A Task Communication Language and Compiler for the NuMesh

by

Jason Steven Cornez

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author .....
Department of Electrical Engineering and Computer Science
May 16, 1994

Certified by .....
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

A Task Communication Language and Compiler for the NuMesh

by

Jason Steven Cornez

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 1994, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The success or failure of parallel computing rests not in the ability to design and build powerful high-speed, advanced machines with parallel architectures, but rather in the ability to organize and harness that power. A software development environment for a parallel computer must provide programmers with a high-level language for describing parallel computations and must also provide a compilation system capable of generating low-level processor and routing code subject to demanding timing constraints. Of particular importance are the abstract programming model underlying the parallel language and the tools for mapping parallel computations to parallel architectures forming an integral part of the compilation system. This thesis explores the details of such an environment for the statically routed NuMesh machine.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I wish to begin by thanking Michael Dertouzos for providing the funding for my research. Thanks also go to Steve Ward for advising this thesis and to Greg Papadopoulos for his involvement as my academic advisor. Jay Keyser, your advise, humor, poetry, and friendship have undoubtedly helped me in my stay at MIT. I also wish to express my gratitude to everyone involved in the NuMesh project for allowing me to be a part of their group for such a brief time. A special word of thanks goes to Anne Hunter for all the things she does and, most of all, for the care she gives to every course VI student.

What would lunch time be without Jon or Neil? Having lunch with Jon has become routine during the writing of this thesis and it is always a welcome pleasure when Neil can join us. Having been friends throughout our stay at MIT, I suppose it was fitting that we should all be finishing our theses at the same time. Thanks for the good ideas, the good times, and our good friendships.

Denise, for those times when for no particular reason I decide to call you just to say hi, thanks for being there. It seems like you and I have been in the same boat of uncertainty for quite some time. We survived. On to new adventures–best of luck to us both!

I'm not sure exactly how this goes, but a poet once wrote that a family gives a person two gifts: roots and wings. Thanks for providing me with so much more as well. To my mother and father, for all those things parents do without question. The love you have given me and the faith you have in me have been a constant source of inspiration and have given me the ability to believe in myself. To Uncle Gerry, for his warm friendship and for the way he teaches me to be a good person through his wonderful example. To my brother, for the ways he reminds me to be young and live life to its fullest. And to Auntie, who just by virtue of being herself, has added sparkle to my life.

My dear friend Susan, I think in the past year you and I have been through it all. You know, I can't think of anyone I would want to go through it all with more. Although I thought I already knew what it was, you have re-taught me the meaning of true friendship. I think there is something very important that you and I both learned this year: when you're feeling down and out, "Stop. Look around you. No one's screaming at you, So you feel all right for ten minutes. If you feel all right for ten minutes, Why don't you feel all right for twenty minutes, Feel all right for forty minutes, Drop it and smile. Why don't you feel all right for the rest of your life?" Thank you Susan, for being such a wonderful person, such a wonderful friend. My life has been so much fuller just by knowing you.

At my grandmother's funeral, I don't remember crying and that always bothered me because I loved her dearly. I just cried as I wrote this line and so this thesis is dedicated to her in loving memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the quest for ever more computing power, researchers have come to agree that parallelism is the vehicle through which this power will be realized. It is no surprise then that much effort has gone into the design and implementation of parallel computing environments. In order to successfully tap the potential of parallel processing, effort must be spent not only on building the hardware for the parallel machine but also on creating the software development environment which will allow programmers to take advantage of parallelism in a general yet straightforward way.

The true challenge that parallel processing presents is one of organization. When tens or hundreds or even thousands of processors are all doing work on the same problem, how can they all communicate with each other in an efficient and reliable manner so that their individual efforts can be combined into a coherent, global solution? This challenge is known as routing and it is more or less the responsibility of the interconnection network to provide a working solution. Many popular concurrent architectures have chosen a dynamic approach based on a general purpose network design with the logic to route each message at the time it is sent. A more static approach to routing, which has traditionally been associated with special purpose machines, relies on a compile-time analysis of the network traffic resulting from each application in order to pre-determine routes for each message.

1.1 The NuMesh Project

The Computer Architecture Group at MIT's Laboratory for Computer Science is exploring the use of static routing in the context of a more general purpose machine as part of the NuMesh project [18]. Thus, the major thrust of the NuMesh project is to define a highly-efficient, generalized communication and interconnect substrate with programmable control paths which is capable of supporting high-bandwidth, low-latency communication as is required for a successful statically routed machine. In a NuMesh machine, this substrate is built by composing nodes, each constituting a digital subsystem that communicates with neighboring nodes via a standardized physical and electrical interconnect, in a Lego-like manner. The result is a scalable, modular, and reconfigurable 3D nearest-neighbor lattice. Figure 1-1 shows a representation of a NuMesh in a diamond lattice configuration.

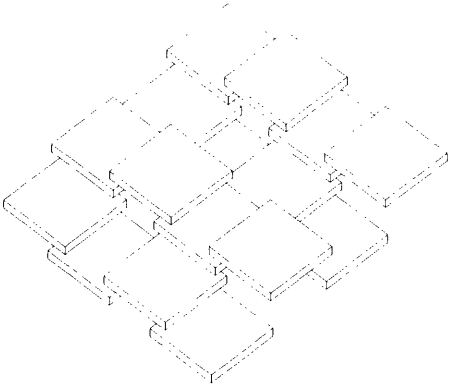

Figure 1-1: A view of NuMesh in diamond lattice configuration.

An important goal of the NuMesh project is to study the degree to which static routing techniques can be effectively employed for a wide range of useful applications. To this end, compiler techniques are being developed to generate detailed low-level routing information from high-level descriptions of parallel programs. There are also efforts to identify and express the mix of communication requirements which characterize a particular application. This work hopes to show that many applications which have been characterized as dynamic may actually be characterized as *nearly* static. An application which is nearly static is one where all but a very few parameters are determinable at compile time. Techniques are being developed to deal with this class of applications.

1.2 Programming a Parallel Computer

Designing and implementing algorithms for parallel computers presents programmers with many new challenges above and beyond those associated with programming sequential computers. The primary consideration is the programming model. Many effective algorithms have been developed for specific network topologies [6]. This approach and its intuitive appeal lead to graph theoretic models, where parallel computations are abstractly viewed as a set of processes communicating via explicit message-passing. These processes and their communications together form a graph of regular topology such as a tree, mesh, or hypercube.

Another major concern in programming a multi-computer is in instructing each processor as well as (in the case of static routing) the routing hardware. In the infancy of many parallel machines, this process of mapping an algorithm to the computer must be performed manually by the programmer. This proves to be a difficult and error-prone task which may be more affordably performed by an automated mapping system as part of a compiler. These systems, in various forms and with varying capabilities, have been or are being developed for today's parallel processing computers.

1.3 This Thesis

The work in this thesis discusses parallel programming models at an abstract level and evaluates those models based on their ability to describe a wide variety of parallel algorithms, their ease of use for the programmer, and their ability to capture information that will be needed to map the algorithm to a parallel machine. The mapping problem is also examined in some detail and methods for its implementation are presented and evaluated. The methods discussed are examined and evaluated with respect to their expected value as part of a parallel programming environment for the NuMesh.

This thesis is divided into five chapters. Chapter 2 describes the current NuMesh network architecture and environment as well as providing background on static routing. A discussion of parallel programming models and task communication languages is presented in Chap. 3. The mapping problem is presented in Chap. 4 along with information

and qualitative evaluation of various ways to decompose and solve the mapping problem. Finally, Chap. 5 provides a wrap-up of the thesis and suggests future directions of this work.

Chapter 2

Static Routing and the NuMesh

The NuMesh Project is an approach to multi-computer design aimed at developing a flexible yet powerful communications substrate for processors in a parallel processing system [18]. This communications substrate should be scalable, modular, and capable of supporting communications between processors in a three-dimensional topology predominantly via static routing. Each module, or node, of the NuMesh provides interfaces to each of its nearest neighbors as well as to a local processor. Each node also includes a Communications Finite State Machine (CFSM) which is programmable and controls the flow information through all interfaces of the node. The NuMesh operates on a globally synchonous clock and is capable of transfering one word of data between each pair of adjacent nodes during every cycle.

2.1 Static Routing

One of biggest problems in programming parallel computers is that of getting the right information to the right processor when it needs it. In a uni-processor system, the processor always has access to all the information necessary for the computation. However, in a multi-computer system such as the NuMesh, a program's data is distributed among the processors and must be sent from one processor to another via explicit message-passing. Because the origin and destination of a message may be located arbitrarily within the interconnection network, that network must be able to route the messages between any

two processors. The performance of the routing system for a parallel machine often proves to be the performance bottleneck for most algorithms. Therefore, deciding on and implementing a routing scheme is one of the major hurdles in multi-computer design.

Consider an arbitrary network of computing nodes which can perform calculations and pass information in the form of messages to one another via the network. Suppose that each node is programmed so that it knows exactly what to do with every message it receives based only on where and when the message arrived. Further, the node has no knowledge of message's origin nor does it necessarily know where the message's final destination might be. Most importantly, the contents of the message itself play absolutely no part in determining what the node should do with the message. The NuMesh substrate is designed to support this type of message-passing which is known as static routing. A statically routed system relies on pre-compiled network traffic information for each algorithm in terms of both space and time in order to instruct the nodes to properly route each message.

The static routing model may appear to be overly restrictive and rely too heavily on pre-compiled information to be very useful for general applications. Later in this section, static routing will be compared to dynamic routing to indicate the strengths of each and what applications are best suited to a statically routed system. A portion of Chap. 4 is dedicated to discussing compiler techniques for generating and analyzing traffic information for a large class of algorithms. Furthermore, one of the general goals of the NuMesh Project is to show that static routing provides significant gains and is useful for a wide variety of important applications.

2.1.1 Example: Merge-sort

Figure 2-1 depicts a simple example of merge-sort being performed on a complete binary tree with depth three. If the compiler is told the size of the array to sort, it can fully determine the static routing requirements of this algorithm. In this example, the input size is eight. Also, for the sake of simplicity and illustration, assume that one element can be transfered between adjacent nodes in 1 cycle and that a node can communicate with at most one other node per cycle. In terms of computation, assume that merging two sorted

arrays of size n and m takes $n + m$ cycles and that sorting two elements takes only 2 cycles.

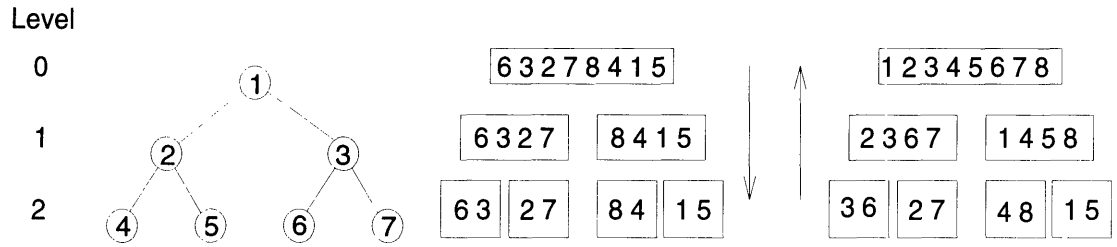

Figure 2-1: Merge-sort as an example of static routing.

Cycles	Action
1 - 4	Send the first four elements at the root to its right child.
5 - 8	Send the second four elements at the root to its left child.
9 - 10	Each node at level 1 sends its first two elements to its left child.
11 - 12	Each node at level 1 sends its second two elements to its right child.
13 - 14	All the nodes at level 2 sort their two elements.
15 - 16	The odd nodes at level 2 send their sorted arrays up to their parent.
17 - 18	The even nodes at level 2 send their sorted arrays up to their parent.
19 - 22	All the nodes at level 1 merge the inputs from the right and left children into a single sorted array.
23 - 26	The even node at level 1 sends its sorted array up to the root.
27 - 30	The odd node at level 1 sends its sorted array up to the root.
31 - 38	The root merges the inputs from its left and right child into the final sorted array.

Table 2.1: Static cycle analysis of merge-sort

All of the information in Table 2.1 could have easily been derived by a complier. This is a good start in showing an example of static routing, but another important factor to consider is the architecture on which the algorithm will be run. So far the merge-sort problem has only been presented and analyzed in terms of its logical connections. This works fine if the tree's logical nodes and edges can be mapped directly to physical processors and channels. In almost all instances, however, this is not the case. Parallel algorithms are typically described in terms of some abstract regular graph topology (see Chap. 3) whereas a parallel computer will have some fixed network topology. Even in

cases where the topology of the algorithm and the machine are the same, the algorithm may abstractly require a number of processes which varies with the size of the input. The number of physical processors in a machine will of course remain constant.

Consider a parallel machine whose architecture is a dimension 3 hypercube. When the merge-sort algorithm presented above is mapped to this machine, it is impossible to map all the logical connections to single physical connections. An optimal mapping of a depth 3 tree to a dimension 3 hypercube is shown in Fig. 2-2. Given the assumptions previously stated, the overall number of cycles required to complete the computation is not changed. Again, it is possible for a compiler to generate the equivalent of Table 2.2 and from that generate static routing code for each of the nodes in the hypercube.

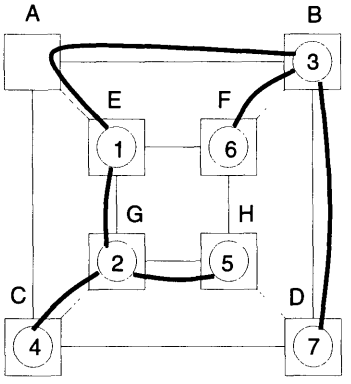

Figure 2-2: A depth 3 complete binary tree mapped to a dimension 3 hypercube.

Cycles	A	B	C	D	E	F	G	H
1 - 4	Rec E	idle	idle	idle	Send A	idle	idle	idle
5 - 8	Send B	Rec A	idle	idle	Send G	idle	Rec E	idle
9 - 10	idle	Send F	Rec G	idle	idle	Rec B	Send C	idle
11 - 12	idle	Send D	idle	Rec B	idle	idle	Send H	Rec G
13 - 14	idle	idle	Merge	Merge	idle	Merge	idle	Merge
15 - 16	idle	Rec F	Send G	idle	idle	Send B	Rec C	idle
17 - 18	idle	Rec D	idle	Send B	idle	idle	Rec H	Send G
19 - 22	idle	Merge	idle	idle	idle	idle	Merge	idle
23 - 26	Rec B	Send A	idle	idle	Rec G	idle	Send E	idle
27 - 30	Send E	idle	idle	idle	Rec A	idle	idle	idle
31 - 38	idle	idle	idle	idle	Merge	idle	idle	idle

Table 2.2: Static determination of processor activity.

2.1.2 Comparison of Static and Dynamic Routing

The alternative to static routing is dynamic routing. In a dynamically routed system, the nodes do not need to be programmed specifically for each algorithm. Instead, the messages have a header associated with them which contains the routing information for that message. This routing information may simply be the destination node or it may be a sequence of nodes corresponding to a virtual channel. This header also might contain a timestamp along with information about the message's origin and the overall size of the message which might be broken up into several packets. Each node has built in programming which looks at the header of the message and at other local message traffic conditions, and then continues the message on its way based on this information. The program at each node must know how to deal with contention for any particular link since messages are not scheduled globally and are therefore subject to collisions with one another.

Static routing enjoys several benefits over dynamic routing. In dynamic routing messages have a header which must be sent along with the message, causing an increase in network traffic. In addition, decoding the header information and interpreting it take network cycles which increase the latency for individual messages traveling through the network. The dedicated hardware required by a dynamically routed machine is complicated and costly to design whereas statically routed machines require much simpler routing hardware. In a dynamically routed system using an adaptive routing strategy, messages may arrive in an unpredictable order. Furthermore, dynamic routing techniques may result in deadlock. Strategies have been developed to prevent deadlock in dynamically routed systems, but often at some cost in performance. Perhaps the most significant benefit of static routing is the potential for reduced contention for network resources. By computing at compile time all the traffic an application generates, routing can be scheduled in space and time such that messages never contend for the same link on the same cycle. This has the potential to significantly improve application performance.

The advantages that dynamic routing has over static routing are important as well. Foremost on this list is that dynamic routing is completely general and works for every parallel application, whereas the techniques developed for static routing are only appli-

cable for a restricted set of application. The popular perception of the restrictions static routing imposes on applications may be greater than in reality and so this too works in favor of dynamic routing. Compiler technology required to determine the rigorous timing considerations necessary for a fully static system are in the early stages of development. Also, each parallel algorithm requires that the routing hardware be reprogrammed for that algorithm's specific message traffic requirements.

Certain programs with highly predictable communication requirements have been found to benefit most from static routing. Circuit simulators, CAD development software, and speech and video processing code are some such applications [12]. These applications exhibit repeated execution of simple computations and have certain real-time constraints requiring high performance which make them suitable candidates for the evaluation of statically routed systems.

2.2 The CFSM

The CFSM replicated at each node of a NuMesh system serves as the low-level communication and interface hardware. The CFSM provides the data path for all messages to and from neighboring nodes as well as to the local processor. The control of the CFSM is determined by a programmable finite state machine with a RAM-based state table. There is also some additional hardware such as looping counters to provide added functionality. Figure 2-3 shows an abstract view of a NuMesh node indicating the role of the CFSM.

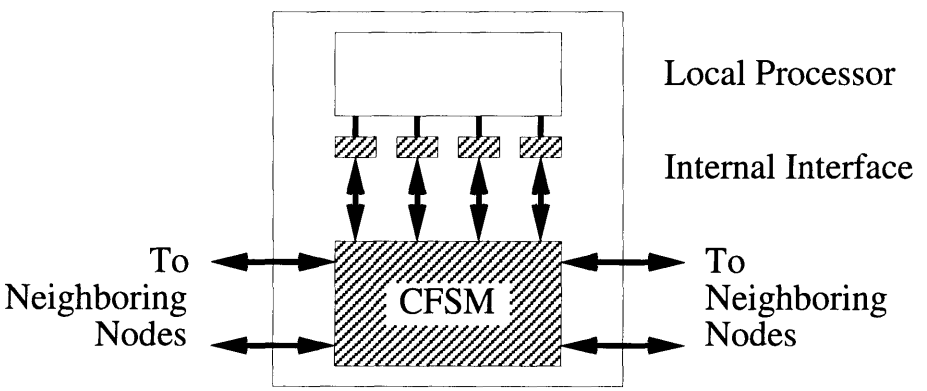

Figure 2-3: A NuMesh node schematic showing CFSM role.

The NuMesh Project is currently involved in an effort to re-design and re-implement a new, second-generation architecture for the CFSM [15]. The new design is aimed at providing control of multiple channels of communication simultaneously. This has resulted in the development of a multithreaded CFSM architecture capable of performing a data transfer to or from each port on nearly every cycle. This design allows for the high-bandwidth, low-latency communication that a static routing system has the potential to provide.

2.3 The NuMesh Programming Environment

The current programming environment of the NuMesh is still at an elementary stage. The programmer must explicity provide code for both the processors and CFSMs of each node. The code for the processors is written in C with the added resource of some NuMesh specific library functions [11]. CFSM code is specified in a Lisp-like assembler code designed to hide certain low-level details of the implementation [14]. A somewhat nicer, although more restricted interface exists for programming the CFSM and processor from common C source code [13]. This interface abstracts away from the CFSM details but imposes flow control restrictions on the processor code. This system also results in less efficient CFSM code while still requiring explicit code specification for each node. There is, however, a NuMesh simulator which provides a nice test environment for developing and testing NuMesh code [10].

Somewhat earlier work presents a graphical programming language based on streams and an environment in which to develop those programs [17]. This language prevents programs whose dynamic behavior depends on run-time data, thus allowing programs suitable for a statically routed system. The system seems most suitable for the development of real-time applications such as digital signal processing. Unfortunately, only the front-end program editor for this system was ever completely implemented.

More recent work provides the NuMesh with a limited system for statically allocating network resources based on an analysis of network traffic as determined at compile time [12]. This work yielded positive results towards the feasibility of a system for mapping applications to the NuMesh. This system has yet to be integrated with the NuMesh

simulation environment or hardware.

Chapter 3

Task Communication Languages

A vital component to any software development environment is the language provided. The programming language is the interface through which programmers instruct the machine. Without a high-level programming language, it remains prohibitively expensive, in terms of both time and complexity, for programmers to use the machine to solve problems, regardless of the capabilities of the machine. This has always been true for sequential machines and is perhaps an even more important consideration for parallel machines where the programmer has more flexibility and power but also more room for error. An effective language should be one which clearly and concisely captures the essential components of a parallel algorithm without needlessly complicating the implementation process for the programmer.

Languages for sequential machines have been undergoing a continuous evolution for more than forty years. It should not be a surprise then that developing languages for parallel computers will take a great deal of effort and will be an evolving process itself. One of the most important considerations in designing such a language will be to decide which paradigms the language should support. If designing a sequential language, this would be analogous to deciding whether the language should support block structure, be recursive, be functional, support object-oriented methodologies, be compiled or interpreted, etc. Each of these paradigms will affect not only the design of the language itself, but more importantly, it will affect the programming model. So in designing a parallel language, the programming model is necessarily being prescribed as well.

3.1 Parallel Programming Models

There are several different ways of viewing the programming of a parallel computer. This is to be expected in light of the numerous ways of programming a uni-processor computer. Consider the procedural approach and the object-oriented approach. It is possible to use either approach to successfully write a program to solve a problem. Neither approach has been shown to be inherently more powerful than the other in its ability to express a problem. Nor has either approach been shown to result in a substantially less code than the other. The fundamental difference in the approaches is in the way the code is designed and organized. This is the same type of difference we expect to have among the several parallel programming models.

3.1.1 Parallelization of Sequential Code

One of the first things people seem to want to do when they think about a parallel computer is take existing code from their uni-processor machines and run it directly on the multi-computer, immediately reaping the benefits of parallelism. It is this desire that leads to the following discussion of the parallelization of sequential code, even though it is not really a model for parallel compututation in it own right.

The idea here is simple although the realization of it may prove to be very difficult or even impossible in all but restricted cases. Let the programmer ignore that the program is being written for a parallel computer by allowing the use of some sequential language. The hope is that this could be some existing sequential language that the programmer is already comfortable using, such as C or Lisp. Even if it is a new language, however, it is likely to have much in common with existing sequential languages and therefore should not be difficult for the programmer to learn. The program, once written, is then processed by a compiler whose job it is to determine the parallelism that can be factored out of the code. In other words, this compiler does some analysis of the program which determines what sections of the code can be run independently and then generates independent code chunks for these sections which will be executed on different processors simultaneously in the parallel environment.

There are two driving forces which make the parallelization of sequential code an enticing idea. The first is that such a system will make it very easy for programmers to write code for parallel computers. There is a fear that if programming multi-computers is too difficult a task, the vast potential of parallelism will remain forever untapped. A system which would allow programmers to program in a format they are already familiar with would certainly alleviate this fear. The second and ultimately more important force is that such a system would mean that an enormous volume of existing software could be moved to parallel platforms with relatively little effort. Imagine if a compiler for effectively parallelizing C code was developed. Then once the run-time libraries were implemented for a parallel machine, any C program could take advantage of the parallel processing available to it. The benefits of this would be many.

Unfortunately, the general parallelization of sequential code appears to be a very difficult problem. There are in fact strong arguments that suggest that it is impossible. Consider the simple sequential algorithm for computing the n^{th} Fibonacci number. Figure 3-1 shows pseudo-code for both a recursive and an iterative version of this procedure. Now consider parallelizing each version of this algorithm.

```
Fib-Recurse(n)
  1. IF n = 0 THEN RETURN 0
  2. IF n = 1 THEN RETURN 1
  3. RETURN Fib-Recurse(n-1) + Fib-Recurse(n-2)


Fib-Iter(n)
  1. IF n = 0 THEN RETURN 0
  2. IF n = 1 THEN RETURN 1
  3. fib  <- 0
  4. fib2 <- 1
  5. FOR i <- 2 TO n
  6.    temp <- fib
  7.    fib  <- fib + fib2
  8.    fib2 <- temp
  9. RETURN fib
```

Figure 3-1: Pseudo-code for sequential Fibonacci procedures.

The recursive Fibonacci procedure might immediately suggest a divide and conquer parallel solution on a binary tree. The parallelization of this code would begin with a single copy of the code on the root of the tree. Every time the recursive call is made, the code would replicate itself onto each of its children passing the left child the argument $(n-1)$ and the right child the argument $(n-2)$. When the base case of the recursion is reached by any of the spawned processes, the result is passed up to its parent and the process terminates and may be deallocated. When any process receives a result from both of its children, it sums those results and passes that sum to its parent at which time it terminates and may be deallocated. The final result will be found at the root when the entire process completes.

The iterative Fibonacci procedure does not immediately suggest any direct parallelization. In fact, every iteration of the loop depends directly on the previous iteration of the loop and furthermore the statements within the loop cannot be reordered. The loop can be reduced to a single assignment, `fib[n] <- fib[n-1] + fib[n-2]`, if an array is used to store all of the first n Fibonacci numbers, but that still leaves it impossible to find a way to parallelize this algorithm.

The conclusions that can be drawn from these observations suggest that, for a general piece of code, parallelization may not be possible. Further, note that the sequential recursive Fibonacci algorithm has a running time with complexity $O(\phi^n)$. [1] It is easy to confirm that the algorithm has exponential complexity by observing that `Fib-Recurse(n+2)` must compute `Fib-Recurse(n)` twice. The parallel version of this algorithm will take about $2n$ steps to complete assuming that spawning new processes takes only one step and that there is a processor for every new process spawned.[2] This is because $(n-1)$ processes will be spawned by following the left-most branches and each of these processes must also pass a value back up to its parent. The iterative algorithm, however, will take about n steps to complete on a single processor. These results also do not bode well for a general system to parallelize sequential code.

Nonetheless, parallelization of sequential code may still prove to be a useful approach

[1] The value of ϕ is the golden ratio: $(1+\sqrt{5})/2 = 1.61803...$

[2] These assumptions are both quite optimistic.

in porting certain applications to parallel machines. Techniques similar to those used by today's optimizing compilers can be extended to discover independent blocks of code which may be run as separate processes. Sequential object-oriented code may perhaps be parallelized with some degree of success by making separate processes out of the various objects. Also, it is often possible to pipeline software onto multiple processors, in much the same way that hardware is pipelined, to make multiple instances of an algorithm run faster.

3.1.2 Static Task Graphs

The remaining programming models all relate to designing algorithms which are explicitly parallel in nature. All but the last of these is based on a graph theoretic model of parallel computation. In fact, these graph models also apply directly to the parallelization of sequential code where, although the programmer isn't concerned with modeling the parallel computation explicitly, the compiler must construct a graph of the parallelized version of the code at some stage. Graph theoretic models are appropriate for modeling parallel computations because, when viewed abstractly, parallel computations always consist of some set of communicating processes. The processes may be viewed as nodes and the communications as edges which form a graph. This graph is equivalent to the abstract view of the parallel computation and so a language which can be used to describe such a graph can also be used to describe a parallel computation. In addition, there are many known algorithms for representing, manipulating, and analyzing graphs that may be beneficially applied to descriptions of parallel computations. Another benefit to the graph models is certainly that the topology of any real parallel computer is necessarily based on some graph model.

The first and simplest of these models is the static task graph [16]. A static task graph is the most basic way of modeling a parallel computation. In the static task graph, each node represents a process and each edge represents where two processes communicate with one another. In this model, the parallel algorithm is designed as a set of static and persistent processes which communicate via explicit message passing. The simplicity of this model is its probably its biggest advantage. Whenever considering a parallel algorithm

in abstract terms, the static task graph is likely to come to mind. If the parallel computation is a divide and conquer algorithm, the graph will usually be a tree; for a parallel matrix multiple algorithm, the graph will probably be a mesh; and parallel FFT algorithms usually correspond to butterfly graphs.

In addition to the conceptual simplicity and applicability of static task graphs, they are also easy to describe textually in the form of a language. There are two common ways of doing this: describing the nodes or describing the connections. In a language describing the nodes, all the different types of nodes are given a description of which other nodes they communicate with. Thus, to describe a complete binary tree, there would be three different node types: one for the root, one for the leaves, and one for all the intermediate nodes. The Prep-P system uses a graph description language GDL [1] which describes graphs in terms of their nodes. Figure 3-2 shows the GDL description of a complete binary tree and Figure 3-3 shows the GDL description of an 8×8 mesh. As can be seen, any graph may be expressed in this way, although perhaps not without some tedium. Languages which describe the connections rather than the nodes may provide more compact descriptions for regular graphs. These languages begin by labeling some arbitrary set of nodes and then describing which nodes connect to which others. The OREGAMI system uses LaRCS [8] which is actually based on the Temporal Communication Graph described in a following section. LaRCS may however used to describe static task graphs in terms of their edges as demonstrated in Figs. 3-4 and 3-5 which describe a complete binary tree and a mesh respectively.

3.1.3 Process-Time Graphs

While the static task graph gives a view of the total algorithm with all its processes and all their communications in one instance, the process-time graph [9] instead shows the activity of each process over time. This is a model which was actually developed in conjunction with the parallelization of sequential code. Here, process activity is viewed as a sequence of atomic events where each event is either some finite computation or a communication with another process. The sequence of atomic events for each process is represented as a linear chain of nodes connected by directed edges indicating the temporal ordering of

```
nodecount = 15

procedure ROOT
  nodetype: { i == 1 }
  port LCHILD: { 2 * i }
  port RCHILD: { (2 * i) + 1 }

procedure MIDDLE
  nodetype: { i > 1 && i < (nodecount + 1) / 2 }
  port PARENT: { i / 2 }
  port LCHILD: { i * 2 }
  port RCHILD: { (2 * i) + 1 }

procedure LEAF
  nodetype: { i >= (nodecount + 1) / 2 && i <= nodecount }
  port PARENT: { i / 2 }
```

Figure 3-2: GDL code for a complete binary tree.

events. Directed edges moving forward in time from one process to another represent where one process communicates with another. The distance a communication moves forward in time indicates how many atomic events take place from when the message is sent to when it is received.

This model, based directly upon Lamport's process-time diagrams [5], provides a detailed view of the algorithm's operation over time. In fact, thinking about the process-time diagram when designing, implementing, or debugging a parallel computation will often be a useful exercise. The model forces the programmer to consider what every process is doing during each atomic step and will immediately show problems with process synchronization as well as showing when parallelism is being exploited effective and when it is not. The major drawback to this model is that it fails to present a simple abstract view of the algorithm which it describes. In other words, glancing at a process-time graph does not immediately suggest the conceptual structure of the algorithm in question. Figure 3-6 shows a process-time graph for a parallel merge-sort algorithm.

```
width = 8
height = 8
nodecount = width * height

procedure UPLEFT
  nodetype: { i == 1 }
  port RIGHT: { i + 1 }
  port DOWN:  { i + width }

procedure UPMID
  nodetype: { i > 1 && i < width }
  port RIGHT: { i + 1 }
  port LEFT:  { i - 1 }
  port DOWN:  { i + width }

procedure UPRIGHT
  nodetype: { i == width }
  port LEFT: { i - 1 }
  port DOWN: { i + width }

procedure LEFTMID
  nodetype: { i > 1 && i mod width == 1 && i < (nodecount - width) }
  port RIGHT: { i + 1 }
  port UP:    { i - width }
  port DOWN:  { i + width }

procedure MIDDLE
  nodetype: { i > width && i < (nodecount - width)
              && i mod width != 1 && i mod width != 0 }
  port RIGHT: { i + 1 }
  port LEFT:  { i - 1 }
  port UP:    { i - width }
  port DOWN:  { i + width }

procedure RIGHTMID
  nodetype: { i > width && i mod width == 0 && i < nodecount }
  port LEFT: { i - 1 }
  port UP:   { i - width }
  port DOWN: { i + width }

procedure DOWNLEFT
  nodetype: { i == (nodecount - width + 1) }
  port RIGHT: { i + 1 }
  port UP:    { i - width }

procedure DOWNMID
  nodetype: { i > (nodecount - width + 1) && i < nodecount }
  port RIGHT: { i + 1 }
  port LEFT:  { i - 1 }
  port UP:    { i - width }

procedure DOWNRIGHT
  nodetype: { i == nodecount }
  port LEFT: { i - 1 }
  port UP:   { i - width }
```

Figure 3-3: GDL code for an 8×8 mesh.

```
static_binary_tree(levels)

nodelabels 1..2**levels - 1;

comtype lchild(i) i <=> 2*i;
comtype rchild(i) i <=> 2*i + 1;

comphase full_tree
  forall k in 1..2**(levels-1) - 1
    { lchild(k); rchild(k); }

phase_expr
  full_tree;
```

Figure 3-4: LaRCS code for a complete binary tree.

```
mesh_2d(width, height)

nodelabels 1..width*height;

comtype right_left(i) i <=> i+1;
comtype up_down(i)    i <=> i+width;

comphase row(i)
  forall j in 1..width-1
    right_left((i-1)*width + j);

comphase column(i)
  forall j in 1..height-1
    up_down(i + (j-1)*width);

phase_expr
  forall i in 1..height
    row(i) ||
  forall i in 1..width
    column(i);
```

Figure 3-5: LaRCS code for a mesh.

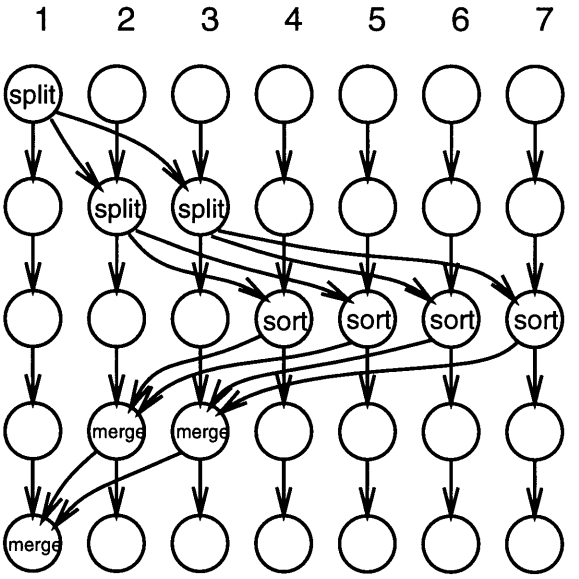

Figure 3-6: A process-time graph for the merge-sort algorithm.

3.1.4 Temporal Communication Graphs

The temporal communication graph [9] is a combination of the static task graph and the process-time graph. It combines the strengths of each of these previous models by providing a concise conceptual view of a parallel algorithm while also capturing the details of the algorithm's temporal behavior. Graphically, this model essentially consists of both the static task graph and the process-time graph. The static task graph may, however, be augumented with color to show which edges are active simultaneously. The process-time graph then may be viewed as the static task graph if unrolled in time; likewise, the static task graph may be viewed as the process-time graph projected onto a single instance in time. The important observations about temporal communication graphs are that they may be concisely described by a textual language and that they provide all the information needed to perform optimal contraction, placement, and routing when mapped to a parallel machine. These observations are what make the temporal communication graph an important model in its own right.

In this model, as derived from the two previous models, a parallel computation is designed as a set of static and persistent processes which compute and communicate at distinct and well-defined instances in time. This is probably how a programmer designing

a parallel algorithm using either of the two previous models alone would actually conceive the algorithm. Using temporal communication graphs allows the programmer to fully describe this natural conceptual model in a concise and unambiguous format. The model is also both expressive and flexible. Most regular communication structures have a natural representation and algorithms described using this model are not constrained to a particular machine or architecture. These features arise naturally in temporal communication graphs because algorithms described in this way are represented abstractly and all timing is relative to atomic events within the computation itself rather than being relative to some external global clock.

As mentioned in the section on static task graphs, LaRCS is a language based on the temporal communication graph model. There has already been an example demonstrating how LaRCS can be used to describe a static task graph. Figure 3-7 is a LaRCS program describing level-by-level behavior in a complete binary tree. Temporal information is expressed by making a distinction between the description of a computation or communication phase and its instantiation. Thus, the algorithm is fully described by first defining the static components which make up the static task graph as communication and computation phases and then composing those phases into a phase expression which dictates the temporal ordering of the phases. This approach is well-suited to capturing regularity in the graph topology and the distinct phases of logically synchronous computation and communication.

3.1.5 Dynamic Task Graphs

The major shortcoming of the Temporal Communication Graph as it is described above is that it can express only parallel computations modeled as *static* and *persistent* processes. Specifically, this means that the processes must all be *static* in that they are created at compile-time and *persistent* in that they exist throughout the lifetime of the computation. While this allows the description of a large class of parallel computations, it leaves other computations, such as the parallelization of the recursive Fibonacci procedure presented earlier in this chapter, difficult or impossible to describe. These compuations are more naturally described as a system of dynamically created and deallocated processes. This

```
level_by_level_binary_tree(levels)

nodelabels 1..2**levels - 1;

comtype lchild(i) i => 2*i;
comtype rchild(i) i => 2*i + 1;
comtype parent(i) i => i/2;

comphase down_tree(k)
  forall i in 2**(k-1)..2**k - 1
    { lchild(i); rchild(i); }

comphase up_tree(k)
  forall i in 2**(k-1)..2**k - 1
    parent(i);

phase_expr
  for phase=1 to (levels-1)
    down_tree(phase) |>
  for phase=levels to 2
    up_tree(phase);
```

Figure 3-7: LaRCS code for leveled execution in a complete binary tree.

power of expression is exactly what dynamic task graphs provide.

It is possible to extend the temporal communication graph to support the dynamic task graph model. The static task graph is replaced with a *potential* task graph. This graph contains nearly the same information as the static task graph, however, the processes in the graph are considered in one of two ways. Every process is either a parent process, which is created statically at compile time, or a descendant, which may potentially be created at run time. The process-time graph retains the same structure but now processes may appear or disappear from one logical time step to the next. The processes present at the initial time step are the parent processes and those which appear at some time step but did not exist on the previous time step are descendant processes. It should be clear that if no descendants are ever created and no processes are ever deallocated, this model is identical to the original temporal communication graph model presented above.

A dynamic system may generally be more difficult to describe than a static system. However, by focusing attention on descibing computations which dynamically spawn tasks in a regular and predictable pattern, languages can be developed which are based on the dynamic task graph model. Essential to such a language is the ability to identify the parents and their spawned descendants in the potential task graph. Further, the language should have constructs which express when processes are born and when they die as well as when they are busy, idle, or communicating with other processes. In fact, the authors of LaRCS plan to expand it to include notation for describing dynamically evolving parallel computations in a similar way [8].

3.1.6 The PRAM Model

The parallel random-access machine (PRAM) is a popular theoretical model for presenting parallel algorithms [2]. If viewed on a spectrum measuring relative levels of abstract parallelism for the various models of parallel computation, where parallelization of sequential code lies toward the concrete end, then the PRAM model most definitely lies at the extreme opposite end of that spectrum. The PRAM is an abstract model of parallelism which unburdens the parallel algorithm designer from worrying about processor interconnection networks, bandwidth, and memory organization. The hope is that in using

the PRAM model the essential attributes of the parallel algorithm will transcend any less abstract model of parallel computation that might be more closely related to real world parallel computers. This simply means that if one PRAM algorithm outperforms another, the relative performance for comprable adaptations to other models should not change substantively.

The basic PRAM model consists of a set of processes which share a large global memory. All processes can read and write to the global memory in parallel within a fixed number of steps that is independent of the number of processes reading or writing the memory at the same time. In this model, processes no longer communicate via explicit message passing as they did in the graph models. Instead, processes communicate with one another implicitly through the shared memory. A process sends a message to any other process by writing to an agreed upon memory location that will later be read by the process receiving the message. This model is nearly equivalent to a fully-connected graph but is actually more powerful because messages need not be explicitly sent to any particular receiver; once data is in the shared memory, any process is free to read it. This model is very powerful and is able to capture ideas of abstract parallelism that are physically impossible to implement.

Suppose several processes wish to read or write the same memory cell on the same cycle. An *exclusive-read* algorithm is one in which no two processes ever read the same memory cell at the same time; whereas, a *concurrent-read* algorithm allows multiple processes to read a single memory cell on the same cycle. The same distinction is made with respect to multiple processes writing a single memory cell at the same time. Thus, four different varieties of PRAM algorithm models arrise based on the ways of dealing with these situations. The four types are commonly known as

EREW Exclusive Read Exclusive Write,

CREW Concurrent Read Exclusive Write,

ERCW Exclusive Read Concurrent Write, and

CRCW Concurrent Read Concurrent Write.

The most popular types of PRAM algorithms are at the extemes, EREW and CRCW. The EWER PRAM is a popular model because it comes closest to modeling real machines, while the CRCW PRAM provides a programming model that is arguably more straightforward.

The remaining models, CREW and ERCW, provide little additional power or flexibility and so it is convenient to think of any algorithm which requires either concurrent reads or writes as being CRCW. The CRCW PRAM has been shown to be more powerful than the EREW PRAM, but it has been shown that a p-process CRCW algorithm can be no more than $O(\log p)$ times faster that the best p-process EREW algorithm for the same problem [2]. Further, every algorithm expressible by one model can be expressed by the other.

3.2 Languages for describing parallel computations

The goal for a model of parallel computation is to capture the parallelism of an algorithm while providing the designer with a model that is easy to work with. The PRAM model succeeds at both of these but the level of abstract parallelism it achieves translates to ignoring several important aspects of real parallel machines. Additionally, many algorithms are much easier to consider in terms of some concrete representation and therefore it is often easier to develop parallel algorithms in some graph model and then generalize them to the PRAM model. For these reasons, the PRAM model may be more useful as a theoretical tool than as a practical model for parallel algorithm development and design. None of the other models, however, is as successful in the goal of capturing parallelism. The parallelization of sequential code leaves the task of discovering parallelism entirely to the compiler. Although this model may be the easiest for the programmer, it is unable to produce satisfactory parallel algorithms. The graph models of parallel computation capture most of the ideas of parallelism and are easy to work. There restriction in capturing parallelism is do to their relationship to some fixed interconnection network which the graph models. This type of network does not allow the use of a one-step parallel broadcast which can easily be described by the PRAM model. However, a one-step parallel broadcast and similar ideas of abstract parallelism are not practical for real machines and so the restrictions to the graph models are of little practical importance. For these reasons, graph theoretic models of parallel computation provide the greatest balance of expressive capability and conceptual simplicity and therefore should be used as the underlying model for parallel languages.

Now that several parallel programming models have been examined and one has been suggested as a basis, it is time to consider what concepts a language for describing parallel computations should be aimed at capturing. Consider again sequential languages and compare Figs. 3-8 and 3-9. As these figures clearly demonstrate, C and Lisp have widely differing syntactic, semantic, and notational conventions as well as separate data storage models. Notice, however, what the languages have in common: the basic structure of the QuickSort algorithm remains almost identical for both the C and Lisp versions. It is this observation that motivates the desire for a parallel programming language to first and foremost capture the structure of the algorithm, putting issues of syntax, semantics, and notation aside.

```
quicksort ( int *a, int first, int last )
{
  int mid;

  if ( first < last ) {
    mid = partition ( a, first, last );
    quicksort ( a, first, mid );
    quicksort ( a, mid+1, last );
  }
}

int partition ( int *a, int i, int j )
{
  int temp;
  int x = a[i];

  while ( TRUE ) {
    while ( a[i] <= x ) i++;
    while ( a[j] > x ) j--;
    if ( i < j ) {
      temp = a[i];  a[i] = a[j];  a[j] = temp;
    }
    else
      return j;
  }
}
```

Figure 3-8: C code for sequential QuickSort algorithm.

```
(defun quicksort (a)

  (defun partition (a x)
    (defun part (a small large)
      (if (null a)
          (cons small large)
        (if (> (car a) x)
            (part (cdr a) small (cons (car a) large))
          (part (cdr a) (cons (car a) small) large))))
    (part a nil nil))

  (if (> (length a) 1)
      (let ((part (partition a (car a))))
        (append (quicksort (car part))
                (quicksort (cdr part))))
    a))
```

Figure 3-9: Lisp code for sequential QuickSort algorithm.

The section focuses on languages which are best at capturing the structure of parallel algorithms. The details of the computation and logic occuring within individual processes are not discussed here because individual processes may be programmed following well-known models of sequential computation with few additional constructs for message passing and synchronization. Here, instead, the primary considerations are structure and organization of the global behavior of parallel algorithms.

3.2.1 Graph Description Languages

Graph description languages are the generalization of task communication languages. In other words, a task communication language is a graph description language being used for the purpose of describing a parallel computation. As mentioned earlier, the nodes of the graph can be viewed as representing processes in the computation and the edges as representing messages sent between a pair of processes. Graph description languages are closely related to the graph theoretic models of computation presented in the previous sections.

3.2.2 TTDL

Task Traffic Description Language (TTDL) [12] is a simple intermediate-level language developed specifically as the front end interface to a scheduled routing system for the NuMesh. The language is implemented in Common Lisp and describes communication patterns in a Lisp-like syntax. Execution of a TTDL program results in the creation of a process-time graph which serves as input to the mapping system. Designed specifically as an intermediate-level language, TTDL was intended by its author to perhaps eventually be generated by a some compiler of a high-level language. While this approach may be feasible, a better approach might be to use a high-level language which itself is capable of adequately describing communication patterns. This alternative approach is the one suggested in this thesis.

3.2.3 LaRCS

LaRCS [8] was developed in conjunction with the OREGAMI project [7] as a language for describing parallel computations for the purpose of mapping. It is a graph description language based on the temporal communication graph model which allows programmers of parallel algorithms to specify the regular communication topology and temporal communication behavior of parallel algorithms. Designed as a Language for Regular Communication Structures, LaRCS provides an efficient and intuitive notation enabling it to efficiently describe families of computation graphs. Thus, for any regular communication graph such as a tree or mesh, the size of the LaRCS code remains constant for all graphs of the same family, independent of the number of nodes or edges in a particular instance of the graph family.

The OREGAMI project as a whole is aimed at the comparative analysis of algorithms for mapping parallel computations to message-passing parallel architectures. This is to say that OREGAMI allows easy comparison of various proposed solutions to the mapping problem. OREGAMI was not designed to map parallel computations to any particular machine nor is it intended as a development environment for parallel computations. Nonetheless, due in large part to the expressive power of LaRCS, the system might successfully be adapted for such purposes. As the system exists now, there are no provisions

for mapping to real machines, but as a software development environment LaRCS and OREGAMI provide a very good start.

There are four main constructs found in LaRCS programs. The *nodetype* declaration which describes the processes and labels them, occurs near the beginning of a LaRCS program. Next are the *comtype* and *comphase* declarations which are used as templates for describing communication edges and sets of synchronous communication edges which form communication phases. The *compute* declaration is how LaRCS accounts for when a process is performing any computation. The language currently does not support any way of describing the details of the computation but does allow specification of a measure of time the computation is expected to require relative to other computations in the program. Finally, LaRCS provides the *phase_expr* declaration which describes the entire parallel program in terms of its relative temporal computation and communication behavior.

Figure 3-10 is a simple LaRCS program modeling a parallel merge-sort algorithm on a binary tree. The behavior of the algorithm can be discerned entirely from the `phase_expr` declaration. Note that the `for` loop indicates sequential iteration whereas the `forall` construct indicates parameterized parallel execution of all selected constructs. The first loop iterates from the root level of the tree to one level above the leaves and at each level splits its data then sends one half down to each child. Once the data reaches the leaves, each leaf sorts the data it receives via some sequential algorithm. Then the data is sent back up the tree where each parent merges the data it receives from its children. The result at the root will be the sorted data. The LaRCS code should be straightforward to follow.

As a language for describing parallel computations to be mapped to parallel machines, LaRCS proves extremely useful. First, it is easy to construct the static task graph from the LaRCS description. The static task graph is the graph that results when all the communication phases are active simultaneously. Thus the compiler can construct the static task graph by taking the union of the graphs resulting from every communication phase specified in the phase expression. Second, the process-time graph is simple to construct from a LaRCS program. Every sequential step in the phase expression represents one time step in the process-time diagram; all parallel activity occurs in the same time step. Communication between a pair of nodes (processes) is assumed to begin in the time step

```
merge_sort(levels)

nodetype node labels 1..2**levels-1;

comtype l_child(i,v)    node(i) => node(2*i); volume = v;
comtype r_child(i,v)    node(i) => node((2*i) + 1); volume = v;
comtype parent(i,v)     node(i) => node(i / 2); volume = v;

comphase down(k)
  forall i in (2**k)..(2**(k+1))-1
    { r_child(i, 2**(levels-k-1));
      l_child(i, 2**(levels-k-1)); }

comphase up(k)
  forall i in (2**k)..(2**(k+1))-1
    { parent(i, 2**(levels-k)); }

compute split(k)
  forall i in (2**k)..(2**(k+1))-1
    { node(i) volume = 2**(levels-k-1); }

compute sort()
  forall i in (2**(levels-1))..(2**levels)-1
    { node(i) volume = 1; }

compute merge(k)
  forall i in (2**k)..(2**(k+1))-1
    { node(i) volume = 2**(levels-k-1); }

phase_expr
  for k = 0 to levels-2
    { split(k) |> down(k) } |>
  sort() |>
  for k = levels-1 to 1
    { up(k) |> merge(k-1) };
```

Figure 3-10: LaRCS code of parallel merge-sort algorithm.

of the phase where it is specified and end in the next time step. These two facts alone mean that a LaRCS program provides all the necessary information for a system to perform optimal contraction, placement, and routing in mapping the program to a parallel machine. Equally impressive about LaRCS are its natural and intuitive constructs which lead to ease of use and understanding by programmers. Therefore, LaRCS fulfills the requirements for a task communication language.

3.3 LaRCS as an Implementation Language

Despite its strengths, LaRCS was not intended to describe parallel computations for the purpose of *implementation*, and as such it requires additional mechanisms before it can be used successfully as an implementation language. Certainly, LaRCS could benefit from a macro preprocessor. The most obvious omission, however, is in its inability to express the details of computation. LaRCS was designed to be used in conjunction with some other parallel programming language whereby the LaRCS code would model the actual parallel computation and the OREGAMI system could be used to determine the best way to map the computation onto the machine for which it was implemented. However, by augmenting the computation phase to include the actual process code in a given phase for all processes active during that phase, LaRCS could itself become a viable implementation language.

The traditional approach to programming a parallel computer is to write the code of each process and embed within that code the communications with other processes. This seems natural since all the code for a process is written together to make up that process. The alternative being suggested is that computation phases be coded as separate entities. Thus, the entire code for a single process then will be found scattered among all the computation phases in which that process is active. This idea and its application should actually make the implementation of parallel algorithms easier because the programmer no longer has to worry about what every process is doing during every step of the computation. Instead the programmer can focus on the algorithm and what logically happens in each phase of the computation. By focusing attention on the algorithm as a whole rather than

the individual processes used to perform the algorithm, the programmer may avoid dealing explicitly with timing and synchonization of communication and may ignore when processes are idle. These details can be figured out by a compiler freeing the programmer from this burden.

Consider by analogy a conductor and an orchestra. The conductor represents the programmer, the orchestra is the set of parallel processes, and the piece of music is the details of the algorithm. The conductor could attempt to direct the orchestra by directing each member of the orchestra simultaneously. This might be ideal for members of the orchestra since they would be given constant and explicit instructions on how to play throughout the entire piece of music. It goes without saying that the conductor would have a much more difficult go at it. Instead, the conductor examines the music and directs certain members of the orchestra at certain times while focusing on other members at other times. In this way, the conductor is able to be merely mortal and still direct the entire orchestra, while members of the orchestra are given enough information to play the whole piece successfully without explicit direction for every note. In the same way, a programmers job should be much easier by focusing on programming the algorithm by instructing various processes only when they are active rather than by explicitly providing instructions to every process for the entire lifetime of the algorithm.

Returning to the merge-sort example, Fig. 3-11 shows sample pseudo-code for merge-sort intended as a single instruction stream for every process. In the code, *pid* refers to the unique process identifier and *root*, *parent*, *lchild*, and *rchild* are intended as symbolic names for the appropriate process identifiers. This description of merge-sort as a parallel algorithm is incomplete because, while the programmer may clearly intend it to be implemented with a binary tree of processes in divide-and-conquer fashion, a compiler can not possibly know this. The compiler must be supplied with that information in a form such as the existing LaRCS merge-sort code from Fig. 3-10. Consider instead, if the pseudo-code were embedded within the LaRCS code. This model allows the programmer to write one source combining communication and computation phase information and ultimately provides a simpler model for implementing parallel algorithms. Figure 3-12 shows the LaRCS augmented with pseudo-code for the parallel merge-sort algorithm. The particular

```
merge_sort_process()
  1. IF pid = root THEN
  2.   list <- the input array
  3. ELSE
  4.   WAIT(parent)
  5.   list <- RECEIVE(parent)
  6. n <- LENGTH(list)
  7. IF n > 2  THEN
  8.   SEND(lchild, list[1..(n/2)])
  9.   SEND(rchild, list[(n/2)+1..n])
 10. ELSE
 11.   IF list[1] > list[2] THEN
 12.     SWAP list[1] <-> list[2]
 13.   SEND(parent, list)
 14. WAIT(lchild, rchild)
 15. lista <- RECEIVE(lchild); a <- LENGTH(lista)
 16. listb <- RECEIVE(rchild); b <- LENGTH(listb)
 17. i <- j <- k <- 1
 18. lista[a+1] <- listb[b+1] <- infinity
 19. WHILE (i <= a) or (j <= b) DO
 20.   IF lista[i] < listb[j] THEN
 21.     newlist[k] <- lista[i]; i <- i + 1
 22.   ELSE
 23.     newlist[k] <- listb[j]; j <- j + 1
 24.   k <- k + 1
 25. IF pid = root THEN
 26.   the output array <- newlist
 27. ELSE
 28.   SEND(parent, newlist)
```

Figure 3-11: Pseudo-code for parallel merge-sort algorithm.

choice of an augmentation language is relatively unimportant and perhaps could be one of several languages which the programmer could choose.

```
merge_sort(levels)

nodetype node labels 1..2**levels-1;

comtype l_child(i,v)   node(i) => node(2*i); volume = v;
comtype r_child(i,v)   node(i) => node((2*i) + 1); volume = v;
comtype parent(i,v)    node(i) => node(i / 2); volume = v;

comphase down(k)
  forall i in (2**k)..(2**(k+1))-1
    { r_child(i, 2**(levels-k-1)); l_child(i, 2**(levels-k-1)); }

comphase up(k)
  forall i in (2**k)..(2**(k+1))-1
    { parent(i, 2**(levels-k)); }

compute split(k)
  forall i in (2**k)..(2**(k+1))-1
    { node(i):  1. IF pid = root THEN
                2.   list <- the input array
                3. ELSE
                4.   list <- RECEIVE(parent)
                5. n <- LENGTH(list)
                6. SEND(l_child, list[1..(n/2)])
                7. SEND(r_child, list[(n/2)+1..n])  }

compute sort()
  forall i in (2**(levels-1))..(2**levels)-1
    { node(i):  1. list <- RECEIVE(parent)
                2. IF list[1] > list[2] THEN
                3.   SWAP list[1] <-> list[2]
                4. SEND(parent, list)          }

compute merge(k)
  forall i in (2**k)..(2**(k+1))-1
    { node(i):  1. lista <- RECEIVE(l_child); a <- LENGTH(lista)
                2. listb <- RECEIVE(r_child); b <- LENGTH(listb)
                3. i <- j <- k <- 1
                4. lista[a+1] <- listb[b+1] <- infinity
                5. WHILE (i <= a) or (j <= b) DO
                6.   IF lista[i] < listb[j] THEN
                7.     newlist[k] <- lista[i]; i <- i + 1
                8.   ELSE
                9.     newlist[k] <- listb[j]; j <- j + 1
               10.   k <- k + 1
               11. IF pid = root THEN
               12.   the output array <- newlist
               13. ELSE
               14.   SEND(parent, newlist)                    }

phase_expr
  for k = 0 to levels-2
    { split(k) |> down(k) } |>
  sort() |>
  for k = levels-1 to 1
    { up(k) |> merge(k-1) };
```

Figure 3-12: LaRCS augmented with pseudo-code of parallel merge-sort algorithm.

Chapter 4

The Mapping Problem

A specific type of parallel programming problem is in consideration for this thesis and the NuMesh in general. The problems being looked at are ones which may be characterized abstractly as a static set of communicating parallel processes. The mapping problem involves the assignment of these processes to physical processors and the routing of logical communication along the physical interconnection network of the machine.

Until recently, most parallel processing systems relied on the programmer to do the mapping and upon a dynamic routing system that did not utilize information about the communication patterns of the the computation. Systems such as Prep-P [1] and OREGAMI [7] attempt to automate the mapping problem by providing modules to perform the three main functions needed in the mapping problem. This chapter presents a broad overview of the problem and presents many possible solutions with some benefits and drawbacks of each listed.

4.1 Abstract View of the Mapping Problem

4.1.1 Three Phases of the Mapping Problem

Given a characterization of a parallel computation and of a parallel architecture, a system automatically solving the mapping problem needs to do contraction, placement, and routing. For an abstract view of the mapping problem, see Fig. 4-1. Contraction is the phase

in which the parallel processes are grouped together into clusters until there are at most as many clusters as processors. A computation may abstractly have many cooperating processes while a given machine will only have a fixed number of processors. The processes within a cluster will have to share the single processor which they are assigned to in the placement phase. The routing phase of the mapping problem has traditionally been done at run-time in a dynamic fashion. In this way, logical communication between tasks is not assigned any fixed physical data path, but rather the path is variable and generated for each message as it travels through the network. The NuMesh approach is to use static (or scheduled) routing whereby the data path for each message is determined at compile time.

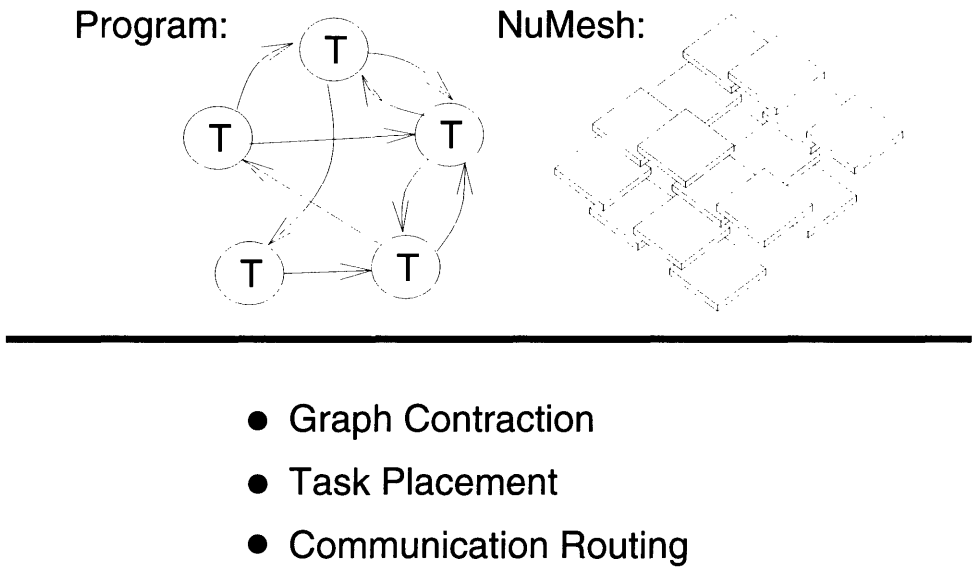

Figure 4-1: An abstract view of the mapping problem.

4.1.2 Goal of the Mapping Problem

The goal of the mapping problem is to map a parallel computation onto a parallel machine in a way which achieves the most efficient use of resources while performing the computation in the most efficient way. In almost every case, both halves of the goal are equivalent. That is to say, if the machine's resources are being used efficiently then that computation will proceed swiftly, and if the computation is running efficiently, that it is likely the case that the machine's resources are being used in a balanced and cost effective manner. This

is exactly as one would expect.

4.1.3 Solving the Mapping Problem

The mapping problem is not a trivial one to solve. It is convenient to consider the problem as having the three distinct phases: contraction, placement, and routing. Abstractly, the phases would be run independently and in sequence to provide a nearly optimal solution. The contraction phase would have the goal of distributing the processes in an equal fashion so that all processors would be assigned n processes before any would be assigned $n + 1$ processes. The placement phase could simply assign a physical processor number to each of the clusters generated in the first phase. Routing would then be accomplished by using the Floyd-Warshall algorithm for finding the all pairs shortest paths solution for all messages which must communicate at logically synchronous times. While this is a nice model and one that may be reasonable to implement, it is often naive in its approach. Figure 4-2 shows the mapping problem phases during an example of mapping an application to an architecture.

Contraction, placement, and routing are more likely to provide an optimal solution to the mapping problem if they are computed together in a cooperative manner. To see why this is so, consider the metrics which are being optimized: efficient use of machine resources and efficient computation.

The metric for computational efficiency is generally either time or space: how long does the computation take and how much memory does it require. Neither of these can be directly mapped to the problems of contraction, placement, or routing. Instead, consider the metrics used to determine efficient use of resources. These will generally be the ratio of busy to idle processors and the total volume of network traffic. For a constant amount of work, a higher busy to idle ratio indicates that more work is being done in parallel and that each processor will become free sooner since it has less work to do. Since network communication is more costly than accessing data local to a processor, a lower overall volume of network traffic will generally lead to faster computation and also require fewer network resources, thereby reducing congestion. This set of metrics has an immediate mapping to the contraction, placement, and routing problems.

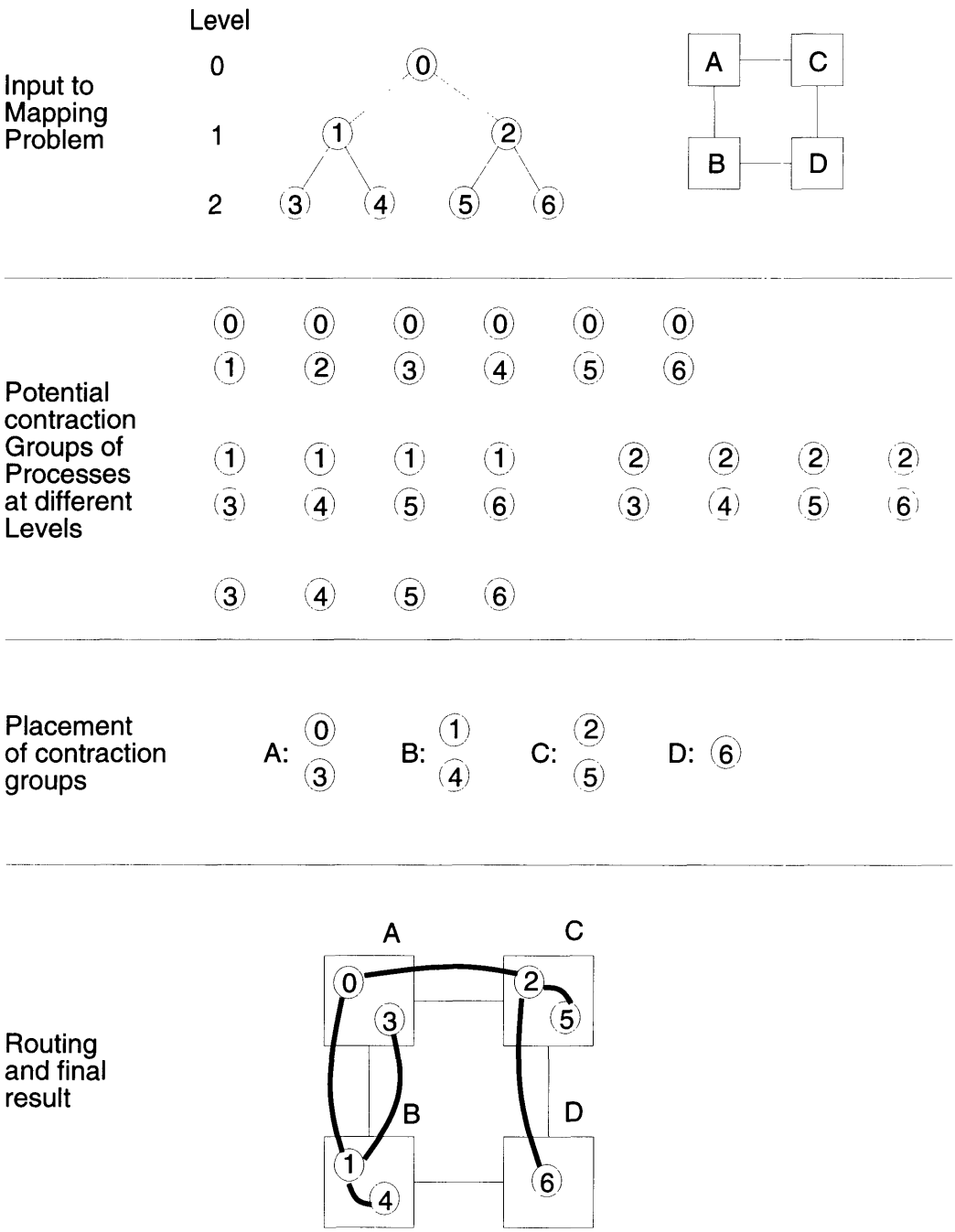

Figure 4-2: An example of the mapping problem's phases.

4.2 Contraction

Given that a higher busy to idle ratio is better, how should contraction proceed? Here, the goal should be to not group together processes which are active at logically synchronous times. This won't always be possible because in some computations all processes are busy all the time, but in computations such as merge sort which exhibit a leveled execution structure (i.e. where all processes at a level compute simultaneously), contraction which accounts for the dynamic behavior of the computation may have clear benefits. Additionally, it is possible to minimize network traffic by grouping together processes which are active at different times but communicate directly with one another between the different computational phases.

Several algorithms have been developed to perform contraction. Some of these algorithms fail to directly account for the metrics listed above but have proven themselves useful in systems where dynamic behavior of the parallel computation is not available at compile time. These algorithms consider the parallel computation only in terms of its static task graph. One such algorithm attempts to minimize the total interprocessor communication volume for a fixed maximum number of processes per processor [7]. This works in a greedy fashion by grouping processes together that have the largest communication volume between them. This maximum weighted matching more or less continues until the number of clusters is less than or equal to the number of processors.

Another contraction algorithm based upon group theory is capable of working only on parallel computations whose static task graph is a Cayley graph [7]. This algorithm does take advantage of the computation phases because it examines the communication phases in determining the generators of the group from which the Cayley graph is derived.

Neither of these existing algorithms is fully satisfactory because some are restricted in scope while others they do not take advantage of the dynamic behavior of the parallel computation. To remedy this situation a new algorithm is proposed and presented in a high-level conceptual form. The algorithm takes advantage of the dynamic behavior in a completely general way and then defaults to a maximum weighted matching solution if a complete contraction has not been generated.

1. Choose a load balancing constraint. This is the maximum number of processes per processor and thus the maximum number of processes which can be grouped in the same cluseter. A good value for this will often be $1 + \#\ processes/\#\ processors$.

2. Partition the processes into groups where every member of each group is busy during the same logical phase of computation. In other words, all the processes busy during the first phase are placed in a group, all the processes busy during the second phase are placed in another group, and so on. Note that these groups do not necessarily form disjoint sets of the processes.

3. Create clusters of processes subject to the load balancing constraint. Let n be the maximum number of processes allowed per cluster by the load balancing constraint. Then clusters can be created by choosing one element from each of up to n different groups such that no element chosen for the cluster appears in more than one of the n groups from which the elements were chosen. Note that if the groups form disjoint sets this additional constraint is not a concern.

4. Organize the clusters into groups. The previous step creates all possible clusters of processes which do not compute during the same phase. The obvious constraint in selecting which clusters to use is that each process should occur only once in the set of clusters chosen. This immediately suggests that the clusters be grouped according to one of the elements in the cluster. This will result in a group of clusters which all contain process 0, a group of all clusters which contain process 1 except if the cluster is already in the first group, and so on.

5. Select the clusters to use for this contraction. Choose a cluster from the first group. Now eliminate from consideration all other clusters which contain any process contained in this cluster. Continue by selecting a cluster from a remaining group and prune the remaining clusters as before. This procedure will obtain one possible contraction. Repeat it, choosing a different cluster from the first group to obtain an alternate contraction. Any contraction which has the exact number of clusters as processors and includes every process is an optimal contraction.

6. If an optimal contraction is not generated by the preceding step, choose a contraction with the largest number of clusters. In this case there will processes which are not in any cluster. Some set of heuristics should now be used to determine which set of clusters will produce the best contraction. An effective method is to use the maximum weighted matching solution as mentioned above. This will choose a set of clusters which will attempt to minimize interprocessor communication among the clusters which have already been selected as well as the processes not yet belonging to any such cluster.

This contraction algorithm is only useful for parallel computations which proceed with distinct computation phases that utilize differing sets of processes. If a parallel computation exhibits behavior where all processes are always busy, then this algorithm will default to a maximum weighted matching procedure and may not produce an optimal solution. However, this algorithm could easily be modified to account for more sophisticated load balancing considerations that might, for instance, take into account the load a particular process would place on a particular processor. In such cases, clusters would be selected that created a lower maximum load over all time.

Still other algorithms exist for contraction [1]. Because none of these algorithms is sensitive to computation phases, any of them may be used as a stand alone algorithm or as a replacement for the maximum weighted matching solution in the algorithm presented above. The first such algorithm is an oblivious *block* contraction algorithm. In the block algorithm, consecutive processes or clusters are grouped together. This simplest of all algorithms turns out to be surprisingly effective in many cases. The reason for this algorithm's effectiveness is probably because many parallel computations are designed with nodes ordered according to some perceived notion of locality by the programmer. Two similar contraction algorithms are *cyclic* and *random*. In the cyclic algorithm process i is assigned to processor (i mod N) while in the random algorithm processes are randomly assigned to processors. These algorithms typically have poor performance, but they can provide good control cases against which to measure more sophisticated algorithms. The remaining two algorithms are closely related to one another and both are sensitive to communication volume as was the maximum weighted matching algorithm. These are general optimization

algorithms applied to the problem of contraction. The first is *simulated annealing* which is described in Sec. 4.3.1 and the second is *local neighborhood search* [1] which is really just a special, simplfied case of simulated annealing where temperature is not a factor and no bad moves are ever accepted.

4.2.1 Another Way of Viewing Contraction

In the above algorithms for contraction, the goal was to produce a contraction for a particular architecture. This grew out of the desire to optimize with respect to the processor busy to idle ratio. Another way of viewing contraction is purely in terms of the abstract static task communication graph. In fact, this is exactly the way the Cayley contraction mentioned above works. The goal in this approach is to produce a graph with the same characteristics as the original graph but with fewer nodes. Graph theoretic techniques which take advantage of a graph's symmetry or recursive structure can effectively contract certain classes of graphs. Unfortunately, these techniques are not applicable to general graphs where contraction only makes sense with respect to some other graph, the target architecture.

All contraction procedures completely determine the processor busy to idle ratio. So only the metric of network traffic volume remains to be minimized during the placement and routing phases.

4.3 Placement

Once a successful contraction has been found, each cluster must be assigned to a physical processor. This process is known as placement or embedding. The goal in doing placement is to assign clusters to processors such that the overall weighted message dilation is kept to a minimum. Achieving this goal is equivalent to optimizing resource usage. A successful placement will certainly contribute to a lower network traffic volume because each message will be in the network for less time on average.

A simple algorithm for doing placement uses a nearest-neighbor greedy approach [7]. This method places highly communicating clusters on adjacent neighbors in the network.

Because the algorithm uses a greedy approach, it is possible for it to find a non-optimal placement in terms of minimizing the total weighted dilation. The algorithm works by first sorting the weighted edges in the graph formed by the clusters. The edges are traversed once in linear fashion and the clusters at the ends of the edges are assigned to the closest available processors until all clusters have been assigned to a processor. The advantages of this algorithm are that it is easy to understand and implement.

Kernighan and Lin developed a divide-and-conquer circuit placement algorithm in [3] which may be applied to the placement of processes onto an interconnection network [1]. The algorithm begins with a random placement of clusters onto the network which, for purposes of description, will be an $n \times n$ mesh. Each recursive iteration first determines the four subgraphs of the cluster graph which have the lowest total number of edges between them. These subgraphs can be determined using methods similar to those used for contraction; a maximum matching solution should prove adequate. Next all $4! = 24$ possible placements into the quadrants are considered and evaluated based on the total distance between communicating processes. Every process is considered to be at the center of its quadrant during this computation since its exact location is not yet known. The algorithm continues recursively on each quadrant. This algorithm can be easily generalized to placing clusters onto any physical network which has a recursive structure.

Most of the techniques mentioned in the section on contraction may be applied in some form to the placement problem as well. Block and cyclic placement degenerate to simply placing cluster i onto processor i since there are guaranteed to be fewer than or an equal number of clusters as processors. Random placement works as expected. Maximum weighted matching is closely analogous to the nearest neighbor greedy approach described above. Simulated annealing (and its simplification to local neighborhood search) has been shown to be quite effective in placement and deserves a more detailed description.

4.3.1 Simulated Annealing

This algorithm also begins with a random embedding but then uses the method of simulated annealing [4] in its attempt to minimize total weighted dilation. Simulated annealing works by evaluating and attempting to minimize a cost function via an iterative process.

A cost function for placement could be the total weighted dilation. At each step of the iteration, a random pair of processors is chosen and the effect of interchanging the clusters assigned to each is considered. The clusters are always exchanged if the move results in lowering the cost function; if the move increases the cost function, there is a probability based on the *temperature* of the system that the exchange will still be allowed. This probability p of acceptance for cost function difference δ and temperature t is given by the Boltzmann distribution:

$$p = e^{\frac{-\delta}{t}}$$

The process begins at a high temperature which essentially *melts* the embedding by allowing random moves to take place. After some number of exchanges are attempted, the temperature is gradually lowered and the some number of exchanges are attempted again. This continues until the embedding becomes *frozen* which occurs when the temperature has reached some minimum and the cost function stabilizes around some minimum value.

Simulated annealing attempts to model the physical annealing process whereby a system is first brought to an excited (high temperature) state and then gradually cooled. This process will tend to order a physical system and thereby reduce its total energy. In the same way that annealing can be done to metal or glass to make it less brittle, simulated annealing can be used to make interprocessor communication less costly during the runtime of a parallel computation.

Because this algorithm randomly chooses processors to consider and has the ability to accept a move that increases the cost function, it is unlikely that it will settle on a local minimum. The algorithm does have the drawback of being non-deterministic in nature and so it is not guaranteed to find a solution. Further, this implies that no useful upper bound exists on the time required to find an optimal solution. However, in all but the most unlucky cases, the simulated annealing approach will eventually arrive at an optimal solution.

It is worth mentioning that while simulated annealing is a powerful optimization tool for solving the mapping problem, it may also be expressed as a parallel computation which would benefit from the techniques described in this thesis. Abstractly, in the parallel version of simulated annealing, there is one global configuration of the system that each

process looks at. But now multiple processes attempt simultaneous exchanges of disjoint pairs of, in this case, clusters. Each process randomly chooses two clusters and locks them so that another process cannot also choose either of them. The cost function is computed for each potential exchange with a snapshot view of the system at the time the clusters were locked and compared with the global cost function value which may be updated at any time and does not necessarily represent the *previous* cost as in the sequential case. This automatically adds some chaos and non-determinism to the system because many clusters may be swapped at a logically synchronous time without knowledge of any of the other exhanges. Again, this non-determinism makes it highly unlikely that the system will settle on a local minimum.

4.3.2 Combining Contraction and Placement

As stated earlier in this chapter, the best way to arrive a good solution to the mapping problem may be to compute contraction, placement, and routing in some cooperative manner rather than as distinct processes. Now is a useful time to notice that contraction and placement may be combined into one operation. A combined approach to contraction and placement has more flexibility in assigning logical processes to physical processors. When contraction is done in isolation, the topology of the network is not considered because it is assumed that the placement phase will handle that. While placement done in isolation ignores processor load constraints and interprocess communication volume by assuming that contraction has taken care of that. In truth, these problems are not independent and therefore benefit from a solution which considers them simultaneously.

Simulated annealing provides a simple way of doing contraction and placement at the same time. The idea is to come up with a cost function which takes into account processor load and message dilation simultaneously. Pairs of individual processes may then be randomly considered for exchange based on the value of the cost function at each iteration. The details of this process are described below.

A critical element in making this algorithm successful is the cost function. Any time two processes which are active on the same phase of the parallel computation are placed together, the cost function should increase. Likewise, the cost should increase when highly

communicating processes are placed further apart. Clearly, many such functions fit these criteria. The more accurately processor load and communication volume and contention are modeled the better the solution generated by this approach. However, more accurate modeling usually makes it more expensive to calculate the cost function. This could greatly affect the performance of the contraction and placement being done here.

A relatively simple cost function provides an inexpensive, yet effective, computation for doing contraction and placement. The cost function takes the form

$$C_t = W_c C_c + W_p C_p$$

where C_t is total cost, W_c is the weight for contraction, C_c is the cost of the contraction, W_p is the weight for placement, and C_p is the cost for placement. The cost function for contraction will simply be

$$C_c = \sum_{p \in Processors} (\text{\# simultaneously busy processes on } p)$$

For placement, the cost function will be

$$C_p = \sum_{e \in Edges} V_e * (\text{min \# hops between start and end processors})$$

where V_e is the communication volume of the edge.

A second critical factor in this combined approach is determining which processes to consider for exchange during each iteration. The placement can begin with a random distribution of processes to processors subject to the load balancing constraint. At each iteration a pair of processors should be chosen at random. Now individual processes must be movable among processors otherwise the random clustering will persist. There must also be the ability to swap n processes on the first processor with m processes on the second processor of the pair provided that this exchange does not violate the load balancing constraint. Without this ability, the random sizes of the clusters would persist.

Thus, once the processor pair has been randomly chosen, a random set of processes should be chosen from each processor of the pair. In other words, for each process on both of the processors, randomly decide whether that process should be moved (flip a coin for each process). This will result in a set of processes, which could be empty, from

each processor to be moved. If one of the sets is larger than the other, the exchange could violate the load balancing constraint for one of the processors. If this happens, either choose new sets as before or throw away a process from the set which is too large. The second method is probably better because the choosing new sets may result in the same problem many times. Once suitable sets have been chosen, calculate the cost function and make the exchange according to the simulated annealing algorithm.

The mapping problem for a general system is known to be NP-complete. This is why the generic simulated annealing approach was chosen rather than some algorithm specifically intended for mapping. Nonetheless, other more specific algorithms may be used for simultaneously solving the contraction and placement phases of the mapping problem. Many such solutions exist for restricted, but well-known cases. These so-called *canned mappings* exist for a number of popular logical topologies being mapped to familiar physical topologies. Canned mappings have been developed by researchers manually and have been shown to be optimal. Table 4.1 [6, 7] contains a partial list of canned mappings.

Besides being useful for the actual mappings they perform, canned mappings may provide useful insights into effective general mapping solutions. The goal of any general mapping algorithm should be to provide an optimal solution regardless of regularity. An algorithm's performance on a mapping problem where an optimal solution is known is often a good indicator of how well the algorithm will perform in general. Furthermore, analysis of where a generic algorithm fails to generate an optimal solution may provide insights in developing better generic mapping algorithms. This analysis is likely to show that certain cost functions result in better mappings to particular topologies. A compiler with such knowledge would prove to be extremely useful.

4.4 Routing

Much of the previous section discussed the mapping problem as if routing were not a part of it. In many ways, this may actually be the case. A prevalent way of viewing a parallel computer is as a collection of (perhaps heterogenous) processors connected to one another by some fixed interconnection network. The machine operates with each processor doing

- An n level complete binary tree to an n dimensional hypercube.
- An n level complete binary tree to an $(n-1)$ dimensional hypercube.
- A complete binary tree to a mesh.
- An n level complete binomial tree to an n dimensional hypercube by Gray code labeling.
- An n level complete binomial tree to a $2^{n/2} \times 2^{n/2}$ mesh if n is even, or a $2^{\lfloor n/2 \rfloor} \times 2^{\lceil n/2 \rceil}$ mesh if n is odd by Gray code reflection.
- A complete binomial tree to a deBruijn graph by combinatorial shift register sequences.
- A mesh to a hypercube where each dimension of the mesh is a power of 2.
- A mesh of trees to a hypercube.
- A pyramid to a hypercube.
- A 1-D array to a mesh of arbitrary dimension.
- A chordal ring to a hypercube.
- A ring to a hypercube by Gray code reflection.
- A ring to a mesh.
- A ring to a deBruijn graph by enumeration of necklaces or shift register sequences.
- A 2^{n-1} node xtree to an n dimensional hypercube.

Table 4.1: Selected canned mappings

some relatively independent computation and communicating with other processors by sending messages onto the network to various destinations. The responsibility for routing messages to their destinations rests with the network itself. In this environment, routing is done not as part of the mapping problem but rather as part of the run-time environment. This is a familiar dynamically routed system.

In the case of a statically routed machine, however, routing must be performed as the final stage of the mapping problem. As with contraction and placement, it is best to attack routing by considering what metrics are to be optimized. An effective routing is one which minimizes total weighted network traffic volume and contention. This means that messages should attempt to always take the shortest possible path while avoiding using any network link that is simultaneously in use by another message. A very useful observation is that minimizing the total time messages spend in the network is equivalent to minimizing network volume and contention. This is because using fewer network links and not having to wait for busy network links both reduce the time the message spends in the network.

One of the most obvious ways to compute routing is by using a greedy approach. For each message, choose the shortest possible path and then either ignore contention by waiting for the path to become available or select some other path to the destination. This solution is simple enough that it is what is often employed by dynamic routing systems. The problem with an approach like this is that each message is routed without considering the other messages in the system. Both oblivious and adaptive routing approaches like this are likely to fail to produce an optimal solution because neither takes advantage of global information and the problem does not exhibit the optimal substructure property.

A better approach to routing would be to consider all messages sent by the parallel computation throughout its lifetime. This then might be viewed as a problem extremely similar to the combined contraction and placement problem. The system starts with the set of all messages and the set of all possible paths between every pair of processors and attempts to place messages onto paths in a way that will minimized overall message volume and contention. Because the set of all possible paths between every pair of processors may be too many to consider, it will usually suffice to consider only all paths between

every pair of processors which have fewer than some fixed number of links. Simulated annealing is a suitable technique for solving this optimization problem just as it was in the case of contraction and placement. In fact, the same simulated annealing code module may be reused by simply passing in a different cost function.

Unlike placement, finding shortest paths is not an NP-complete problem and therefore more specific optimization techniques other than simulated annealing may be applied. An all pairs shortest path algorithm such as the one developed by Floyd-Warshall may be used here. Note that a weighted implementation of this algorithm must be employed here to allow a message to take a longer route than the *physically* shortest path in terms of number of links. In other words, once a path has been used, its weight should be increased to deter other messages from using it. The shortest paths being computed should ultimately be the shortest in *time* rather than physical links. The time for such an algorithm is generally $O(V^3)$ where V is the number of vertices in the graph for which shortest paths are being computed [2]. Thus, this type of solution may take a while to compute for large graphs but it is guaranteed to find the optimal solution and it exhibits a definite upper bound on run time whereas simulated annealing has no definite termination point in its execution.

An alternative to the all pairs shortest paths solution is a bi-partite matching solution [7]. This algorithm attempts to find a disjoint set of physical paths over which to route the messages. This can be accomplished by constructing a bi-partite graph where one partition consists of the communication edges from the task graph (these correspond to the messages) and the other partition consists of the available shortest potential routes in the interconnection network which could service these communication edges. Bi-partite matching then attempts to find a one-to-one mapping from the first partition to the second. This will generally not be possible when a large number of messages need to be routed on a relatively small number of potential paths. Instead, the bi-partite matching algorithm will find a many-to-one mapping with the least amount of contention. The running time for a bi-partite matching algorithm is $O(V \cdot E)$. The number of vertices is again described by V and E refers to the number of edges in the graph. For any graph, E is bounded above by V^2, so the bi-partite matching algorithm is of roughly the same complexity as the Floyd-Warshall algorithm. Unfortunately, bi-partite matching is greedy with respect

to individual links in the route. Therefore, while the algorithm is guaranteed to find the optimal routing of messages for any given step in the computation, a particular route chosen now may lead to contention at a later step in the routing.

Several other routing solutions have been explored and are worth mentioning [12]. *Force Driven Routing* is based on the idea of allocating paths by considering the trajectory of a moving particle in a force field created by point masses representing the source and destination. An advantage of this method is that routes are not dependent on the order in which the messages are considered. *Linear Programming* techniques have also been used to produce a global routing algorithm and may be especially effective in scheduling messages over busy links. Finally, a promising approach to solving the routing problem may be by a *Multiflow* formulation. Similar problems have been effectively solved using multiflow techniques.

There is an obvious improvement over considering all messages sent by the parallel computation throughout its lifetime. Just as parallel computations may have computation phases (as in leveled execution), they may also have communication phases. Instead of trying to route all messages simultaneously, just consider messages from each communication phase as independent routing problems. This greatly simplifies the routing problem regardless of which routing algorithm described above is used and allows for computing solutions much more expediently. By breaking the large routing problem down into independent problems, the time to solve the whole problem becomes the sum of the independent problems rather than their product. So any time a problem can be broken down into a set of independent subproblems, this approach should be taken, as it will always yield huge gains in efficiency.

4.4.1 Combining Placement and Routing

In the same way that contraction and placement could be combined into a single algorithm, placement and routing may also be combined. Placement and routing are very closely related because the set of possible routes for messages depends directly on where the source and destination processes are placed on the physical network. It is even true that certain embeddings of a given task graph may make it impossible for an optimal routing

solution to be found, whereas other placements easily allow for the discovery of such solutions. A simple example of this is to consider mapping a logical complete binary tree to a physical complete binary tree. The natural embedding is obviously to place the logical root at the physical root and so on. Such a mapping clearly makes things easy for the routing algorithm which can simply map logical edges to the corresponding physical links. However, if the logical root is embedded onto a physical leaf, then it will not be possible for any routing system, no matter what algorithm it uses, to produce the optimal, no contention solution generated by the more intelligent embedding.

Recall that in placement by simulated annealing, the sample cost function attempted to measure the total number of links traversed by all messages. The simulated annealing process worked by attempting to minimize this function. This system could be modified so that while computing the cost function, the path producing the least possible number of links be remembered. Only those paths affected by the proposed exchange would need to be recomputed. Such a system would simultaneously compute placements and routings. This cost function, however, does not account for contention on links. More sophisticated cost functions could be developed and implemented, but cost functions of sufficient complexity to effectly model routing and placement costs are generally expensive to compute. This could make an effective simulated annealing solution painfully slow for any moderately sized problem.

The idea of constructing the routing paths during placement works for algorithms other than simulated annealing as well. Even the most oblivious random placement may benefit by constructing paths during the placement. For every process placed, compute the best path available to all processes which have already been placed where communication occurs. This is a greedy solution and therefore will not be optimal, but it does have the advantage of being simple and efficient to compute. In algorithms, such as circuit placement [3], where all process clusters are placed simultaneously, computing routing paths during the process does not suffer from this problem because all messages are always given some routing (just as all clusters are always given some placement). Thus, new message routings can not be chosen in a greedy fashion but must compete for a good route with every other message.

4.4.2 Combining Contraction, Placement, and Routing

It is certainly possible to combine all three phases of the mapping problem by combining techniques described in the section on combining contraction and placement and the section on combining placement and routing. However, as noted above, such a solution may prove unpractical because it may be extremely expensive to calculate. Combining all phases of the mapping problem into one calculation will take time proportional to the product of the time to compute the individual phases, whereas independent computation of the phases takes time proportional to their sum. If the running times for each of the phases take $O(x)$, then running time of the combined solution can be expected to be $O(x^3)$ whereas running the phases in sequence is expected to be $O(3x)$. Also recall that the mapping problem is known to be NP-complete and so x by itself may not be polynomial in the size of the input graph. These considerations make a fully integrated solution appear impractical for all but a restricted set of mapping problems where canned solutions have been discovered.

4.5 Multiplexing

In some descriptions of the mapping problem there is a fourth phase [1]. When multiple processes are assigned to a single physical processor, multiplexing is required to schedule the execution of each process on the processor. This is generally done in a round-robin fashion where each process executes for some (perhaps weighted) number of cycles. It is certainly possible for a compiler to determine the multiplexing of the processes on each processor and some benefits might perhaps be gained by this approach. The approach suggested, however, is to defer multiplexing to the run-time system where each processor will be responsible for scheduling its own processes. This approach proves quite effective on uni-processor systems and is usually the assumed approach taken by today's multi-processor systems.

The main reason that multiplexing is left to the run-time system is that it has generally been the case that a processor's local performance was much faster than the network to which it was connected. When this is true, the processor's high speed allows the network

to view it as if all the processes are executing almost simultaneously. As network speeds increase, which is one of the goals of NuMesh, the benefit of computing the multiplexing of processes during compilation as part of the mapping problem will increase. This is an area that does not appear to be well researched to date.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In considering a software development environment for a parallel computer such as the NuMesh, the programming model, programming language, and the compiling system must be examined. The programming model should effectively describe parallel computations in the abstract. The description of the computation should be natural and intuitive, should capture the parallelism, and should model the computation's spatial and temporal requirements. Parallel programming languages are naturally based upon parallel programming models. These languages should be high-level and must be able to concisely provide a description of the model they are based on while also providing information to the compiling system so that the machine's processors and routing system can be programmed at a low-level. The compiling system for a parallel machine must perform the normal compiler tasks of parsing and code generation but must also deal with the problem of mapping the application onto the machine. The compiling system needs to provide a solution to the mapping problem whereby the processes are embedded on processors and message traffic is routed and scheduled onto the interconnection network.

A thorough understanding of parallel programming models is important to those wishing to develop parallel algorithms and applications. Even in the abstract, a parallel solution may be far from obvious; however, well-defined ways of modeling parallel computations provide the programmer with the tools for success in this endeavor. Programming models

are also necessary for comparative performance analysis of algorithms and provide a solid mathematical basis for showing the correctness of parallel algorithms. For those not involved in the development and analysis of new parallel algorithms, parallel programming models are still important because they form the basis of many parallel programming languages. So an understanding of these models is necessary for successful implementation of parallel programs in these languages.

The idea of augmenting LaRCS for use as an implementation language is believed to be a powerful one. The extended version of this idea suggests a new approach to implementing parallel algorithms that has the potential to make this task much simpler for the programmer. Programs could be implemented in a language which closely follows the abstract programming model in which the algorithm was likely conceived, whereas previous programming approaches force the programmer to map that model into some other form at the time of implementation. This new approach should prove useful in implementing many common parallel algorithms, such as those presented in [6]. This programming paradigm also provides the compiler with sufficient information to perform detailed network traffic analysis and potentially come up with an optimal mapping. The compiler should then be able to generate low-level code for the NuMesh to take full advantage of its high-bandwidth, low-latency communications substrate. If programming the NuMesh in this manner proves viable, this would push the NuMesh development environment from its current state to that of a state of the art system.

The examination of the mapping problem should provide sufficient background for those wishing to understand the problem of mapping an abstract parallel program to a real-world parallel machine. The efforts of manually programming each processor in a multicomputer, while heroic, are tedious and error-prone. A system for automatically mapping parallel programs to specific machines, based entirely upon a high-level description of the program, is the only way to affordably implement mid- to large-scale applications on parallel computers. Without such a system, the true power of parallelism will never be realized.

5.2 Future Work

Much of the tone of this thesis was prescriptive, rather than descriptive, in nature. Comparisons and evaluations of models and languages were presented in a purely qualitative manner. These were compromises that had to be made to cover this level of material in the time during which this thesis was developed. As such, much in the way of future work can be suggested.

The work done by Minsky [12], Lo [7, 8], and Berman [1] form excellent starting areas for detailed quantitative analysis of various contraction, placement, and routing algorithms. As parallel computing becomes more prevalent and static routing becomes a more accepted methodology, effective and efficient solutions to the mapping problem will be demanded. Detailed analysis of how different contractions algorithms compare with each other and of how they affect later stages of the overall mapping is needed. The same can be said for placement and routing algorithms. Performance analysis of these algorithms is also necessary to determine when multiple stages of the mapping problem can affordably be combined and when such combinations become prohibitively time-consuming. Results also might show that little or no benefit is gained through this combination of mapping phases. This is an area where little quantitative work appears to have been done to date. The information gained from this analysis could be used by the next generation of off-line parallel language compilers to choose specific algorithms for mapping a problem based on some prior (presumably simpler) analysis of the input program.

Another important direction that future research should take is to augment existing task communication languages, such as LaRCS, to include source code for the processes. As mentioned above, this approach is seen as a useful, practical, and powerful approach to the implementation of parallel programs. A language and compiler supporting this programming model are vital to determine if the enthusiasm for this approach is warranted. In augmenting LaRCS, for example, the researcher(s) need not start building the language from scratch, but can build upon the compiler and intermediate forms that have already been developed. Given this head-start, the project of implementing and evaluating the new language should be reasonably managable, although there are clearly many details

of this language which deserve more attention than they are given here.

In conjunction with the new CFSM that is being developed for the NuMesh, a generic, intermediate- to low-level router assember language should be developed. This multi-threaded CFSM will be capable of implementing complex routing behavior. A multi-threaded assembly-style language should be designed to hide the low-level architecture details of the CFSM from the programmer. It should be the case that this language is not tied to any specific details of the new CFSM and that it can be used to provide formal specification for any abstract routing operation.

5.3 Evolution of this Thesis

This thesis was originally intended to present a language and compiler for the NuMesh. The goals at that time were to provide a framework for a complete programming environment for the NuMesh. While the language and compiler were not implemented, the goals in large part were still achieved.

The research for the design of the intended programming language quickly showed that the development of such a language was well beyond the scope of what could be reasonable accomplished during the time this thesis was developed. Fortunately, however, research also uncovered the LaRCS language for the OREGAMI system [8, 7] developed at the University of Oregon. This language has many of the features desired for the NuMesh language. With reasonable modifications and enhancements, LaRCS should be suitable as a program development language for the NuMesh.

The original proposal also hoped this thesis might provide low-level software for the new CFSM. Two factors prevented the realization of this goal. First, the development of the new CFSM proceeded at a slower pace than was expected. At the time this conclusion was written, the new CFSM had been specified to a relatively low level of detail [15], but no work had yet been completed to simulate the design or evaluate its feasibility. Second, during the development of this thesis, the focus was shifting decidedly toward a high-level, abstract view of parallel computation on statically routed machines. This refinement of goals toward more abstraction at higher levels made assembler software for the new

CFSM an almost unrelated topic and thus it was not pursued in favor of more discussion of the former topics.

Ultimately, many of the assumptions in the original proposal were overly simplified and the goals often unrealistic. Work done during the development of this thesis helped to redefine the goals of the research. The new goals became to investigate parallel programming models and evaluate them with respect to the mapping problem which was itself examined in detail. These new goals were achieved.

Bibliography

[1] F. Berman and B. Stramm. Mapping function-parallel programs with the Prep-P automatic mapping preprocessor. Technical report, University of California, Dept. of Computer Science and Engineering, San Diego, CA, January 1994. Submitted to the *International Journal of Parallel Processing*.

[2] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press and McGraw-Hill Book Company, Cambridge, MA, 1990.

[3] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[4] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 200(4598):671–680, May 1983.

[5] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[6] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1992.

[7] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M.A. Mohamed, and J. Telle. OREGAMI: Tools for mapping parallel algorithms to parallel architectures. *International Journal of Parallel Programming*, 20(3), June 1991.

[8] V. M. Lo, S. Rajopadhye, M. A. Mohamed, S. Gupta, B. Nitzberg, J. Telle, and X. X. Zhong. LaRCS: A language for describing parallel computations for the purpose of

mapping. Technical Report CIS-TR-90-16, University of Oregon, Dept. of Computer Science and Information, Eugene, OR, 1990. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[9] Virginia M. Lo. Temporal communication graphs: Lamport's process-time graphs augmented for the purpose of mapping and scheduling. *Journal of Parallel and Distributed Computing*, 16(4), December 1992.

[10] Chris Metcalf. The NuMesh Simulator, nsim v3. NuMesh Systems Memo 24, January 1994.

[11] Chris Metcalf. Writing NuMesh code. NuMesh Systems Memo 16, January 1994.

[12] Milan Singh Minsky. Scheduled routing for the Numesh. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, Cambridge, MA, September 1993.

[13] John Nguyen. A C interface for NuMesh. NuMesh Systems Memo 3, February 1991.

[14] John Nguyen. CFSM assembler description. NuMesh Systems Memo 2, January 1991.

[15] David Shoemaker. NuMesh CFSM Rev 3: working draft. NuMesh internal document, March 1994.

[16] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[17] Sean E. Trowbridge. A programming environment for the NuMesh computer. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, Cambridge, MA, May 1990.

[18] Steve Ward, K. Abdalla, R. Dujari, M. Fetterman, F. Honoré, R. Jenez, P. Laffont, K. Mackenzie, C. Metcalf, M. Minsky, J. Nguyen, J. Pezaris, G. Pratt, and R. Tessier. The NuMesh: A modular, scalable communications substrate. Technical report, MIT Laboratory for Computer Science, Cambridge, MA, December 1992. To appear in *Proceedings of the International Conference on Supercomputing 1993*, Tokyo Japan.