONTENTS

Chapter Page

1 INTRODUCTION AND BACKGROUND 1

2 CLUSTER-M MODEL 4

2.1 Cluster-M Specifications 4

2.2 Cluster-M Representations 7

2.3 Mapping Specifications to Representations 8

2.3.1 A mapping methodology 10

2.3.2 An Example 11

3 HETEROGENEOUS ASSOCIATIVE COMPUTING 13

3.1 Instruction Execution 15

3.2 HAsC Administration 16

3.3 HAsC Instruction Set 18

3.4 Associative Instruction Levels 19

4 CONCURRENT USE OF CLUSTER-M AND HASC 22

4.1 Switching between Cluster-M and HAsC 22

4.2 Cluster-M aided HAsC 23

5 SCALABILITY 24

5.1 Homogeneous Case 24

5.2 Heterogeneous Case 26

5.2.1 Fundamental Theorem of Scalability 26

5.3 Scalability of HAsC and Cluster-M 28

6 CONCLUSION AND FURTHER RESEARCH 30

REFERENCES 31

LIST OF FIGURES

Figure **Page**

2.1 Cluster-M Specification graph of a unary operation on an array of 5

2.2 Cluster-M Specification of associative binary operation. 5

2.3 Cluster-M Representation of n-cube of size 8. 8

2.4 Cluster-M Representation of a completely connected system of size 8. . . 9

2.5 Cluster-M Representation of an arbitrarily connected system of size 8. . . 9

2.6 Mapping onto n-cube of size 8 10

2.7 An example for mapping algorithm 12

3.1 Associative Configuration of a Network. 14

3.2 A Heterogeneous Network as a SIMD. 14

3.3 Instruction Synchronization 20

4.1 Switching between Cluster-M and HAsC 23

4.2 Cluster-M aided HAsC computation within HAsC nodes 23

5.1 Hierarchical breakdown of a task 27

CHAPTER 1

INTRODUCTION AND BACKGROUND

Heterogeneous Computing (HC)[16, 14] provides an environment where a parallel application is executed utilizing a number of autonomous computers communicating over an intelligent network, and offering more than one type of parallelism. This approach aims at providing high performance by executing portions of code on suitable machines offering similar types of parallelism. The hardware and software requirements of HC can be classified into three layers: network layer, communication layer, and intelligent layer [20]. The network layer deals with the physical aspects of interconnecting the autonomous high performance machines in the system. This includes low level network protocols and machine interfaces. The communication layer provides a uniform system-wide communication mechanism operating above native operating systems to facilitate the exchange of information between different machines. The intelligent layer provides system-wide tools that insure proper and efficient execution of tasks using the heterogeneous suite of computers. The services provided by this layer include language support, task decomposition, mapping and scheduling.

A number of existing parallel programming tools developed for homogeneous systems may be used in the intelligent layer, but may not be suitable for the heterogeneous systems. These tools can be classified into two categories; machine specific and machine independent. Machine specific tools such as Linda [5] and Poker [21] are only suitable for the corresponding architectures they are designed for, and therefore not generic enough to support the heterogeneous networks. For example, Linda [5] is a parallel programming tool developed for shared memory architectures. The tuple space defined in Linda is a logically shared data structuring memory

mechanism. Tuple space holds two kinds of tuples: process tuples which are under active evaluation, and data tuples which are passive. Process tuples execute simultaneously, and exchange data by generating, reading, and consuming data tuples. Once a program is written based on Linda, each step must get implemented using the underlying architecture. However, it is difficult to implement Linda on architectures not supporting shared memory structure.

Machine independent programming tools can be further categorized into two groups, with respect to how the mapping of the problem tasks is done onto the target architectures. The first group uses a library of pre-defined routines for mapping [1, 23]. This may not be suitable for HC systems due to the limitation on the number of mapping techniques stored and available in the library. In the second group, the mapping is determined online based on graph matching technique. The mapping problem here is the same as the classic one defined and studied by several researchers over the years [22, 3, 17, 4, 8, 18]. The input to the mapping problem is two graphs. The first graph is called the problem graph which is similar to the data flow representation of the execution process, where each node is a computation task and edges represent dependency and flow of data. The second graph is called the system graph which is a trivial representation of the underlying architecture. The mapping problem is defined as the matching of these two graphs such that the overall execution time is minimized. This problem has been proven to be computationally equivalent to the graph isomorphism problem and hence is an NP-complete optimization problem [3]. Tools that use this approach are not time efficient to be used in heterogeneous computing.

To reduce the complexity of the mapping problem, a number of approaches such as graph contraction and clustering have been studied [7, 2, 15, 24, 25, 18]. However, in all these graph matching based techniques, still the entire problem graph is considered against the entire system graph, which results in an embedded

huge time complexity. In this thesis, we propose to use Cluster-M programming paradigm for heterogeneous computing. Cluster-M, introduced recently in [9], has a mapping module which does multi-level clustering on the problem graph as well as the system graph. Also, presented in this thesis is HAsC programming paradigm [19], which models a heterogeneous network as coarse grained associative computer and is designed to optimize the execution of problems with large ratios of computations to instructions. Ease of programming and execution speed, not the utilization of idle resources are the primary goals of HAsC. On the other hand, Cluster-M is a generic technique that can be applied to both coarse grained as well as fine grained networks. Cluster-M provides an environment for porting various tasks onto the machines in a heterogeneous suite such that resources utilization is maximized and the overall execution time is minimized. We illustrate how these two paradigms can be used together to provide an efficient medium for heterogeneous programming.

The rest of the thesis is organized as follows. In chapter 2, Cluster-M components and mapping methodology are presented. Presentation of HAsC in chapter 3. Introducion of the concurrent use of HAsC and Cluster-M in chapter 4. The definitions of scalability for hardware, tasks, and software are presented in chapter 5. The conclusion of this thesis is described in chapter 6.

CHAPTER 2

CLUSTER-M MODEL

Cluster-M is a novel parallel programming model which facilitates the effchaptericient design of highly portable software. Cluster-M has three main components: Cluster-M Specifications, Cluster-M Representations and Cluster-M mapping module [9, 11, 10]. Cluster-M Specifications are machine independent algorithms represented in a multi-layered problem graph, such that each layer represents concurrent computations. A Cluster-M Representation on the other hand, represents a multi-layered partitioning of a system graph corresponding to the topology of the underlying architecture or heterogeneous network. The mapping module then generates an efficient mapping of the Specification graph onto the Representation graph. Using Cluster-M, portable and scalable software can be developed.

2.1 Cluster-M Specifications

A Cluster-M Specification of a problem is a high level machine-independent program that specifies the computation and communication requirements of a solution to a given problem. A Cluster-M Specification can be translated into a graph consisting of multiple levels of clustering. In each level, there is a number of clusters representing concurrent computations. Clusters are merged when there is a need for communication among concurrent tasks. For example, if all n elements of an array are to be squared, each element is placed in a cluster, then the Cluster-M specification would state:

For all n clusters, square the contents.

Figure 2.1 Cluster-M Specification graph of a unary operation on an array of size n.

Note, that since no communication is necessary, there is only one level in the Cluster-M Specification graph as shown in Figure 2.1. The mapping of this Specification to any architecture having n processors would be identical.

The basic operations on the clusters and their contained elements are performed by a set of constructs which form an integral part of the Cluster-M model. For a complete listing and description of these constructs which are essential for writing Cluster-M Specifications, refer to [11, 10]. All these constructs have been implemented in PCN [10, 12]. Below we show an example for computing the associative binary operation $*$ of N elements of vector A, using the constructs implemented in PCN. The resulting Cluster-M specification will be as follows, where $CMAKE$, $CMERGE$ and CBI are Cluster-M specification constructs. The Cluster-M Specification graph of this example is shown in Figure 2.2.

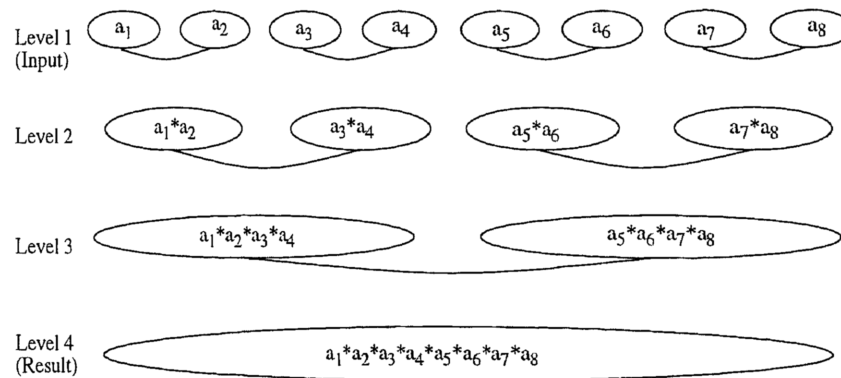

Figure 2.2 Cluster-M Specification of associative binary operation.

```
ASSOC_BIN(op, N, A, Z) /* op: operation, Z: return value */
int N, A[ ];
{ ; lvl = 0,
```

```
    make_tuple(N, cluster),
    { ; i over 0 .. N-1 ::
        { ; CMAKE(lvl, [A[i]], c),
            cluster[i] = c
        }
    } ,
    Binary_Op(cluster, N, op, Z)
}


Binary_Op(X, N, op, B)
int N, n;
{ ? N > 1 -> { ; n := N / 2,
                 make_tuple(n, Y),
                 { ; i over 0 .. n-1 ::
                     { ; BI_MERGE(op, X[2 * i], X[2 * i + 1], Z),
                         Y[i] = Z
                     }
                 } ,
                 Binary_Op(Y, n, op, B)
               } ,
    default -> B = X
}


BI_MERGE(op, X1, X2, M)
int e;
{ ; CBI(op, X1, 1, X2, 1, e),
    CMERGE(X1, X2, [e], M)
}
```

The above constructs have been implemented using PCN (Program Composition Notation). PCN is a system for developing and executing parallel programs. It comprises of a high-level programming language with C-like syntax, tools for developing and debugging programs in this language, and interfaces to Fortran and C allowing the reuse of existing code in multilingual parallel programs. Programs develop using PCN are porable across many different workstations, networks, parallel computers. The code portability aspect of PCN makes it suitable as an implementation medium for Cluster-M.

2.2 Cluster-M Representations

For every architecture, at least one corresponding Cluster-M Representation graph can be constructed. Cluster-M Representation of an architecture is a multi-level nested clustering of processors. To construct a Cluster-M Representation, initially, every processor forms a cluster, then clusters which are completely connected are merged to form a new cluster. This is continued until no more merging is possible. In other words, at level LVL of clustering, there are multiple clusters such that each cluster contains a collection of clusters from level $LVL - 1$ which form a clique. The highest level consists of only one cluster, if there exists a connecting sequence of communication channels between any two processors of the system. A Cluster-M Representation is said to be *complete* if it contains all the communication channels and all the processors of the underlying architecture. For example, the Cluster-M Representation of the n-cube architecture is as follows: At the lowest level 1, every processor belongs to a cluster which contains just it self. At level n, every two processors (clusters) which are connected are merged into the same cluster. At level 2, clusters of previous level which are connected belong to the same cluster, and so on until level $n + 1$. The complete Cluster-M Representation of a 3-cube, a completely

connected system of size 8, and of a system with arbitrary interconnections are shown in Figures 2.3, 2.4 and 2.5, respectively.

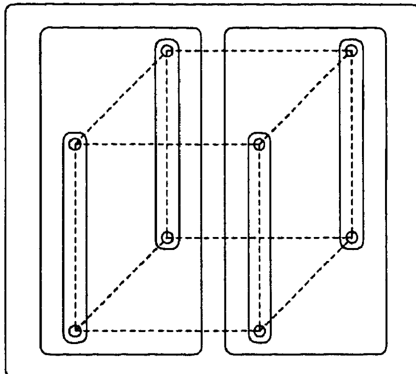

Figure 2.3 Cluster-M Representation of n-cube of size 8.

An algorithm for generating a Cluster-M Representation for any given architecture has been presented and implemented in [10]. The algorithm has a running complexity of $O(N^3)$ where N is the number of processors.

2.3 Mapping Specifications to Representations

The most challenging task in the Cluster-M model is the mapping of the Specifications onto the fixed Cluster-M Representations of various architectures. Although in some cases this may appear simple, the mapping of certain Specifications may be non-trivial. For example, consider the associative binary operation example of the last section. We assume that it will take one time unit for a single communication along a link. Its mapping onto a 3-cube is shown in Figure 2.6 and is straight forward and can be done in 3 steps.

On the other hand, to map the same onto a ring of size 8 will lead to a greater time complexity since there are not enough communication channels available to support the communication request specified in the Cluster-M Specification. Similarly, there is going to be a slow down if there are not enough processors in the

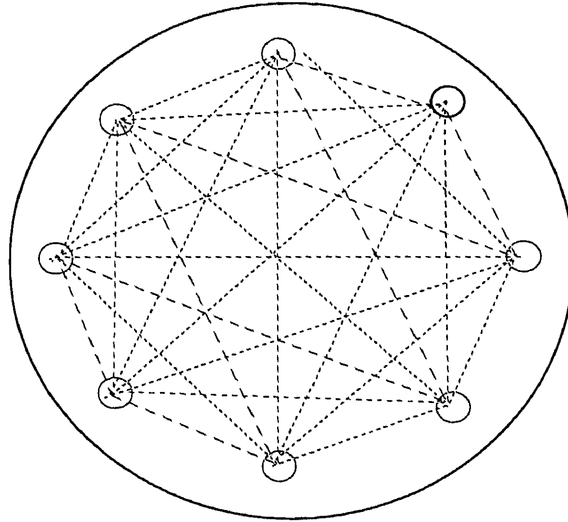

Figure 2.4 Cluster-M Representation of a completely connected system of size 8.

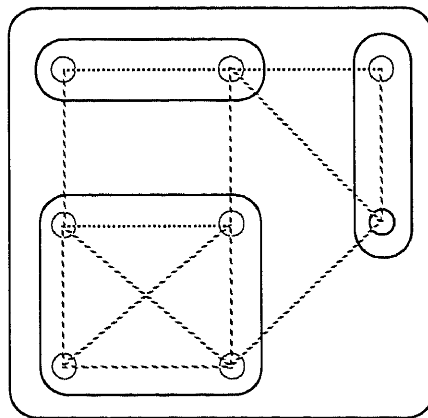

Figure 2.5 Cluster-M Representation of an arbitrarily connected system of size 8.

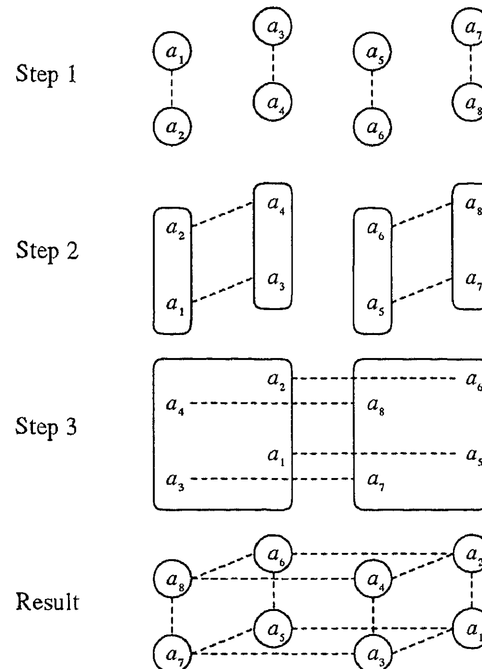

Figure 2.6 Mapping onto n-cube of size 8

Representation available as specified in the Specification. For example, the same problem described above, will take at least twice as much time if it to be mapped on a Cluster-M Representation having half the number of processors. Mismatch of the number and structure of clustering in Cluster-M Specification versus Cluster-M Representation may lead to a significant slow down in performance. In the following section we present an efficient methodology for mapping an arbitrary Specification to an arbitrary Representation.

2.3.1 A mapping methodology

The Cluster-M paradigm simplifies the mapping process by formulating the problem in the form of Cluster-M problem Specification (a layered problem graph) emphasizing its computation and communication requirements independently from the target architecture. Similarly, the Cluster-M Representation of the system emphasizes the topology of the target multi-processor system (a layered system graph). Once

both, the Cluster-M problem Specification and system Representation, are obtained the mapping process proceeds as follows:

Start from the root of Cluster-M Specification. At level i, there is a number of clusters. Each cluster has a size K which is defined by the cumulative sum of the number of computations involved in all its nested subclusters. On the other hand, in Cluster-M Representation, there is a collection of subclusters as part of a Cluster-M Representation of a single connected system. We next look for a number of clusters in the Representation to match the number of clusters at the ith level of the Specification. Furthermore, we select the clusters such that the size of the corresponding pair matches. The details of this algorithm are beyond the scope of this paper. For more information, see [6]. As part of the proposed algorithm, several graph theoretic techniques have been used. In the next section, we give an example to illustrate the functionality of the mapping module.

2.3.2 An Example

In this section, we present a complete example to illustrate the Cluster-M mapping methodology presented above.

Figure 2.7 shows the mapping from a Cluster-M Specification to Representation. First, two clusters at the top level of Specification are mapped onto two clusters of Representation. The Specification cluster of size 5 is mapped onto the Representation cluster of the same size, however the Specification cluster of size 4 has to be mapped onto the Representation cluster of size 3 since this is the closest matching of sizes. Then the same procedure is applied for the clusters at the lower level of Specification. As shown in step 2 in Figure 2.7, Specification cluster a is mapped onto Representation cluster H, which is a single processor. In step 3, Specification clusters b, e, f, g, h and i are mapped onto corresponding processors. Finally in step 4, Specification cluster c and d are both mapped onto processor F.

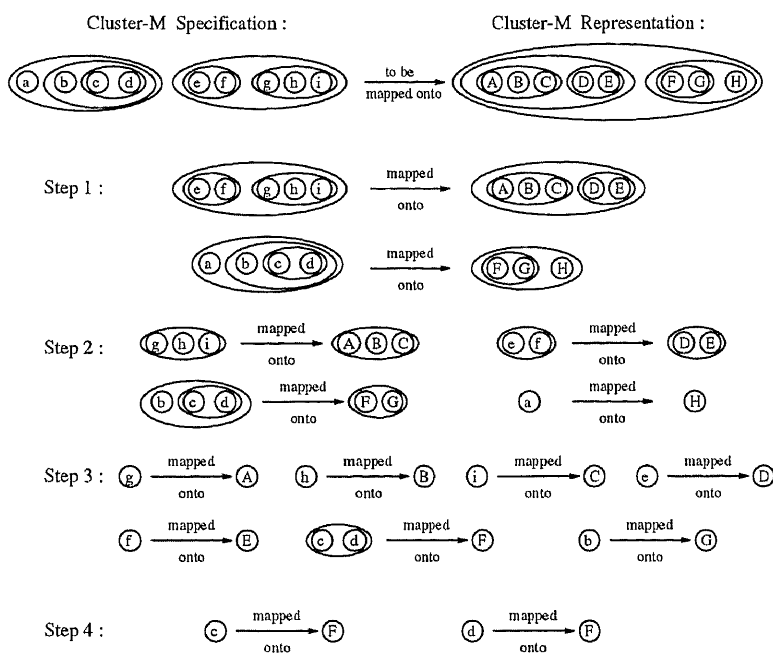

Figure 2.7 An example for mapping algorithm

CHAPTER 3

HETEROGENEOUS ASSOCIATIVE COMPUTING

Heterogeneous Associative Computing (HAsC) models a heterogeneous network as a coarse grained associative computer. It assumes that the network is organized into a relatively small number of very powerful nodes. Basically, each node is a supercomputer architecture (vector, SIMD, MIMD, etc). Thus each node of the network provides a unique computational capability. There may be more than one node of a specific type in the case that special properties are present. For example, one SIMD node may be specialized for associative processing, a second SIMD node may contain a very powerful internal network configuration.

Figure 3.1 illustrates the logical similarity of an associative machine and a heterogeneous network. In particular, a disk- computer node on a network can be compared to an associative memory-PE cell. That is, effectively, the node's computer is dedicated to processing the data on the node's disk(s). The disk-to-machine data transfer rate is much more efficient than the node-to-node transfer rate, just as the memory-to-PE transfers are much faster than PE-to-PE transfers. Note that the SIMD and network diagrams are quite different from the shared memory MIMD models. The shared memory configurations emphasize the concept that all data is equally accessible from all processors. This is not the case in a heterogeneous network.

HAsC is "layered" in that any node in the HAsC network may again be another network. Thus a HAsC node may be a HAsC cell containing more than one computer, or may be a port to another level of computing in the HAsC network. For example, most nodes may contain general purpose computers in addition to a supercomputer, to function as the node's port to the rest of the HAsC network

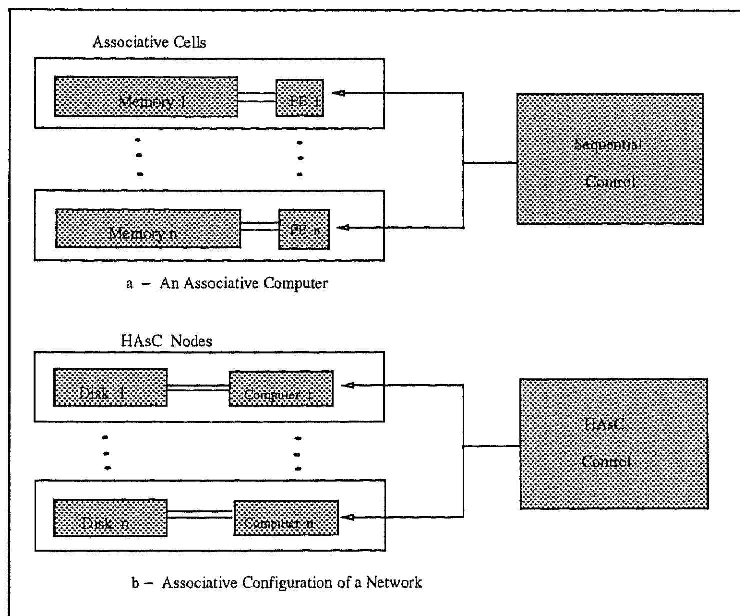

Figure 3.1 Associative Configuration of a Network.

and for file management and other support roles. Figure 3.2 shows a typical HAsC network organization. Such a port, or transponder node will accept a high level command and "translate it" into the commands(s) appropriate for the subnetwork.

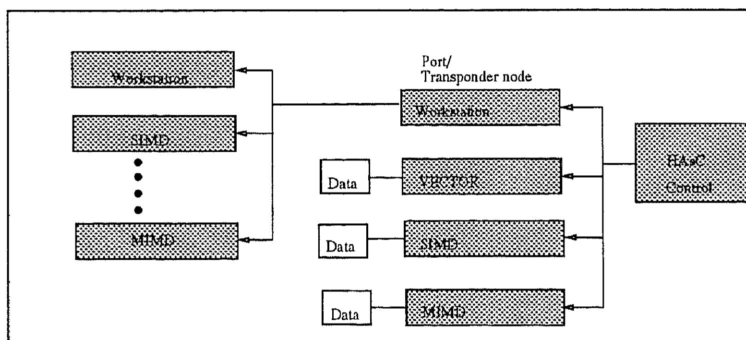

Figure 3.2 A Heterogeneous Network as a SIMD.

Some of the properties of the associative computing paradigm which make it well suited for heterogeneous computing are: i) efficient programming and execution with large data sets and small programs, ii) optimal data placement, iii) scalability, iv) cellular memory allocation, and v) search-process-retrieve synchronism [19].

3.1 Instruction Execution

In conventional machines, instructions are delivered to a CPU and they are executed without question. In HAsC, instructions are broadcast to all of the cells listening to a channel, but each individual 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 *and* data files.

The unification process is borrowed from Artificial Intelligence. Several languages such as Prolog and STRAND [13] incorporate the process. HAsC is different in that it uses unification only at the top level. Thus there is only one unification operation per data file, as opposed to one per record or field. This difference is critical in a heterogeneous network where communication of individual data items would be prohibitively expensive.

If there is a match, the appropriate instruction is initiated. The "instruction" may in turn issue more instructions. Thus control is distributed throughout HAsC. That is, a "program" starts by issuing a command from a control node. If a receiving node receives a command that is in effect a subroutine call, it may become a transponder control 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 subnet as well as to its own network. Thus it is possible for multiple instruction streams to be broadcast simultaneously at several different logical network levels in a HAsC network.

In general, HAsC assumes that data is resident in a cell. As a result, data movement is minimal. However, it is common for one cell to compute a value and broadcast it to other cells. Thus, in general, there is a need to synchronize the arrival of commands and data. There are basically two cases which are handled automatically by the HAsC administrator as a part of the search-process-retrieve protocol.

The normal case is for data to be resident at a cell when the HAsC command arrives. Instruction unification and execution proceeds as described above. HAsC allows data transfers, but protocol insists that the data transfer be complete before any associated commands are broadcast.

The second case involves command parameters. When a command arrives and is unified with resident data at a node, but parameter data is missing, the unified command is stored in a table to wait for the parameter in a synchronism process called a data rendezvous. When parameter data arrives, the rendezvous table is searched for a match. If found, the associated command is executed.

3.2 HAsC Administration

HAsC uses network administrators and execution engines to effect the paradigm. Each HAsC network level has a system administrator and each node in a network has its own local administrator. The local administrator monitors network traffic capturing incoming instructions and checking for illegal commands. It is also responsible for maintaining the local HAsC instruction set.

The administrator receives all incoming HAsC instructions from the local network. It then verifies if each instruction is a legal HAsC instruction. If it is, the administrator puts it in the Execution Engine queue. Otherwise, it attempts to identify the source and makes a report to the system administrator. Repeat offenses cause escalating diagnostic actions as determined by the network administrator.

If a Meta HAsC instruction such as (un)install, (un)extend, or (un)augment, is received, it is processed immediately. The Meta instructions will create, modify and delete HAsC instruction from the local HAsC instruction set respectively. The administrator contains logic which prevents it from installing duplicate HAsC instruction. Meta instructions can also modify local data structure definitions.

Since the instruction set can be dynamically expanded by the users, it is possible for two users to install the same instructions. The node administrator distinguishes between the two instructions by a user *id* and program *id* which is broadcast with every HAsC instruction.

Instructions can be added at several different logical levels: i) system, ii) project, iii) user. Typical systems level instructions would be data move and formatting commands. Project commands would be project oriented. For example, a numerical analysis project would have a matrix multiply and vector-matrix multiply instructions, while a logic programming project might have specialized logic instructions, such as unification. At the user level, one user might specify a SAXPY operation while another might want a dot product. Scalable libraries may exist at any level, but most commonly at the project level.

Each node/cell has an execution engine which controls instruction execution at that node. The execution engine selects the next instruction, makes the bindings specified by instruction unification and causes the instruction to be executed. The execution engine performs the following tasks:

Get Next Unified Instruction
Establish Environment
Save Local Variables
Bind Unified Variables
Execute Unified Instruction
Restore Environment

Instruction execution may take two basic forms. First the instruction may be a HAsC program which is executed in the transponder mode. Second, the instruction may be a library call written in FORTRAN, C, LISP, etc. In this case, the established environment restrictions, produces the proper interface for the appropriate language.

3.3 HAsC Instruction Set

This section defines the nature of the operations, the instruction format and the instruction synchronization classes of the HAsC instruction set.

HAsC is dynamic. As such, it must allow for a dynamic instruction set and data structure modifications. Thus the HAsC *install* meta instruction consist of an associative pattern and a body of code. When it is broadcast to the system, all nodes which successfully unify with the instruction gather the body of code and install it on the local node. The *extend* instruction consists of a pattern and a data definition. Responding nodes add the data definition to the local associations. *Extend* may add a named row or column to an existing association. *Augment* can be used to add an entire new association.

The patterns in these instructions contain administrative data. Such as job id, project id, etc. If the node is not participating in the project or job, then it does not unify and the instruction is not installed or the data definition not extended. *Uninstall*, *unextend* and *unaugment* perform the inverse operations.

Basic to the HAsC philosophy is the concept that data when initially loaded into the system is sent to the appropriate node and never moved. While this would be ideal, there will always be a need to move data from one node to another. Accordingly there are a number of HAsC move commands. Move commands can be divided into intra-association and inter-association instructions. Intra-association instructions are very much like expressions in conventional languages and are not discussed here because of lack of space. Inter-association instructions include file I/O as a special case. Inter-association *moves* must have node identifiers and for I/O, a disk or other peripheral is a legal node.

3.4 Associative Instruction Levels

This section describes a hierarchy of instructions from the highest, most global (least responsive) to the most local (most responsive). HAsC will perform most efficiently if the programs are written using top level commands. The lower the level of command, the more inter-node communication is required. Five levels of instruction coupling are required to implement all of the HAsC statements.

The communication and synchronization are built into the HAsC instruction. There is no need for the programmer to be aware of the degree of instruction communication. The five levels of instructions are presented here to more clearly delineate the relationship between associative and heterogeneous computing.

The highest level of instruction synchronization is pure associative data parallelism and involves the use of the local channel registers only - i.e. there is no global coupling. There are two types of top level instructions: i) ones which execute based on the channel register content only, such as logical and arithmetic expressions and ii) ones which set the channel register. Data parallel logical expressions (associative searchers) can be used to set the channel registers and are "automatically" incorporated into many HAsC statements. Thus a data parallel WHERE consists of only an associative search, followed by a sequence of data parallel expressions. It is a top level instruction. Top level instructions execute in real time and require no global response or communication. Most computation is done at the top level.

Figure 3.3 gives some examples of instruction synchronization. In Figure 3.3, $ is the parallel marker and is read as a plural. That is, A$ is read as As. Result$ is a data parallel pronoun referring to the last performed data parallel computation. "It" is a reduction pronoun referring to the last performed reduction. The top level synchronization shows the programming style for algebraic expressions supported by HAsC.

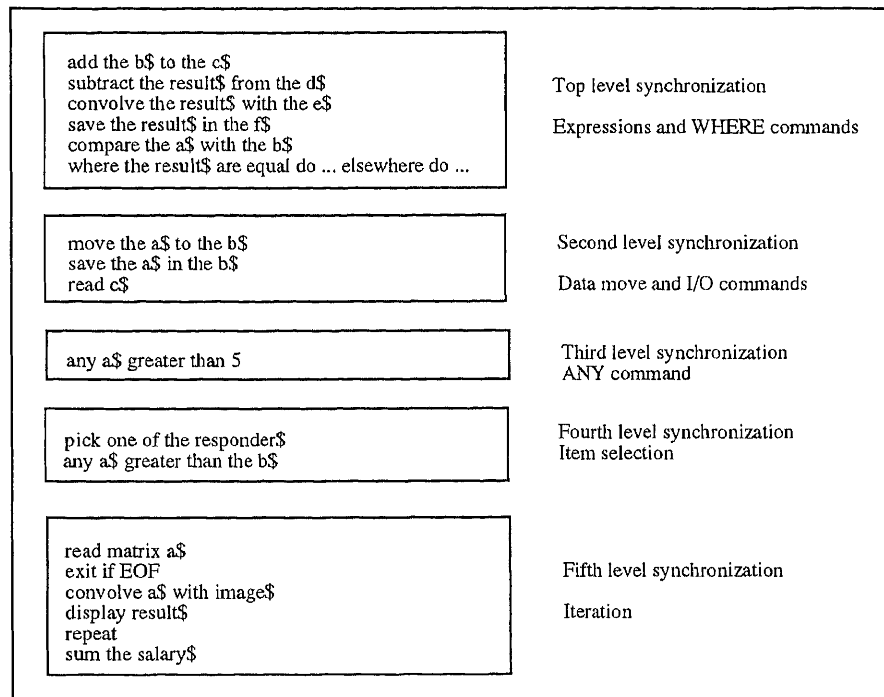

Figure 3.3 Instruction Synchronization

The second level of instruction coupling requires only global synchronism. Prime examples are the data transfer and I/O commands. I/O is always local to the virtual PE, but in general the virtual PE's may be quite different physically and therefore I/O times may vary dramatically requiring synchronization before the next HAsC command is issued. Again, the programmer need not be aware of the synchronization requirements of this class of instructions. The synchronization is automatic. The programmer only recognizes the need for I/O or data movement.

The third level of complexity consist of simple responder commands. These commands require the ORing of the responder results of all PEs (i.e. an OR reduction). On a SIMD this is a single instruction. In HAsC, it is the simplest form of a HAsC reduction communication. The instructions at this level, such as ANY, are used to check for error conditions, or determine whether special case computing needs to be done.

The fourth level is random selection. The HAsC commands in Figure 3.3 at this level consist of an associative search, followed by the selection of a responder by the "first reduction" operation. The data object of the selected responder is broadcast to the entire HAsC network for further processing.

The fifth level is iteration. The only use for iteration at the top level of HAsC is for user interaction. For example, a typical program might be one which allows the user to interactively specify kernels to be convolved with an image and to review the results, as shown in Figure 3.3.

HAsC is a programming paradigm designed to facilitate the utilization of heterogeneous networks. The parallel associative programming techniques are well suited for this purpose.

CHAPTER 4

CONCURRENT USE OF CLUSTER-M AND HASC

As shown in the previous sections, HAsC is most suitable for coarse grain heterogeneous parallel computing. It is to ease programming and increase execution speed, while not taking into account resource utilization. Cluster-M, on the other hand, provides both coarse grain and fine grain mapping in a clustered fashion. It aims at maximizing both execution speed as well as resource utilization. Therefore, both paradigms can be used concurrently to achieve a better overall performance. In the following, we show two possible concurrent use of these two paradigms.

4.1 Switching between Cluster-M and HAsC

Before we run an application task on a HAsC system, we first generate Cluster-M Specifications of that task, which are multi-level clusters preserving information of computation and communication at each step. Since all the clusters of the same level represent concurrent computations at a certain step, therefore this set of clusters can be sent to the HAsC control unit, and then be broadcast to HAsC nodes (Figure 4.1). Each node then decides which clusters out of all the clusters received are most suitable to itself, according to the type of parallelism labeled to each cluster. After all the nodes finish computation of the corresponding clusters, the results are sent back to control unit. Then at the next level clusters are fetched to the control unit to start next step computations. Therefore, there is a switching between Cluster-M and HAsC at each clustering level of Cluster-M Specification.

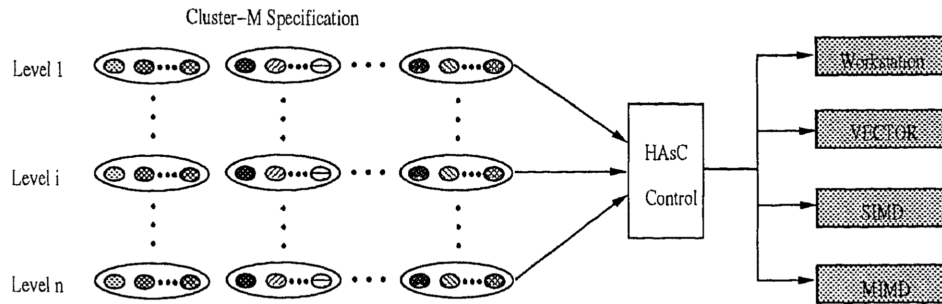

Figure 4.1 Switching between Cluster-M and HAsC

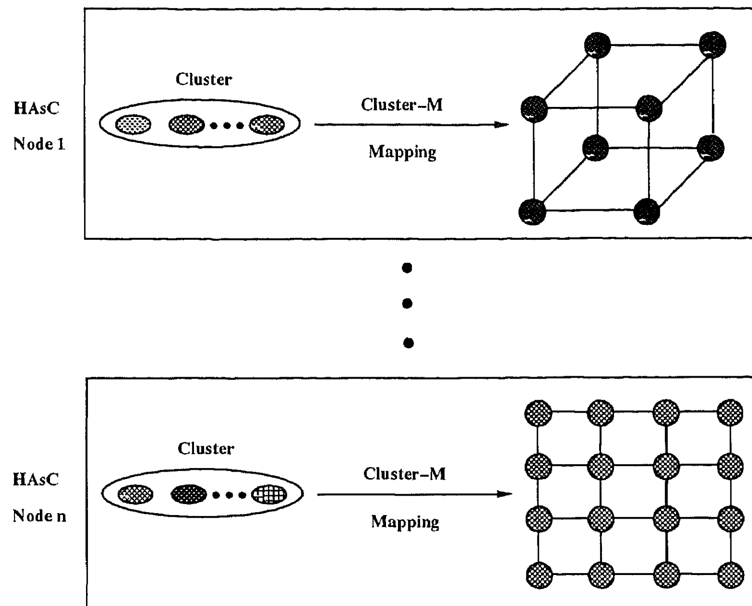

Figure 4.2 Cluster-M aided HAsC computation within HAsC nodes

4.2 Cluster-M aided HAsC

Cluster-M mapping can be applied to HAsC in two ways. First, Cluster-M mapping can be used to decide where the data is to be mapped onto before HAsC computation begins so that the overall execution time is minimized. Secondly, Cluster-M mapping can be used to decide the fine grain mapping of HAsC nodes as shown in figure 4.2.

CHAPTER 5

SCALABILITY

One of the basic issues related to and addressed in both HAsC and Cluster-M, as well as many HPC (High Performance Computing) and MPP (Massively Parallel Processing) schemes, is that of scalability. Scalability is often understood differently by different authors. For our purposes we will consider scalability to refer to hardware, tasks, and software in roughly analogous fashion. In addition scalability may refer to both homogeneous or heterogeneous architectures.

5.1 Homogeneous Case

The homogeneous case refers to multiple machines which are of the same basic architectural type, typically various-sized versions of the same vendor product. For example an eight processor CRAY is a hardware example of "scaled"-up version of a two-processor CRAY.

Definition 1 *We define the hardware scalability function, $\chi(a, b)$, between two homogeneous architectures, a (the larger) and b (the smaller), to be the rational-valued function giving the size multiple of a over b. In the example above, the eight-processor Cray has a $\chi = 4$ over the two-processor.*

Task scalability is more complex. What is typically implied is the ability to take a task (algorithm plus data) executing on a small machine and execute the "same" task on a "scaled"-up machine, using the additional resources of the larger machine, with performance reasonably close to χ. One ambiguity in this concept is what we mean by the "same" task. If it means only the same algorithm, but with possibly different, i.e., larger data, then tasks often "scale", particularly if the

scaling factor of the data size is equal to χ. However, if we follow the definition of task given above, fixed data and algorithm, then tasks often do not scale, even on scaled up homogeneous hardware. To give a simple example, suppose we are computing a pixel-based imagery problem on a SIMD machine in which both the number of pixels and the number of processors is 1K. If we scaled-up to a 16K processor ($\chi = 16$), typically this task would not scale, i.e., it would not be able to exploit the additional 15K processors, and we would get no increased performance. However if our original task had started with a 16K pixel problem, we would typically be able to scale in performance, on the 16K machine over the 1K machine.

Definition 2 *We define task scalability, between two homogeneous architectures, a (the larger) and b (the smaller), to be the potential to exploit the inherent hardware scalability between them on some task of a size that fills a.*

Software scalability refers to the ability to exploit task and hardware scalability, with little or no changes, other than parameters.

Definition 3 *We define the software scalability function, $\sigma(a, b)$, in the case of software scalability between two homogeneous architectures, a (the larger) and b (the smaller), to be the real-valued function giving the increase in performance of a over b. Typically we do expect some increase in performance but we do not generally (at least in the homogeneous case) expect "super-linear" performance, i.e., $1 \leq \sigma(a, b) \leq \chi(a, b)$. In most cases we expect σ to be a simple multiple of χ, i.e., $\sigma(a, b) = \lambda \times \chi(a, b)$, where $1/\chi(a, b) \leq \lambda \leq 1.0$. If λ is close to 1.0, i.e., $\lambda = 1 - \varepsilon$, we usually feel we have scaled up well.*

Many examples exist of scaling up in this homogeneous sense, though, since it depends on a problem data size large enough to "fill" the large machine, it thus sometimes depends on an unrealistically large data size. In particular it appears to

us that some of the most recent HPC machines are "scalable" only in the sense that they could run matrix or other similar scientific problems of a size that no one is yet ready to do.

5.2 Heterogeneous Case

The heterogeneous case is clearly more complicated, though it is also the case in which we can aspire to the ultimate in heterogeneous computing potential, i.e, to achieve σ's significantly greater than χ; this is what we mean by super-linear performance. In the heterogeneous case, there may be no commonality between two different architectures, so that the only way to talk about "scaling" is based on the performance potential. That means, we will have two different scalability standards, namely peak MFLOPS (in either fixed 63 or 32 bit mode) or GBS ("gibbs"), billions of bits per second (processed). Using this, we can extend the χ function to the heterogeneous case. For example if we had a large vector machine, a, capable of processing 8.7 billion bits per second or 8.7 GBS, and a small SIMD machine, b, of 1.3 GBS, then $\chi(a,b) = 8.7/1.3 = 6.69$. Having extended the hardware concept of scalability to heterogeneous cases, the task and software scalability follow immediately.

5.2.1 Fundamental Theorem of Scalability

To understand this theorem, we need to look at the figure 5.1.

We consider there to be at least four levels at which a task is defined. One is at the overall functional level, here considered to be the problem "Find a datum". Next, below this is the approach. By "approach" we mean something at a higher level than algorithm, perhaps meta-algorithm would be another term. In any case, for this problem, there is a radical difference in the approach for SIMD machine, used associatively (see [19]) or non-SIMD machines. In the former case, we can use simple associative search, which is $O(1)$; in the latter case we would typically use a sort, then

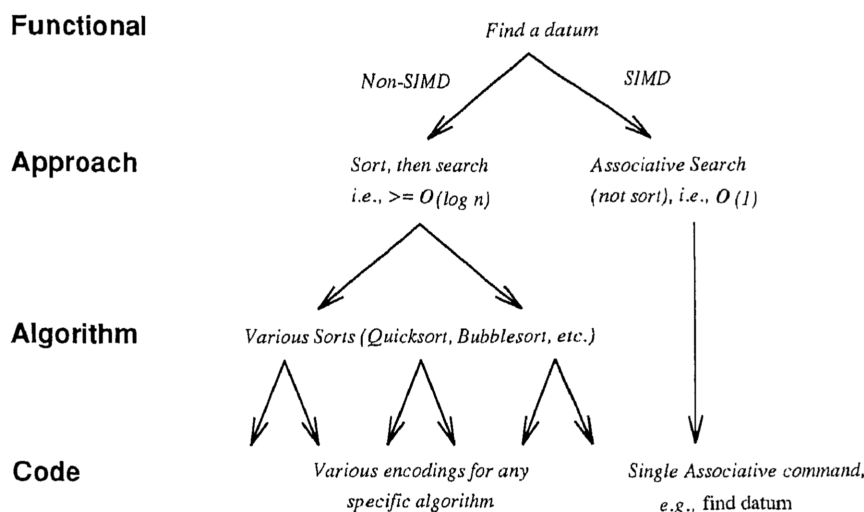

Figure 5.1 Hierarchical breakdown of a task

search operation, i.e., the asymptotic performance is bounded by $\Omega(\log n)$. For the associative search on a suitable SIMD machine, there is really only one instruction "find datum", so that there is no room for differing algorithmic or code variations. However in the case of non-SIMD, non-associative sort and search, there are many variations possible. For example, depending on data, parameters, architecture, etc., we could use a number of different search techniques, and similarly we could use a number of different coding schemes for each algorithm.

In this context, most researchers, when describing "scalability", certainly do not mean that the specific code is heterogeneously scalable, and generally do not mean that the the algorithm is heterogeneously scalable. For example, a matrix times vector operation might best be done with a SAXPY style algorithm on one machine and an SDOT on another. At the same time, the term "scalability" almost never applies to the functional level, since this is far too general to have any real meaning (in the usual context of scalability). WHAT IS ALMOST ALWAYS INTENDED IS THAT THE TERM "SCALABILITY" APPLY TO THE APPROACH LEVEL. However the example above shows that this is inadequate to support efficient

MPP/HPC performance. That is, a "scalable" approach to finding data would almost certainly be based on the non-SIMD, non- associative approach of "sort, then search". This might get maximal performance on non-SIMD machines, and might also work on SIMD, but certainly not optimally! That is the scalable approach is $\Omega(\log n)$, whereas the non-scalable SIMD version is $O(1)$. This example illustrates two thing:

a. A case where the non-scalable (at the approach level) SIMD implementation is inherently more effective than the scalable approach implemented on the same machine.

b. In this case suppose the non-SIMD machine has a hardware scalability factor of κ over the SIMD, i.e., $\chi(\text{non-SIMD,SIMD}) = \kappa$. However if n (the data size) is large enough, i.e., $n \geq 2^{\kappa}$, then the SIMD machine would have a task scalability OVER the non-SIMD, i.e, $\sigma(\text{SIMD,non-SIMD}) \geq O(\log n/\kappa)$. That is we have hardware scalability one way, and task/software scalability the other! In other words the scalable approach is inherently ineffective in this case. Thus we get:

Theorem 1 *Issues of hardware, algorithmic, and software scalability are inherently insufficient to exploit the potential of HPC in heterogeneous parallel environments.*

5.3 Scalability of HAsC and Cluster-M

Both programming paradigms presented in this paper are machine independent as explained in detail and are therefore scalable. In HAsC, a program is broadcast to the entire network, the individual nodes determine locally which instructions to execute. The global broadcasting approach means that there is no need to know how nodes are interconnected in the network, or how data is distributed across the nodes. This allows data files to be analyzed dynamically at run time as they enter the HAsC system and to be directed to the node(s) (i.e. computers) best suited to process them. Broadcasting allows scalability. That is, the hardware can be expanded or

modified and the problem can be changed without having to reprogram or recompile the basic HAsC program. New nodes consisting of new machines with installed HAsC software can be added to a network at any time, and at any location. HAsC is not dependent on any physical machine or network configuration. This is because the instruction broadcast, cell memory organization and associative searching allows the removal of any reference to data set size and type from the program. The basic component of a HAsC command is to "process all data which matches the following specifications." Changes in file sizes and data types are handled automatically at the node level. Similarly, Cluster-M is also scalable. When a new machine is added to the heterogeneous networks, a new Cluster-M representation of the new suite can be generated and a Cluster-M specification can be efficiently executed without any change. Also, an appropriate new mapping function can be computed to map the Cluster-M specification to the new Cluster-M representation.

CHAPTER 6

CONCLUSION AND FURTHER RESEARCH

In this thesis, two programming paradigms for heterogeneous computing called Cluster-M and HAsC has been presented. HAsC models a heterogeneous network as a coarse grained associative computer. In HAsC a program is broadcast to the entire network, the individual node determines which instruction to execute. Broadcast allows scalability. Cluster-M also allows scalability since programs written using Cluster-M are machine independent and can be efficiently mapped and ported among different systems. Both mechanisms were discussed in detail and their scalability and merits for heterogeneous computing were studied. Concurrent use of HAsC and Cluster-M was also presented. Cluster-M paradigm can be used to aid the shortcomings of HAsC, while HAsC can be used when the associative computing features is more desirable.

REFERENCES

[1] G. Agha. *Actors: A Model of Concurrent Computations in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] F. Berman and B. Stramm, "Prep-p: Evolution and Overview," Technical Report cs89-158, Dept. of Computer Science, University of California at San Diego, 1987.

[3] S. H. Bokhari, "On the Mapping Problem," *IEEE Trans. on Computers*, c-30(3):207–214, March 1981.

[4] S. H. Bokhari, *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publisher, 1990.

[5] N. Carriero, D. Gelernter, and J. Leichter, "Distributed Data Structures in Linda," In *Proc. of the 13th ACM Symposium on Principles of Programming Languages*, January 1986.

[6] S. Chen, M. Eshaghian, and M. Shaaban. "Automatic Fine Grain Mapping with Cluster-M," Technical Report, Submitted to International Parallel Processing Symposium, 1994.

[7] K. Efe. , "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, 15(6):50–56, 1982.

[8] H. El-Rewini and T. G. Lewis. "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, pages 138–153, 9 1990.

[9] M. Eshaghian. "Cluster-M Parallel Programming Model," In *Proc. International Parallel Processing Symposium*, pages 462–465, Mar. 1992.

[10] M. Eshaghian and M.Shaaban. "Cluster-M Parallel Programming Paradigm," *International Journal of High Speed Computing*, Accepted to Be Published.

[11] M. Eshaghian and M. Shaaban. "A Cluster-M Based Mapping Methodology," In *Proc. International Parallel Processing Symposium*, pages 213–221, April 1993.

[12] I. Foster and S.Tuecke. "Parallel Programming with PCN," Technical Report, Argonne National Laboratory, University of Chicago, January 1993.

[13] Ian Foster and Taylor Stephen. *STRAND, New Concepts in Parallel Programming.* Prentice Hall, 1975.

[14] R. F. Freund and D.S. Conwell. "Superconcurrency: A Form of Distributed Heterogeneous Supercomputing," *Supercomputing Review*, 3:47–50, Oct. 1990.

REFERENCES
(Continue)

[15] A. Gerasoulis, S. Venugopal, and T. Yang. "Clustering Task Graphs for Message Passing Architectures," In *ACM International Conference of Supercomputing*, June 1990.

[16] A. Khokhar, V. K. Prasanna, M. Shaaban, and C. Wang. "Heterogeneous Supercomputing: Problems and Issues," In *Proc. Workshop on Heterogeneous Processing*, pages 3–12, Mar. 1992.

[17] S. Lee and J. Aggarwal. "A Mapping Strategy for Parallel Processing," *IEEE Trans. on Computers*, 36:433–442, April 1989.

[18] R. Ponnusamy, N. Mansour, A. Choudhary, and G. C. Fox. "Mapping Realistic Data Sets on Parallel Computers," In *Proc. 7th International Parallel Processing Symposium*, pages 123–128, April 1993.

[19] Jerry L. Potter. *Associative Computing.* Plenum Press, New York, NY, 1992.

[20] Muhammad E. Shaaban. "Mapping Methodologies for Heterogeneous Supercomputing," PhD thesis, Dept. of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA, 1993.

[21] L. Snyder. "Parallel Programming and the Poker Programming Environment," *Computer*, pages 27–36, July 1984.

[22] H. S. Stone. "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Eng.*, SE-3:85–93, January 1977.

[23] S. Rajopadhye V. M. Lo, S. Gupta, D. Keldsen, M. A. Mohamed, and J. A. Telle. Oregami: "Software Tools for Mapping Parallel Computations to Parallel Architectures," In *Proc. International Conference on Parallel Processing*, 1990.

[24] J. Yang, L. Bic, and A. Nicolan. "A Mapping Strategy for MIMD Computers," In *Proc. International Conference on Parallel Processing*, 1991.

[25] T. Yang and A. Gerasoulis. "A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors," In *Proc. IEEE Scalable High Performance Computing Conference*, April 1992.