Scheduled Routing for the Numesh

by

Milan Singh Minsky

B.S., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1986

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1993

Author . . . .
Department of Electrical Engineering and Computer Science
August 6, 1993

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

Accepted by . . . .
Fredric R. Morgenthaler
Chairman, Committee on Graduate Students

Scheduled Routing for the Numesh
by
Milan Singh Minsky

Submitted to the Department of Electrical Engineering and Computer Science
on August 6, 1993, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

A framework for the expression and analysis of statically defined communication patterns present in computations targeted for execution in the NuMesh multicomputer environment is developed. A system which solves optimization problems to statically allocate network resources for network traffic with real time constraints has been implemented. Results from the system, which uses simulated annealing and linear programming techniques to automate the process of placing and routing this traffic are presented.

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

Acknowledgements

To Professor Steve Ward:

During one's stay at university
(In this case one called M.I.T.)
There is with no great trouble found
Professors professing knowledge sound.
Seen with a little less likelihood
Are those who teach more than equations would.

For in Education's ivied walls,
Its fabled classrooms, hallowed halls,
One's not quite as sure to find
An Educator's kind of mind.

With great good fortune I found such
And worked with him to my good luck.
This tome and my education owes
A debt of guidance to one who knows
The gentle art of teaching minds
With knowledge, kindness,
Enthusiasm, humor, (and beer sometimes).

For my parents:

Parents come and parents go
(Well no, this isn't really so)
But parents like the ones I've got
Can't be sold and can't be bought
For they taught us quite carefully
The following philosophy:
Life improves when one can find
A way to open up one's mind
More than this we also learned:
While education's often earned
With sweat and candles two ends burned
Love is one commodity
That should be given constantly
And priced quite fairly when it's free

So while parents come and parents go
(Again this is not true, you know)
These two are top of the parent line
And this rhymster's thankful
That they're mine.

To Henry:

To the other half
Of this life.
My husband, I,
His lucky wife.
There isn't room to list it all
But since this thesis was a small
Electric surge within my brain
Right through to this completion date
Your support has never waned,
Your love and laughter's kept me sane,
And it's finally finished, dear! (Though somewhat late)

To Andy Ayers:

Where heaven meets earth there's a sign that reads:
Wherever you go whatever your deeds
However life for you proceeds
If luck smiles a little you might just find
That friend in a million - one of a kind

Working with that friend, if done
Will be less like work and more like fun
And talking to that friend as well
(About 6.004 or Compilers of L)
Will improve the aspect of your days
And when you move to other phases
Of your lives, whatever the cost,
Make sure this friendship's never lost

For Marvin and Gloria:

If there's one conception has a flaw
It's that of the Wicked Parent-In-Law.
This old legend has me troubled,
For when I was married, my parents doubled.
The love and care flowing from these two
The soft good humor and strong support,
I can't describe these things to you
And also keep this poem short.
But if I continued on and on,
Listing golden qualities,

You might stop reading 'fore too long,
So I'll finish up and hope you've seized
The value which they hold for me.

To John Nguyen:

An officemate who shares your space
But manages with great good grace
To offer help when its most required
To provide diversion when you are tired
Of working. And who with quiet dignity
Hacks hacks of legendary quality
Such an officemate is hard to find
But such was the one who once was mine

To Sharon:

I'm certain that the thesis
Which follows these first lines
Would still be all in pieces
If not for Sharon's kind
And gentle-strong persuasion
To finish what you start,
And to follow your imagination,
Which she gives from her heart.

To Asim:

Thanks to my brother, Asim Singh
Who makes me laugh about everything.
Who's made me laugh since we were small
Who's never let me take a fall
Without being there with his helping hand
And making sure I still could stand.

To Margaret, Julie and Meher:

Through no fault of my parents
But because of the chromosome X
I never had any siblings
Who shared my particular sex
Now I do
And I'm thankful too
For their brilliant example
Which proves to me just one thing
If a man can do what a man can do
Then a woman can do everything

Thank you to Anne for putting up with my wanderings into her office over the years to ask for so many things which she so cheerfully provided.

This project benefitted a great deal from interesting discussions with Kathy Knobe, Mike Noakes, Glenn Adams, Frank Honore, Norman Margolus, Nate Osgood, John Pezaris, Ian Eislick, David Shoemaker, Russ Tessier, Gill Pratt, and Anant Agarwal. Very valuable too, was the general environment of creative curiosity encouraged at M.I.T.

A special thanks to Tom Knight who is a constant source of clearly described good ideas, and to Clark Thomburson who took the time to help me understand linear programming basics and to find lp solve.

Thanks to Professor D. Troxel, a patient and helpful course advisor.

Thanks very much to Chris Metcalf for a careful reading of initial drafts of this thesis and for many constructive comments regarding the material.

Thanks to good friends: Melissa Cohen, and Michael Fredkin who never let me go too long without having some fun, Yen-Kuei Chuang for always taking time to send warm and interesting email from the land of sunshine, busy as she was, Carol for taking wonderful loving care of the dogs during these busy last few weeks, Julie Fouquet for teaching me how to organize the time devoted to a thesis. I only wish I'd been a better student.

This thesis is dedicated to my aunt, Nirmila Beniwal.

Contents

List of Figures

Chapter 1

Introduction to the Problem

1.1 The NuMesh Project: An Approach to Multicomputer Communication Substrate Design

Research in multicomputer design has resulted in a variety of approaches to the problem of communicating information between the various elements of a distributed machine architecture. Interconnection networks in particular form a critical component of any parallel computer and have received much attention from the architects of these machines. Several popular concurrent architectures including shared memory machines such as the M.I.T. Alewife and Stanford Dash architectures as well as distributed memory model based systems such as the M.I.T. J Machine and Caltech Cosmic Cube, [ACD+91, D+89, DS87, G+91, Sei85] have opted for a general purpose network design which is optimized for a dynamic approach to handling network traffic.

Thus *dynamic routing*, in which arbitration of contention for network resources is relegated to the run time system, has been incorporated at a very low level into the design of many of today's multicomputer networks. While this approach results in general purpose network architectures, there is a potential loss in network performance for applications with highly predictable communication needs.

An alternative approach, systolic architectures [Kun82] uses compile-time approaches to communication scheduling that offer potential cost/performance advantages in applications where communication patterns are largely predictable.

An important goal of the NuMesh project is to study the degree to which dynamic routing is a necessary component of parallel computer network design. This research goal is being pursued at several different levels.

At the hardware level several different network node designs [Hon92, Pez92] are currently being explored in order to decide what hardware mechanisms will best support the mix of communication requirements characteristic of applications typically executed on parallel machines. At the software (compiler) level, techniques are being developed to identify and express the mix of communication requirements which characterize a particular application. These requirements tend to form a spectrum: purely static communication, for which the point in time when an application transmits a message is known to within some upper bound, and the size, source, and destination of the message are each known at compile time lies at one end of the spectrum. Completely dynamic communication for which nothing about the message traffic can be predicted at compile time, defines the opposite end. In many cases only partial information about the parameters describing application communication requirements may be available statically. The communication patterns generated by a particular computation executing on a parallel machine may occupy one or more points or ranges in this 'communication characterization spectrum'. A subgoal of this larger effort to characterize application communication requirements is the use of this type of characterization to optimize the allocation of network resources. This subgoal is the subject of the thesis work described here.

1.1.1 Motivation for Static Routing

One advantage of precompiled communication patterns is hardware simplicity. Runtime routing decisions can be eliminated altogether or handled infrequently with little dedicated routing hardware. More significant is a potential for reduced contention for network resources because of the predictability of such traffic. In particular, applications with hard real time constraints are excellent candidates for static allocation of network resources. Detecting and avoiding network congestion at compile time for applications of this type can ensure that a particular desired throughput is achieved.

1.2 This Thesis

The work done for this thesis ties compiler analysis of communication requirements for a limited set of applications to (the low level) hardware design of the NuMesh network, supplying offline placement, routing and scheduling modules to allocate network resources for computations with static communication needs. Applications which are periodic and have real time constraints are targeted in particular.

Methods from an approach developed by researchers at the North Carolina State University [SA91], called *scheduled routing* have been adapted and modified here in a first attempt to provide network resource allocation software for the NuMesh environment.

This thesis is divided into 6 chapters. Chapter 2 presents some background on network design and discusses some of the motivation behind implementing a scheduled router for the NuMesh. Chapter 3 describes the current prototype NuMesh network architecture. Chapter 4 contains an overview of the scheduling router as well as a detailed description of each component of the system, contrasting different approaches to the problems solved by each subsystem. Chapter 5 presents results of applying the scheduled router to various applications. Finally, chapter 6 discusses limitations and explores possible extensions to the system.

Chapter 2

Background

2.1 Communication in Multiprocessors

2.1.1 Dynamic Versus Static Routing

In many modern multicomputer networks, the routing of network traffic is handled dynamically. Communication processing hardware, resident at each network node, inspects a portion of an incoming message deriving control information which is then used to route the associated data portion of the message. Thus network nodes in these architectures are often designed as general purpose interpreters of encoded communication control information. A clear benefit of static routing techniques (when they can be applied) is the elimination of network cycles devoted to decoding routing information encoded in network traffic. [1]

A second advantage of static routing is compile time elimination of network contention. Messages propagating through any but the most powerful (all to all) interconnection topologies must share network resources. Building fully connected networks, in which each node is directly connected to all others, become prohibitively expensive beyond a very small number of nodes. Thus most multicomputer networks are based on various approximations to this topology.

Contention for network resources is dictated primarily by network topology, available network resources and network traffic patterns. The degree to which these parameters influence contention depends in turn on a host of design and implementation issues. The physical network design is often based on cost, packaging, and availability of hardware, as well as the requirements of the processor and memory configuration for which the network is targeted. Traffic patterns depend on the network, on the kinds of applications which actually run on the machine and the effectiveness of task allocation, placement, load balancing, and routing algorithms which distribute the work load amongst the resources available in the system. The following subsections discuss typical strategies used to map an application to a parallel environment and describe some results from theory and simulations developed by researchers to model how contention actually affects network performance.

[1] However, when messages are long and routing information in a header can be shared by multiple bytes of data, performance loss due to dynamically routing data is minimized

2.2 Mapping an Application to a MultiComputer

Communication (and thus contention) patterns which arise in an interconnection network when an application executes on a parallel machine are the direct result of several stages of processing which transform source code written by the programmer to executable code resident on the machine's processing elements. These stages and their influence on network traffic are outlined below.

2.2.1 Partitioning

Partioning divides a program into chunks of computation (*tasks*) suitable for execution on a parallel machine. Partitioning algorithms attempt to balance parallelism with the amount of data which must be exchanged through the network. Data partitioning distributes data across processors executing a single program. Distributing an algorithm across multiple processors which execute code in a pipelined fashion is an alternative approach to the problem. This second strategy is more common in real time processing applications which require high throughput, as the data distribution step required in data partitioning can degrade system performance. For applications where computation can proceed independently on easily seperable data sets data partioning is preferred. In general, partitioning determines **how much** message traffic travels through the interconnection network. Paritioning is not addressed in this thesis. The assumption is that a higher level compiler or the programmer has already specified a set of tasks which will execute in the NuMesh environment as well as the communication edges present between the tasks.

2.2.2 Placement

Placement or *task allocation* maps tasks created by partioning to the physical nodes of the multicomputer. Because tasks generate message traffic, task placement affects the spatial and temporal distribution of message traffic in the network. Issues such as communication locality are considered and exploited in typical placement strategies; if two tasks communicate repeatedly with each other they are placed in close proximity. In systems which support dynamic placement, changing communication and processing requirements can result in task migration from node to node at run time [Fox88]. Static placement however tries to map a group of tasks to processor nodes based on information about an application's communication requirements specified at compile time. This information may be a single collapsed graph expressing all of the communication which must take place during the application's execution, or it may be several separate graphs each of which describes communication patterns in separate phases of the application's execution. In general, placement influences **where and when** message traffic is active in the interconnection network.

2.2.3 Routing in Networks

Rings, meshes, hypercubes, k-ary n cube, busses, multistage switch networks and crossbars have all been proposed and implemented as interconnection network topologies for various parallel architectures. In all of these non fully connected topologies, contention for network resources contributes to the latency and throughput available for network traffic. A network's *routing technique* determines how contention is handled when it actually occours. Typically, different routing techniques perform better or worse depending on the amount of loading present in the network.

Dynamic routing techniques include *store and forward routing* which requires every node to store an entire message in node memory before the message can be transferred to the next node in its path, *cut through* routing which still requires buffering at intermediate nodes when messages collide but which allows messages to traverse several hops if the links involved are not busy, and *wormhole routing* [DS87] which is essentially a blocking version of cut through routing. The head of a message advances along its path as long as the required links are available. If the message encounters a link which is occupied, the message **blocks**. All previous network links being used by the message remain busy and unavailable to other messages until the link which blocked the head of the message is released.

While *oblivious* routing determines a single possible route based on a message's source and destination, *adaptive* routing strategies exploit multiple available paths from source to destination. Simulation of adaptive routing techniques [Nga89] have shown that the point at which a network saturates under high load conditions shifts by a factor of two when adaptive rather than oblivious routing is used. However the complexity of switching decisions made by routers increases when adaptive routing techniques are used. Thus some simple greedy strategies which do not require too much routing logic complexity have been adopted. Always routing incoming packets to an available idle channel if the idle channel is one of the profitable channels for the packet, is one suggested approach. Since this technique uses only local information to decide which way to route a message, it is possible for it to make suboptimal decisions. Offline routing can use global information to make routing decisions at compile time. Adaptive routing is also susceptible to *deadlock*.

Deadlock Avoidance

Deadlock [DS87] is a condition in which traffic can not advance in the network because of a circularity in network resource usage. Routing techniques which are susceptible to deadlock must reliably avoid it. In dynamically routed networks which use wormhole routing, deadlock is typically avoided by assigning paths to all messages in which the use of network resources always proceeeds in a fixed order. Thus *dimensional routing* routes a message completely in a given dimension before moving on to the next. Since all messages are routed this way: first in `x`, then in y, etc., deadlock is not possible. Message `MA` can not be waiting for resources currently occupied by another message, `MB` which is in turn waiting for resources occupied by message `MA`. A method to avoid deadlock when the existence of communication operations but not their execution times are predictable at compile time is outlined in [Kun88]. Developed for a systolic communication environment, this technique defines a 'crossing off' procedure which analyzes a given communication program to determine if it is deadlock free. It avoids overwhelming network resources by only allowing users to write communication programs consisting of *executable pairs* of read and write operations, where every read operation matches a write operation in the crossing off procedure.

Static routing avoids deadlock completely. If messages have known arrival times and orders, flow control evaporates, resource usage is determined at compile time, and deadlock is not possible.

2.3 Performance of Multiprocessor Interconnection Networks

In an effort to provide a metric by which interconnection network designs can be evaluated and compared, researchers have developed analytical techniques to measure the performance of communication networks in parallel computers. Modelling of network contention in direct and indirect buffered networks has produced closed form expressions which describe the latency and throughput of these networks. This analytical work allows designers to evaluate a network architecture by choosing the appropriate parameters and examining the resulting values for network performance under different conditions.

Some of the factors incorporated in these performance models include parameters which characterize the physical network, such as bisection width, network dimension, and topology. Considered as well are parameters describing the nature of network traffic such as packet size, request rate, and degree of communication locality. Because successful application of static routing techniques can eliminate the contention component of expressions for network latency and throughput, it is important to understand the root of this contention in various network models.

A Model For Direct Buffered Networks

Models of traffic congestion in a data network are often developed in terms of the statistics of the arrival of packets at the network queues. In a model developed by[Aga91], the author derives expressions for network performance in the presence of contention for a direct buffered network using oblivious wormhole routing. By making a few simplifying assumptions about the nature of message traffic in the network, and invoking queuing analysis to compute the average number of cycles a packet or message waits before being routed, a simple expression for network latency is developed:

$$T_c = nk_d + B + \frac{\rho B(n+1)}{1-\rho}$$

In the above equation, T_c is the average transit time through the network taking into account contention. ρ, the channel or link utilization is given by

$$\rho = mk_d$$

Tc, the latency increases rapidly as ρ, the link utilization, approaches one. In the above expression, m is the probability that a node will generate a message, B is a factor reflecting the packet size of the message, k_d is the average number of hops travelled by a message in a given dimension. [2] Assuming n dimensions, nk_d is the average total distance traversed. Here, the assumption of a dimensional routing protocol results in a probability that goes as $1/k_d$ that a message will switch dimensions as it travels through the network.

[2] This expression results from simplifying a previous expression with the assumption that k_d is $>> 1$

Wormhole routing results in second order effects. If a message is blocked from advancing it remains in the network blocking resources along its length from the node with the blocked resource all the way back to the source node and thus prevents other traffic from moving. These second order effects are hard to model; the above expression for latency has been compared to simulation results under varying conditions and has proven surprisingly accurate.

A Model For Indirect Buffered Networks

In [Kru83], a model applicable to buffered Banyan networks results in an equation for average network transit time:

$$T = \log_k N t_k$$

Where t_k, the average transit time of a packet through a kXk switch, is given by:

$$t_k = t_r + t_c \frac{(1 - \frac{1}{k})p}{2(1 - p)}$$

Here $\frac{(1-\frac{1}{k})p}{2(1-p)}$ is the average number of queueing cycles suffered by a packet. p is the probabilty that a packet arrives at each input of a kXk switch. The component of transit time due to network loading for this model is also derived using queuing theory. Again, t_k rises rapidly with increasing network load.

2.4 What Kinds Of Applications Can Benefit From Static Routing?

Programs with highly predictable communication requirements are often found in circuit simulators, CAD development software [3], and in speech and video processing code. Repetitive execution of simple computations, as well as the real-time constraints present in the latter types of applications, combine to provide problems to which static routing methods can be applied and the technique evaluated. Circuit simulation, though usually without real-time constraints, often proceeds by repeatedly invoking a statically defined circuit on changing input sets. Data flows from input to output in the circuit in a predictable fashion with only the results of the simulation changing from cycle to cycle.

Figure 2-1 shows a potential candidate for static routing. Simulation of a pipelined circuit with each component or gate assigned to an individual NuMesh node proceeds with inputs arriving at the input task nodes in waves. Communication operations which will take place between the different components is known at compile time and does not change during this particular phase of the simulation.

As an example real-time application, Figure 2-2 shows a task level description of code written for the NuMesh as part of an implementation of a real time speech recognition system. [Duj92]. The execution of the subtasks shown proceeds in a pipelined fashion,

[3] There is an interesting recursive aspect to targeting CAD as a candidate for static routing: The techniques which are used to place and route components and wires in CAD systems mirror the techniques used here to place tasks and route communication

Figure 2-1: Circuit Simulation as a Candidate for Static Routing.

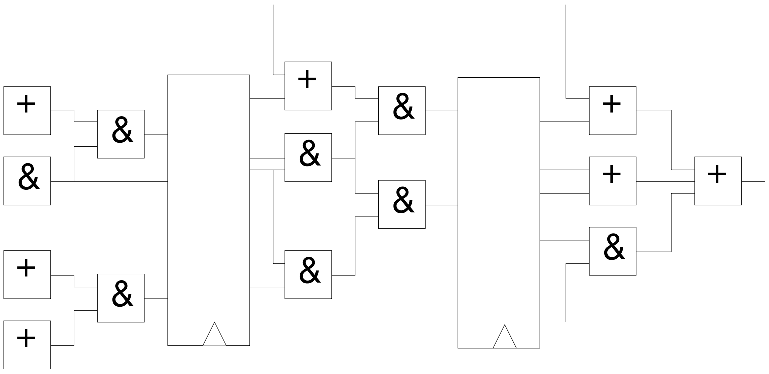
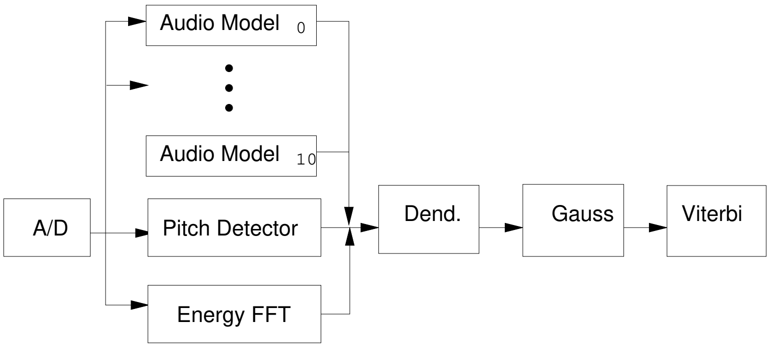


Figure 2-2: Front End of Speech Recognition System Implemented for the NuMesh.



with each subtask assigned to a seperate NuMesh processor/communication finite state machine pair. Task placement and communication routing and scheduling was optimized manually for this implementation in the absence of available tools.

For periodic real time applications in particular, traditional networks which support completely dynamic routing protocols often do not provide the required network performance. Especially if they are executed in a pipelined fashion, these applications are susceptible to throughput loss if the network's routing protocol handles data transfers without regard to the hard real time constraints governing the generation and arrival times of messages communicated through the network. In [SA91] the authors argue that cut-through and wormhole routing techniques, while offering minimum latency for traffic under light load conditions, do not meet real time constraints when the network loading is high. Static network resource allocation policies which explicitly incorporate the throughput requirements of a particular application can identify potential congestion 'hotspots' at compile time and reroute or reschedule communication operations if needed. This can lead to improved network throughput. A condition that the authors calls *output inconsis-*

tency can result when periodic execution of a pipelined *task flow graph* [4] relies on worm hole routing to deliver network message traffic.

Latency as well as throughput is an important performance measurement. If a task emits a long message repeatedly at a predictable frequency and the message blocks at a resource because of contention, the latency of the computation waiting for the arrival of this message may needlessly increase, if the receiving computation is tightly coupled to the sending task.

When network traffic is statically predictable, the processor-network interface benefits as well. Since message arrival order at a processor node is known at compile time there is little overhead associated with transferring a message from the network to the processor.

2.5 Motivation for Scheduled Routing Backend

Given the improved bandwidth available in a network when static routing techniques can be applied, it seems worthwhile to develop techniques to statically place tasks and route and schedule communication for those applications which can take full advantage of this bandwidth. This thesis implements a set of algorithms for this purpose. A simulated annealing algorithm places statically specified tasks. Statically declared messages are assigned routes through the network using an iterative path assignment algorithm. Scheduling of communication on network links solves a linear programming formulation of the system constraints to allocate switching schedules for the Communication Finite State Machines.

The framework for the system, tasks are first placed, routes are then assigned, and finally messages scheduled on network links, as well as the algorithms used to solve the scheduling problems use *scheduled routing* techniques described in [SA91]. These techniques were originally developed to solve the problem of handling high bandwidth communication for vision processing code with hard real time constraints. Since the NuMesh will be a platform for many applications of this kind, the framework developed here to place, route, and schedule communication will hopefully form an important component of the overall NuMesh programming environment. Scheduled routing allows real time constraints to act as inputs of the network resource allocation problems which should be solved in order to map an application with static communication needs to the NuMesh environment.

[4] The task flow graph representation of a periodic real time application partitions the application (using algorithm partitioning) into a set of tasks which executes in a pipelined fashion

Chapter 3

The NuMesh

3.1 Architecture of the NuMesh

Abstractly, a NuMesh consists of modules, or *nodes*, that may be connected together to populate a three-dimensional mesh. For example, Figure 3-1 shows a simplified view of a small mesh of current prototype nodes. This figure depicts each module as a unit whose peripheral connectors provide signal and power contacts to four immediate neighbors.

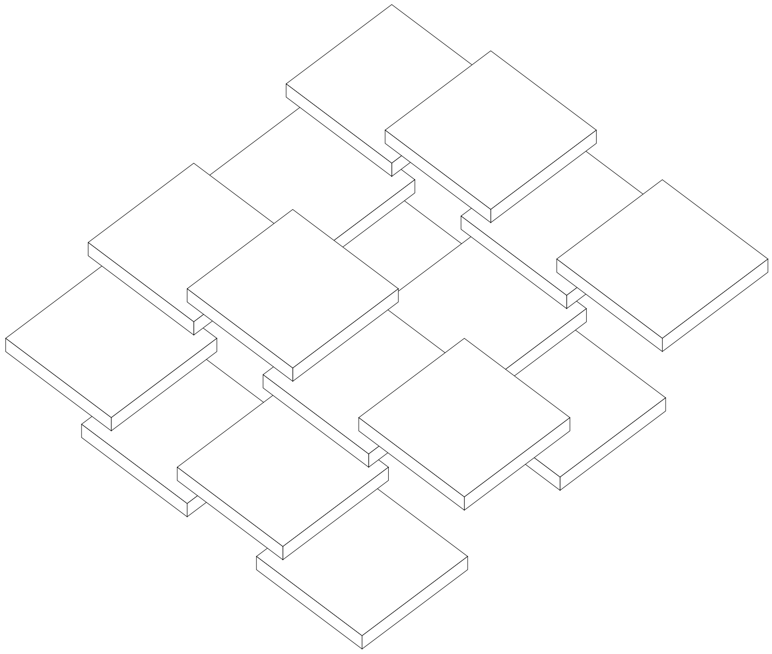
Each node in the mesh constitutes a digital subsystem that communicates directly with each of its neighbors through dedicated signal lines. During each period of the globally-synchronous clock, one datum may be transferred between each pair of adjacent modules. Currently, our prototype runs at just over 1.2 Gbits/second per port; next-generation modules will be clocked faster.

3.1.1 NuMesh modules

The approach in the design of the NuMesh communication substrate involves standardizing the mechanical, electrical, and logical interconnect among modules that are arranged in a three-dimensional mesh whose lowest-level communications follow largely precompiled systolic patterns. The attractiveness of this scheme derives from the separation of communication and processing components, and the standardization of the interface between them. By making the communications hardware as streamlined and minimal as possible, and requiring the compiler to do almost all the work for routing data within the mesh, it should be possible to maintain high-bandwidth, low-latency communications between the processing nodes distributed throughout the NuMesh.

An idealized NuMesh module is a roughly rectangular solid with edge dimension on the order of two inches. A node is logically partitioned into two parts: a *local processor* that implements the node's particular functionality, and a *communications finite state machine* (CFSM), replicated in each node, that controls low-level communications and interface functions. A node's local processor may consist of a CPU, I/O interface, memory system, or any of the other subsystems out of which traditionally-architected systems are constructed. A node's CFSM consists of a finite state machine, data paths for inter-node communication, and an *internal interface* to the local processor. A typical module is depicted schematically in Figure 3-2.

Figure 3-1: Configuration of NuMesh nodes in a diamond lattice.



3.1.2 CFSM Structure

The core of the CFSM control path is a programmable finite state machine. The transition table, held in RAM, is programmed to control all aspects of routing data to other nodes and interfacing with the local processor. A small amount of additional hardware reduces the required number of states by providing special-purpose functionality such as looping counters.

The CFSM data path consists of a number of ports connected through a switching network allowing data from one port to be routed to another. Most of the ports are for communication to other CFSMs; however, one port supports CFSM-local processor

Figure 3-2: Communciation Finite State Machine.

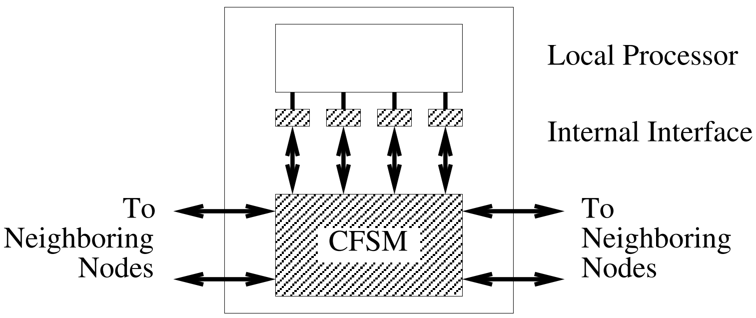
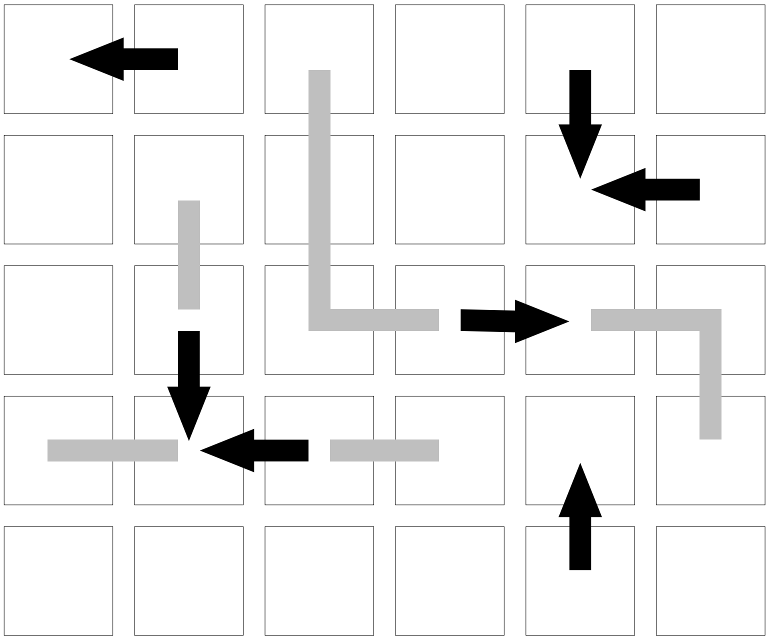


Figure 3-3: CFSM activity during a given clock cycle .



transfers and allows the CFSM to move data between the processor and the mesh. This port may be wider than the network ports, and provide out-of-band signals in addition to the ordinary data path. Optimally, any combination of ports may be read or written on each clock cycle, but this flexibility may be constrained in a given implementation.

Each CFSM also contains an oscillator to generate the node's clock (the local processor has the option of an independent clock), and circuitry to control the phase of the oscillator. The clocks can be kept globally synchronized by any of a number of methods [PN93]). The clock cycle time is constrained by the time necessary to transfer a data word between adjacent nodes, which can be made quite short because of the prescribed limited distances between nodes, the point-to-point nature of the links, and the use of synchronous communications.

3.1.3 CFSM Programming

The transition table of the CFSM is typically programmed to read inputs from various neighbors or the local processor into port registers and send outputs from various port registers to other neighbors or the local processor on each clock cycle. It may be thought of as a programmable pipelined switch.

The CFSMs in a mesh, operating synchronously at the frequency of the communication clock, follow a compiler-generated preprogrammed, systolic communication pattern. The aggregate CFSM circuitry constitutes a distributed switching network that is customized for each application; its programmability allows this customization to be highly optimized.

3.2 Comparison With Systolic Arrays

Arrays of synchronous processors connected together in fixed interconnection topologies [Kun82] offer a high peformance environment for certain kinds of applications. Code which can be partitioned into a tightly coupled set of computations exchanging data in fixed communication patterns fits very well into this model of distributed computing. Reconfigurable communication paths have not been a part of traditional systolic array designs. However the IWarp [PSW91] computer does allow 1D and 2D mesh connectivity between computing units. Software technology developed for this kind of environment may offer a good starting point for further development of compilation technology which predicts and schedules communication in the broader set of topologies and applications currently being studied in the NuMesh project. A scheduling compiler in particular [Lam89] as well as improvements to the lower level static routing methods implemented for this thesis will need to be implemented.

3.3 The NuMesh Programming Environment: Work to Date

Previous work on the NuMesh programming environment has centered on a static programming model. A high performance approach [Fet90] relies on a precise cycle count of the computation performed on each node's processor to efficiently schedule communication between the nodes. This work has resulted in the specification of an intermediate language to which programs written in (restricted subsets of) higher level programming languages can be compiled and executed on the Numesh.

Language and user interface issues associated with providing the programmer with an environment in which to develop programs whose dynamic behavior does not depend on run-time data have also been investigated in some detail [Tro89] Results include a pictorial stream-based front end implemented for the development of real time applications such as digital signal processing for speech and video applications.

A C-language interface for writing programs with flow controlled communication requirements has also been implemented [Ngu91]. The user of this system is able to write programs in a restricted subset of C and produce compiled processing element code as well as binary executables for CFSMs.

This thesis extends some of the language and user interface components required to express static communication. It also automates much of the placement and network resource allocation to message traffic which must be done in order to automatically generate Communication Finite State Machine Programs.

Chapter 4

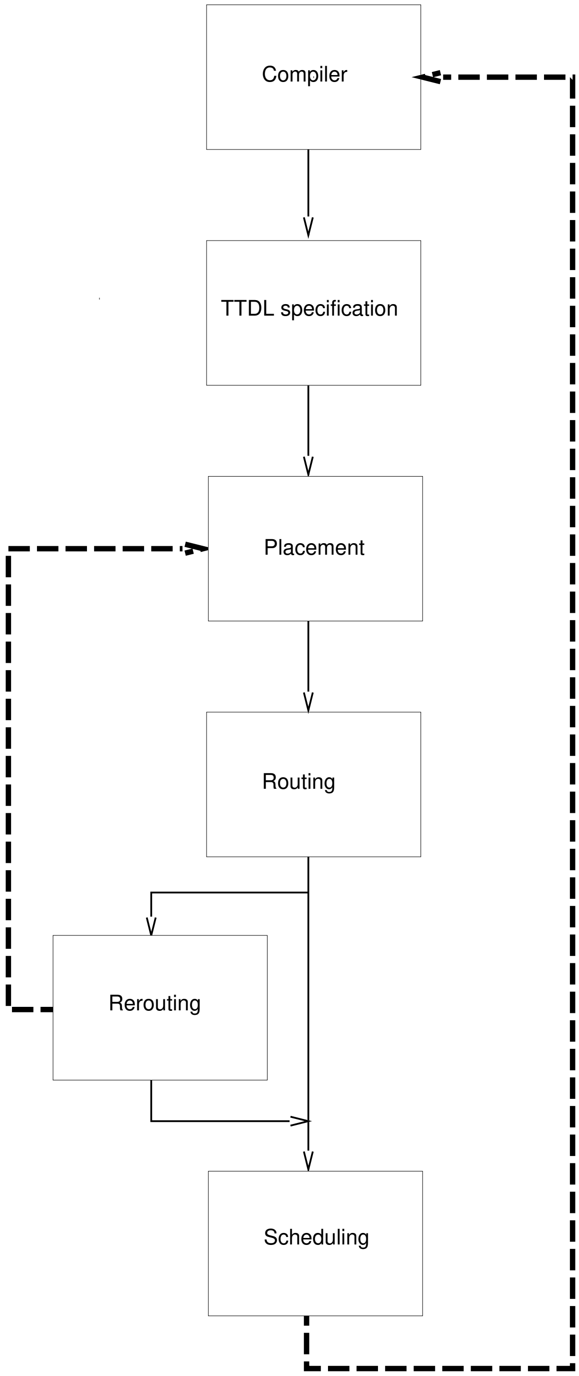
Implementation of the Backend.

Allocation of network resources to message traffic proceeds in several stages in the scheduled router backend. Figure 4-1 gives a block diagram description of the system. Eventually a source level scheduling compiler will extract communication pattern descriptions from application code and generate an intermediate form representation. This form will be the input for the placement, routing and message scheduling back end. Currently however, a Lisp like intermediate level language is supplied as the front end to the system. The programmer can express an application's task communication graph in this intermediate language. Execution of forms written in this language generates a representation of the message traffic which will be present in the network at run time. This traffic information is then passed to the placement, routing, and scheduling subsystems. A placement algorithm based on simulated annealing places all tasks specified in the intermediate language description onto physical network nodes. After placement , a path assignment phase distributes the message traffic in *space*, assigning routes through the network to each message. Finally, a linear programming based scheduling phase distributes the message traffic in *time*. In this phase, communication finite state machine switching schedules are arranged. Messages which conflict in space but which are allowed a degree of freedom in time, are assigned transmission intervals ensuring that messages are not simultaneously scheduled onto the same links.

4.1 Intermediate Form

For the purposes of this thesis, a somewhat restricted intermediate form representation language was designed and implemented in Common Lisp. The approach taken here is similar to, though less sophisticated than the work done to develop LaRCS [LRea90], a language to express regular communication structures. T(ask) T(raffic) D(escription) L(anguage) is a simple intermediate representation in which static communication patterns characteristic of a particular application can be expressed as functional definitions in a Lisp like syntax. It defines a language interface at a level intermediate to the source level compiler and the scheduled routing backend so that users of the system may express application communication patterns at a moderately high level in a friendly Lisp-like environment. The execution of special forms provided in the language ultimately results in

Figure 4-1: Structure of Backend Placement, Routing, and Scheduling System.



the generation of a communication graph with nodes which are placed, and edges which are routed and scheduled by the backend network resource allocation modules. Thus this description language enables the programmer or compiler of parallel algorithms [1] to specify information about the static and dynamic communication behavior of the computation being mapped onto the NuMesh.

TTDL's model of communication is based on *streams*. To communicate via the network a task opens a set of output and input streams [2] and sends and receives data to and from other tasks in the system. A regular expression describes the traffic which flows along a stream during a particular communication phase. The regular expression represents statically the traffic which will be present on a stream during a real time interval.

The TTDL intermediate form evaluation environment is written in Common Lisp [Ste90] The choice of Common Lisp as an intermediate form and interpretation environment has several advantages over using C. [3] Since parsing and lexical analysis are implemented as part of the primitive Common Lisp read facility, this analysis need not be reimplemented as part of the current system. The special forms provided in the language, which are used to describe communication patterns in TTDL, fit naturally into the paranthesis based prefix notation of a Lisp like language. Finally, an extensive macro definition facility which allows the definition of arbitrary functions that convert Lisp forms into different forms before evaluating them is provided in Common Lisp. This facility is exploited here to define evaluation functions for TTDL language special forms.

4.1.1 Implementation Details of TTDL

The structure of TTDL at the level implemented for this thesis is quite simple. Methods to establish definitions for:

1. A function which encapsulates a particular communication pattern
2. An iteration function which applies this function to a prescribed set of tasks (which in turn can be specified functionally) and
3. Mapping functions which capture known optimal mappings from logical to physical communication topologies

have been provided as part of the initial programming environment.

Communication patterns themselves are expressed in terms of special forms which allow iteration, sequential execution, a compute operation [4] and write and read operations to stream objects.

4.1.2 Basic Language Constructs

TTDL language forms play the role of *special forms* in Lisp. Each form is defined either as a macro or a function in the toplevel Lisp environment. Execution of programs written with

[1] if this or a similar intermediate form is eventually adopted by source level compiler(s) developed for the NuMesh

[2] where a stream is specified completely by a source task and a destination task

[3] As one example of an alternative language

[4] Which is a general escape mechanism to represent processor computation

these special forms builds a directed acyclic graph (or *DAG*) representing computation and communication operations.

This representation is then used to assign **release** and **deadline** times. Time in the current system is assumed to be expressed in terms of network clock cycles, and the processor and network are assumed to have an integer ratio [PW92]. The actual degree of synchronization between processor clocks and between the processor and network clock is assumed to be quite accurate [PN93]. Message objects created by the execution of the special forms `write-stream` and `read-stream` are assigned release and deadline times based on the execution environment's representation of time.

A short description defining the semantics of each primitive form and some simple programming examples are presented in the following section.

Primitive Forms

define-comm (macro) (define-comm <name> <args> <body>) Define-comm is a macro which establishes a function *name* with arguments *args* in the toplevel Lisp environment. The body of the function contains one or more Lisp forms but must end with a regular expression describing the communication pattern to be associated with <name>. Define-comm is the basic means of associating a communication pattern definition with a name in the Lisp environment.

sequence (macro) (sequence <statement0> <statement1> ....) Sequence allows concatenation of statements in a communication pattern description.

iterate (macro) (iterate <body> <n>) Iterate allows repeated execution of statements in a communication pattern description.

compute (function) (compute <t>) Compute is a general escape mechanism to represent computation. It advances time by <t> cycles.

read-stream (function) (read-stream <message> <stream>) Read-stream checks a queue of pending writes to match this read from <stream> with any previously encountered writes. If a matching write is not found, a message instance is created and placed in the pending read queue.

write-stream (function) (write-stream <message> <stream>) Write-stream checks a queue of pending reads to match this write from <stream> with any previously encountered reads. If a matching read is not found, a message instance is created and placed in the pending write queue.

broadcast (function) (broadcast <message> <stream0> <stream1> ... <streamn>) Broadcast iterates write-stream of <message> over the argument streams <stream0> <stream1> <stream2> .. Low level support for broadcast does not yet exist. The routing and message scheduling programs do not currently allocate routes and time intervals to messages broadcast to more than one destination.

synchronize (function) (synchronize <P>) Synchronize creates a seperation of communication into different *phases*, emitting an instruction in the communication finite state machine code stream to check for flow control before beginning the execution of a schedule. Synchronize is useful if the execution time of code distributed over multiple processors is not accurately known, but *relative* timing information is available. All nodes executing a schedule which begins with a synchronization point will switch data on cycles measured relative to this point.

A simple example program is shown in Figure 4-2. `m0` is a message type describing the size of the message transmitted or received. When the form `(reader t0 t1)` is invoked, the call to `read-stream` in the body of the function `reader` is encountered. Execution of `(read-stream m0 t-north)` with `t-north` bound to `t0` results in the creation of a subtask object. The `read-stream` code searches for a pending write from `t0`. Since the read is evaluated before the write, no message is found and a message from t1 to t0 is enqueued on the pending read queue, marking it as read and the <u>destination</u> task field of the message is filled in with the id of the subtask object. Evaluation of the compute form assigns a computation time to the subtask created by `read-stream` When the `write-stream` in `reader` is evaluated, a pending read is searched for. None will be found and a message with a <u>source</u> field filled in with the subtask id is enqueued on the pending write queue. When an invokation of `writer`, `(writer t1 t0)`, is evaluated, a match is found for both the `write-stream` and `read-stream`. This entire operation is iterated n times, creating the communication graph which is then passed to the backend.

Figure 4-2: A Simple Example of TTDL.

```
(define-comm reader
  (source-task t-north n)
  (iterate
     (read-stream m0 t-north)
     (compute 100)
     (write-stream m0 t-north)
   n))
(define-comm writer
  (source-task t-south n)
  (iterate
     (write-stream m0 t-south)
     (read-stream m0 t-south)
   n))
(reader t0 t1)
(writer t1 t0)
```

Evaluation of TTDL Forms and Generation of Task Flow Graphs

A task flow graph is generated when a program description is evaluated in the task description language. The task flow graph is analyzed to compute the message release and deadline times. An auxiliary task flow graph is used to do the detailed analysis of task dependencies. For each message which is sent or received by a task in the task description, a *subtask* is created in the auxiliary task flow graph. A critical path algorithm is run for each subtask in the graph. Each subtask in the task flow graph sends and receives at most two messages; one message to the next node in the subtask graph (an *intra-task message*) and one message to or from a subtask which belongs to another task (an *intertask message*). Intra-task messages are really an artificial construct to account for the serial order in which messages have already been ordered by the compiler; the intra-task message adds the constraint that a subtask cannot begin execution before the completion time of the preceding block of code in the task's execution sequence.

The critical path algorithm must find, for each subtask, the earliest time at which that task could possibly be activated, based on the latest arrival time of the (at most two) received messages. This is called the subtask *delay time*. The subtask delay time is added to its computation time to produce the *release time* of its outgoing messages. For each outgoing message, the *deadline time* of that message is set to be delay time of the destination of that message. In Figure 4-4, the release time of the outgoing message from subtask **b** has a release time of 10, and a deadline time of 50, which is the delay time of the destination task, **e**. The dotted line shows the critical path through the graph, which determines the total latency task. The difference between the release and deadline times of a message, and the message length, determine the amount of slack which the scheduler/router has to shift the message in time, without adding to the latency of the task flow graph. In a pipelined application, there will be extra permissble slack between tasks based on the clock rate of the pipeline.

The delay times of subtasks, and hence the release and deadline times of messages, are computed recursively using the algorithm shown in Figure 4-3.

4.2 The Backend

4.2.1 Placement

Placing computational tasks on multicomputer processor nodes is quite analogous to placing components in a VLSI layout. Algorithms to solve the placement problem have thus been well explored in both contexts. General task assignment algorithms which place multiple tasks per physical node seek to balance goals which sometimes conflict: good load balancing, minimization of interprocess communication, and a high degree of parallelism should all result from a good solution. Task assignment formulations are known to be NP-complete in all but a very few restricted cases [GJ79].

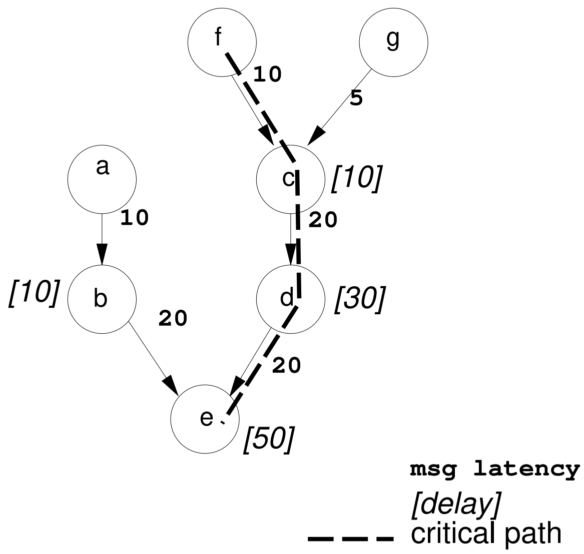
Solutions to this assignment problem impact the resulting routing and scheduling problems and optimal placement of tasks is a key component of any system designed to efficiently allocate network resources (whether statically or dynamically). Graph theoretic techniques to derive optimal mappings for *regular* communication topologies exist and are often available as system level software in parallel machine environments [Lo88]. When the communication topology is irregular, however, there is no standard way of assigning tasks to physical processors. The approach for this thesis has been to implement a task assignment module which automates placement when required using a very simple version of the more general method of simulated annealing.

In addition, since a placement routine that generates results for the *diamond lattice*

```
FIND-TASK-DELAYS▷ Find delay, release, deadline times in graph
  IF task.state = DONE return;
  ELSE
  LOOP for msg in task.dmsgs ▷ loop over all arriving messages
  FIND-TASK-DELAYS(msg.src);▷ finding max arrival time

  LOOP for msg in task.dmsgs
  delay ← max(delay,msg.arrival-time);

  LOOP for msg in task.dmsgs
   msg.deadline ← delay ▷ Set deadline of arriving msgs
   SEND-MSGS(task);
   task.state ← DONE
   end
SEND-MSGS(task)
  xmit ← task.delay + task.compute-time
  ▷ add in task computation time to task start time
  LOOP for msg in task.smsgs ▷ loop over all outgoing messages
  msg.release-time ← xmit ▷ set release time of outgoing msg
  msg.arrival-time ← xmit + msg.length ▷ set arrival time to msg length + start
 time

  end
```

Figure 4-3: Pseudo-code for Critical Path Algorithm

Figure 4-4: Critical Path Algorithm.



was desired, [5] automating the placement process using simulated annealing seemed like a useful approach. The diamond lattice is currently being studied within the NuMesh project as an alternative four neighbor topology. Details of the topology can be found in [PWM+93, WAD+93]

The task placement module allows a two-fold strategy for placing. If a mapping from a particular logical configuration to a physical network has been defined within TTDL, it is used to place tasks. Ultimately, the Numesh programming environment should include a library of such "canned" mapping which can be directly applied to specific regular communication problems as specified in the intermediate representation language. Simple mappings from a logical torus to a physical mesh have already been provided within TTDL and were used to place examples of the Jacobi relaxation problem . These results are discussed in more detail in Chapter 5. If mappings are not available, the placement module tries to optimally place tasks using simulated annealing.

Simulated Annealing

Simulated annealing [KJV83] provides a powerful method of solving optimization problems. The technique, which is rooted in statistical mechanics, has been used extensively to optimize the allocation of resources in computer design. Simulated annealing uses iterative improvement to arrive at a system configuration which minimizes some objective function. Changes to the system configuration which lower the objective function are always accepted. However if a change raises the objective function, it is accepted with probability given by the Boltzmann distribution, $\exp^{\delta} C/T$, where δC is the change in the value of the objective function. This feature prevents the optimization process from settling into local minima. T is a parameter analogous to temperature in a physical system, and its value relative to the objective function value may be varied during the execution of the simulated annealing problem. A drawback of using simulated annealing is that no useful upper bound exists on the time it takes to find an optimal solution.

In general, objective functions appropriate to task placement should include the effect of both communication cost and dependencies between tasks. Thus tasks which are independent should be placed on different physical nodes and tasks which must communicate should be placed close to each other. For the purposes of this thesis, the task assigment module places a single task per processor. Thus only communication cost is considered in the objective function computed during placement.

Communication cost can be estimated with varying degrees of precision. Usually the more accurate the model of communication cost, the more expensive it is to calculate this cost. An example of a simple cost function computes the Manhattan distance between a message source and destination. This cost obviously does not incorporate the presence of contention for network resources, but it is simple. A more sophisticated cost function might include the effect of path assignment, increasing the placement cost of a configuration which results in two messages sharing the same link. An even more detailed placement algorithm might schedule all the communication and calculate the placement cost based

[5] For which canned mappings did not exist at the time this subsystem was being developed, although they are an area of active exploration within the NuMesh project [KP93]

on actual collisions between messages in both space and time. Clearly the placement algorithm's running time is strongly affected by the choice of cost function.

Figure 4-5 shows the structure of the simulated annealing algorithm used for placement, routing and rerouting in the current system. The application communication graph, as expressed in TTDL, is reduced to a set of messages with sources, destinations and lengths. Tasks which send and receive these messages are placed in one or two phases, depending on the application requirements. It is possible to specify an initial configuration for the system. If an initial configuration is not specified, one is chosen at random. The placement proceeds by swapping two tasks at random and computing the incremental change in configuration cost. Negative changes are always accepted. Positive changes are accepted with a probability given by the Boltzmann distribution function.

Task Placement Cost Function

The cost metric used to place tasks in the current system varies during different placement phases. If a **coarse** placement phase is used, the cost function simply computes a sum of message source-destination Manhattan distances. The Manhattan distance is the total number of hops travelled from message source to message destination. Thus for a message sent from *Task A* located at a node in a network with n dimensions, with address $x_0A, x_1A..., x_nA$ to *Task B* located at a node with address $x_0B, x_1B..., x_nB$, this distance is:

$$Md = \sum_{i=0}^{i=(n-1)} abs(x_iB - x_iA)$$

The option to run a **fine** placement phase is also available in the system. The cost metric for this phase combines the Manhattan distance with an estimate of network link utilization in a product. Since link utilization is considered in the fine placement phase, message paths are assigned by this module and the resulting network peak and aggregate utilizations [6] are evaluated. This metric is more expensive to compute than the Manhattan distance. However it allows a more accurate estimate of the cost of a particular placement configuration since contention for network links is captured in the cost function. [7]

4.2.2 Routing

The goal of the routing module is to assign paths to all message source destination pairs such that some measure of the network congestion is minimized. In particular, the peak *link utilization* [Shu91] is required to be below one at the completion of the routing phase. This requirement increases the likelihood of computing a feasible set of switching schedules for the communication finite state machine during the scheduling phase. Path assignments which result in a peak network utilization > 1 always lead to infeasible solutions.

[6] Link utilization is defined in the next section

[7] Currently the necessity of a fine placement phase is not computed automatically ,it is up to the programmer to specify that it is required. Ultimately, the system should decide based for example on whether a feasible schedule is computable, whether to run a fine placement phase

Figure 4-5: Iterative Improvement Using the Boltzmann Probability Distribution Function.

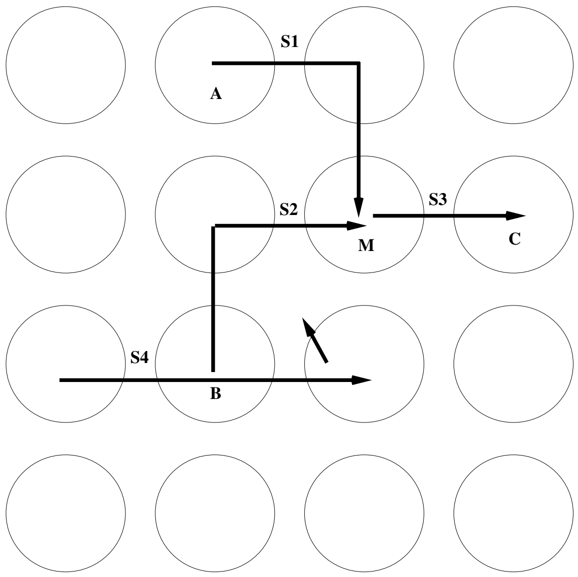
```
simanneal(Network *nw, void initial_config(),void compute_new_config(),
              void reset_configuration(),
              float deciding_cost(),
              float min_acceptable_cost(), Sa_sched *sched)
{
  float initial_temp = sched->start_t;
  float final_temp = sched->fin_t;
  float temp_step = sched->t_step;
  int n_iter = sched->n_iter_start;
  int n_iter_step = sched->n_iter_step;
  float delta_cost,t;
  int i = 0;
  float old_delta_cost = 0;
  initial_config(nw);
  for(t=initial_temp;t>final_temp;t=(t-temp_step)){
    i = 0;
    for(i=0;i<n_iter;i++){
      compute_new_config(nw);
      delta_cost = nw->cost - nw->previous_cost;
      if(delta_cost>=0){
        if(!should_accept(delta_cost,t,history)){
          reset_configuration(nw);
        }
        else accept_config(nw);
      }
      else if(delta_cost<0)
        accept_config(nw);
    }
    n_iter += n_iter_step;
 }
   /*  If we still haven't reached min acceptable cost,
     keep going.
   */
   if((t <= final_temp) &&
      (deciding_cost(nw)>min_acceptable_cost(nw)) &&
      (sched->start_t> BAIL_OUT_TEMP)) {
     sched->start_t = (initial_temp / 2);
     sched->n_iter_start = n_iter;
     simanneal(nw, do_nothing, compute_new_config,
               reset_configuration,deciding_cost,
               min_acceptable_cost, sched);
   }
}
```

Number of Available Paths

While oblivous routing limits the number of alternative paths available to a message, offline path assignment can take advantage of multiple alternative paths from source to destination. When congestion occours in time alternate paths in space can be exploited.

Consider as an example, an image processing application running on the NuMesh which requires the following simple operation: two streams, `S1` and `S2` of video input data are merged together at a node, M, using some operation like `xor`. The data in each stream must first pass some sort of threshold operation at the source node in order to qualify for merging with the other stream. This example is sketched in Figure 4-6. Assume that a high volume stream, `S4`, is also active during this communication phase. Suppose that `S2` is allocated the route shown. Data can be efficiently routed to node C using these assigned paths. This happens without the danger of contention between `S2` and `S4` which might occour at the East port of node B if the routers were programmed to route dimensionally, first in x and then in y.

Figure 4-6: Dimensional Routing Versus More Flexible Path Assignment Algorithm.



The number of available paths rises rapidly with path length and depends also on the distribution of the total distance over all available dimensions. For a two dimensional mesh, the number of available paths is given by:

$$\frac{(\delta x + \delta y)!}{\delta x! \delta y!}$$

and for a three dimensional mesh it is:

$$\frac{(\delta x + \delta y + \delta z)!}{\delta x! \delta y! \delta z!}$$

Finally the number of shortest paths available in the diamond lattice is :

$$\frac{(\delta x + \delta y)!(\delta z + \delta t)!}{\delta x! \delta y! \delta z! \delta t!}$$

Path Assignment Strategy in the Current System

The simulated annealing based optimization code written for the placement phase of the current system was written in terms of a general notion of **configuration** and functional abstractions to change a given configuration, evaluate the cost of the configuration, and restore the state of the system to a previous configuration if a new configuration is to be rejected. The same code used in placement was reused for routing; only the cost evaluation functions are different. This approach makes the software in the different phases of the system quite modular. Thus the routing module can search for a solution to the path assignment problem using the same general simulated annealing method used for placement.

The path assignment module allocates an initial route to each message active in the system. This assignment can be done in various ways: randomly assigning an initial route (a coin flip at every junction decides which link to take to reach the destination), allocating a path using dimensional routing, or assigning a route using a shortest path algorithm are all supported. Once the paths are assigned, a *link utilization function* [Shu91] is calculated for each link in the network. This utilization function measures the amount of traffic supported by each link in the entire time interval during which traffic is being analyzed. If the message length for the pth message [8] is given by L_p, the link utilization, $lu_j i$ for the jth link may be computed for n messages as:

$$lu_j = \frac{\sum_{i=1}^{n} b_{m_p l_j} L_p}{\sum_{k \in Q_p} t_{i_{k+1}} - t_{i_k}}$$

Where Q_p is the set of indices of time intervals i_k during which the n messages are active, $t_{i_{k+1}} - t_{i_k}$ is the length of the interval i_k, and $b_{m_p l_j}$ is a variable which is equal to one if message m_p is active on link l_j and zero otherwise. The division of time into intervals during which messages are active is determined by message release and deadline times.

Intuitively, this equation expresses the notion that the amount of data carried by a particular link in a given time interval must not exceed the time alloted to switch this data. Note that if $lu_j > 1$, link j is required to carry more message traffic in the given time interval than it physically can.

The amount of *slack* available to a message is the difference between its length and the time available to transmit it. Thus a message with a large number of network cycles available for its transmission (as determined by the difference between the message's deadline and release times) compared to its length has slack.

The path assignment module in the current system first searches through the network for a peak value of link utilization [9]. The messages on this peak link are next targeted for **rerouting**. The cost of the rerouted configuration is computed by summing link utilization

[8] normalized by a bandwidth B

[9] if there are several peaks of equal value, one is chosen at random

over all the links in the network. If the new cost is less than the previous cost, it is accepted. If it is greater it is accepted with a probability given by the Boltzmann distribution. The entire procedure continues until a minimal acceptable routing cost [10] is reached.

The rerouting module prioritizes messages based on a measure of their flexibility, [11] targeting the most flexible message for rerouting. It reroutes messages using a weighted shortest path algorithm instead of randomly trying one of a set of precomputed available shortest paths. This eliminates the need to compute these paths. It also allows rerouting to sense and avoid congested paths. Taking a longer than physically shortest path is a degree of freedom available to messages which do not affect the critical path of the computation. If messages are forced to choose from a precomputed set of physically shortest paths, as in [Shu91], this flexibility is not exploited.

Rerouting Using Weighted Shortest Path

Once the placement phases and initial routing phases are complete, the simulated annealing code is reused yet again in a *rerouting* phase. The rerouting module identifies peak link utilization in the network and reassigns paths to certain messages to alleviate congestion. Messages assigned to the link with peak utilization are ordered according to their flexibility. The message with the greatest flexibility is rerouted using a weighted shortest path algorithm. An all pairs shortest path algorithm such as Dijkstra's has a complexity that goes as $\Omega(N^3)$ [12] where N is the number of nodes in the graph for which shortest paths are being computed. Thus it would be quite expensive to route all messages in the system using this algorithm. Targeting only those messages which are involved in a hot spot limits the number of messages which must be routed using the shortest path algorithm. In pathological cases it is possible that all messages may need to be rerouted using the shortest path algorithm (i.e. if all messages end up using the same link) but these cases are hopefully relatively rare.

Using a *weighted* shortest path algorithm implies that messages may take a longer route than just a shortest path from source to destination. If there is a lot of traffic already present on a particular link this link will be avoided by a message being rerouted After a reroute, a message may no longer travel from source to destination via a **physically** shortest path. However if a particular message is not the limiting factor in the system throughput or if all messages in the system are long compared with the distances they travel, rerouting to a longer path will not cause much of a performance loss. In the current system a maximum misrouting distance is associated with each message [13]. If a message must be misrouted beyond this distance the next most flexible message is chosen for rerouting. The rerouting process is iterated using the same basic optimization code used in the coarse and fine placement phases. Again only the function which changes or resets a configuration and the cost evaluation function are changed.

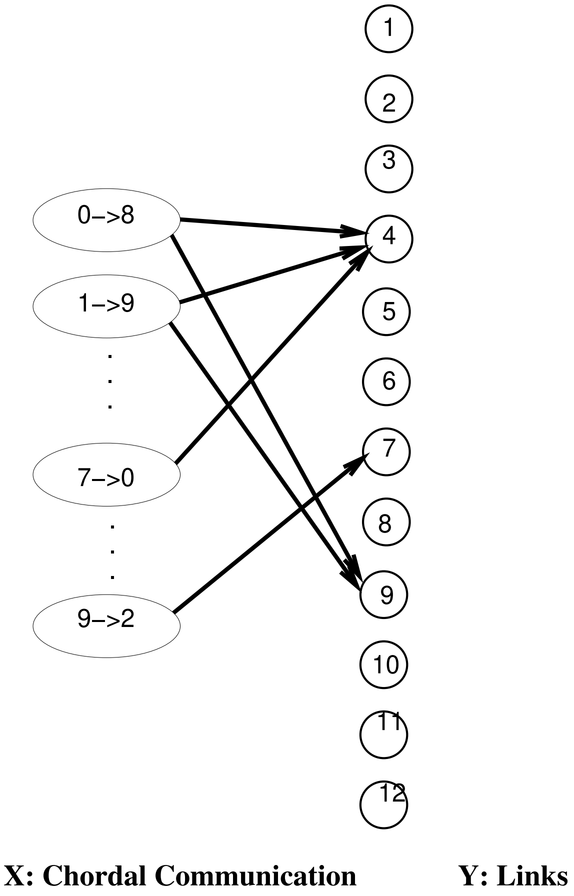
[10] currently this cost is determined by the user

[11] Message A is more flexible than message B if it has a greater number of shortest paths available from source to destination

[12] Or at best as $\Omega(N^2 logN)$ if heaps are used

[13] The maximum misrouting offset is based on the message length, the Manhattan distance between message source and destination, and the message release and deadline times

Figure 4-7: Path Assignment Algorithm Using Maximal Matching to Assign Routes.



4.2.3 Alternative Approaches to Offline Path Assignment

Maximal Matching

Alternative approaches to allocating routes to messages in a network have been explored in the literature. Maximal matching, a general technique applied to the problem of optimally routing network traffic has been used by researchers at the University of Oregon [LRea90] and by [Dah90] in a comparison of Connection Machine [Hil85] performance of some scientific computing code with and without static routing. Routing by maximal matching proceeds by examining in a given cycle, the messages emitted by each node in the system and building a bipartite graph with messages as source nodes and links as destination nodes of the graph. The edges in the graph represent allowed assignments. The messages are matched to available links using the algorithm illustrated in Figure 4-7. The left half of the bipartite graph shows messages active on this particular time step. The right half shows links available in the network. The arrows represent possible assignments from message to link. The figure shows a message travelling from node 0 to node 8 which can reach its destination with a minimal number of hops if it advances during the illustrated time step onto link 4 or link 9.

Adopting this technique has some advantages. The algorithm has a complexity of order $\Omega((M^2) * N)$ where M is the number of messages routed and N is the number of links in the network. Maximal matching is iterated for K timesteps where K is the maximum number of cycles it takes to deliver all messages starting out at an initial time, $T0$.

One disadvantage of this method is that the algorithm proceeds locally. When a link is allocated to a message at a particular network node, there is no attempt made to predict and avoid congestion which might occour at a subsequent node. Allocation of routes is greedy with respect to a given node. Thus if the maximal matching approach is used, and link assignment is done at every node for every time step, a particular route chosen based on traffic present in previous cycles may lead to a collision later on.

Computing link utilization as described above avoids this problem by incorporating temporal information into route assignment. Since link utilization includes message activity times, the value computed is actually an estimate of the fraction of time a link will be occupied.

Force Driven Routing

In [HL87], the authors introduce a global router for IC layout based on the novel idea of allocating paths by computing the trajectory of a moving particle in a force field created by point masses representing the path source and destination. This method has some advantages. By incorporating 'force fields' from all active source destination pairs during routing, results are not as dependent on the order in which wires are routed as they would be in a more traditional sequential router. It may be worthwhile to test this algorithm or some variant as an alternative routing strategy for the current system.

A Linear Program Global Router

[Tho87] presents a linear programming formulation for a global router. This approach, similar to the one adopted here for scheduling has the advantage that the entire problem is solved globally. Nets are routed simultaneously rather than sequentially, with no back-tracking. The paper shows good results for run time of the algorithm: a feasible solution for input data net sizes of upto 1266 connections was found in less then a minute.

Multiflow Approach

Probably the most promising approach to solving the problem of optimally allocating routes and transmission intervals to messages is the use of network flow algorithms. Recent work by [LMP+91] presents a fast method for solving the multicommodity flow problem. If the multicommodity flow formulation can be modified to accomodate message traffic which starts and stops intermittently it may be very useful in the current context.

4.2.4 Linear Programming Formulation of the Scheduling Problem

[SA91] have developed a linear programming formulation of the problem of scheduling messages onto network links once the messsages have been assigned routes. This formulation is based on standard scheduling techniqes used to schedule processes on multiple processors [Hor71]. If the problem is feasible, solution of the formulation generates cycle by cycle switching schedules for communication processors. The technique has the advantage of expressing the problem globally. All constraints are solved for simultaneously. In addition, allocation of bandwith to messages proceeds at a relatively coarse scale in time and is thus computationally less expensive than (for example) simulation of traffic at every time step in the network. An important disadvantage of this approach is that the formulation assumes **link simultaneity** when a message is transmitted. As in a circuit switching model of network operation, a channel is opened from source to destination when a node begins transmitting and all links in the path must be available for the duration of the message transmission. Messages are first divided into *maximal subsets*. They are then assigned the bandwidth they require based on their length and on their release and deadline times. Finally each message is simultaneously scheduled onto all links used by the message during its transmission.

Maximal Subsets

In order to reduce the number of constraints which must be satisfied in solving the message scheduling problem, all the messages active in a given phase are first divided into maximal subsets. These sets are then treated as independent groups of messages for which the message interval allocation and interval scheduling problems (described below) are solved. Maximal subsets contain messages which conflict with each other in **both** space and time either directly or transitively. Thus if both message A and message B occupy link l_0 and time interval i_0, they are assigned to the same maximal subset. If A conflicts in this way with B and B conflicts with C than B and C are assigned to the same maximal subset as A.

Once messages have been grouped into maximal subsets, constraint equations based on *message interval allocation* and *interval scheduling* are generated.

Message Interval Allocation

Message interval allocation is the problem of assigning transmission times to messages within a given interval. Two conditions must be satisfied in order for the linear programming module to find a feasible solution to the message scheduling problem. Consider a maximal subset of messages $S_m = m_0, m_1, ...m_n$ which are active in a set of intervals $S_i = i_0, i_1, ...i_n$ and active on a set of links $S_l = l_0, l_1, ...l_n$ For each message $m_p \in S_m$, the following equation must hold:

$$\sum_{i=1}^{n} a_{m_p i_k} x_{m_p i_k} = \frac{m_p}{B}$$

This equation merely states that the sum of all transmission intervals alloted to a message must be equal to the length of the message (normalized by the network bandwidth). $a_{m_p i_k}$ is a variable which is equal to one if m_p is active in interval i_k and equal to zero if it is not. $x_{m_p i_k}$ is the transmission time assigned to message p in interval k and is the variable being solved for. Thus the message interval allocation problem assigns values of $x_{m_p i_k}$ for each message in each interval.

To ensure that no link is overutilized during any given interval, the following equation must also hold for every link, l_j and every interval i_k used by message m_p in maximal subset S_m:

$$\sum_{m_p \in S_m} a_{m_p i_k} b_{m_p l_j} x_{m_p i_k} \leq \delta_{i_k}$$

This equation ensures that on any given link, the sum of transmission times alloted to all messages using this link in a given interval does not exceed the size of the interval. Here $b_{m_p l_j}$ is a variable which is equal to 1 if message m_p occupies link l_j.

These constraints, along with the objective function:

$$\delta C = \sum_{m_p \in S_m} a_{m_p i_k} b_{m_p l_j} x_{m_p i_k} - \delta_{i_k}$$

were supplied to lp solve as a file of equations. As an objective function I chose the excess capacity on network links. This choice of objective function is similar to the approach taken by [Tho87], where a linear programming formulation for global routing of VLSI components is developed. The excess capacity of a link is the difference between the amount of traffic allocated to the link in a given interval and the size of the interval.

Interval Scheduling

Message interval allocation allocates transmission time for each message in each interval. Interval scheduling adds the constraint of simultaneous link availability. This constraint

[14] requires that a clear path is available for a particular message when the message is scheduled for transmission. Again, a set of equations enforce this constraint. Figure **??** shows a simple example of a case where a solution to the message interval allocation problem has been found, the constraint of simultaneous link availability is not satisfied.

Interval scheduling groups the messages in a maximal subset into *link feasible sets*. The intersection of the set of links used by m_a and the set of links used by m_b is $\emptyset$ if m_a and m_b are in the same link feasible set. Messages in the same link feasible set can be transmitted simultaneously. The link feasible sets are then assigned transmission times y_j during which all the messages in the link feasible set are transmitted simultaneously such that the following constraint holds for each interval i_k and each message m_p

$$\sum_j y_j = x_{m_p i_k}$$

where $j \in Q_p$ and Q_p is the set of indices of link feasible sets in the maximal subset which contain message m_p. The objective function minimized is:

$$\sum_{j=1}^{j=N-link-feasible-sets} y_j$$

Again, a file of equations is input to the linear program solving program lp solve, described below, which produces transmission times for each message in each interval.

Linear Programming Module: lp solve

Given the constraints described in the previous section, it is straightforward to set up a system of equations and associated objective functions which can then be solved by a linear programming module. **Lp solve** [Ber], a shareware program integrated into the current system to handle message scheduling problems, was developed to solve mixed integer linear programming problems. Lp solve uses a 'Simplex' algorithm and sparse matrix methods for pure LP problems. If one or more of the variables is declared integer the Simplex algorithm is iterated with a branch and bound algorithm until the desired optimal solution is found. For the problems analyzed in this thesis (some of which produced close to one hundered variables) lp solve proved efficient and reliable. Thus any future versions of the scheduling component of the current system would do well to include some of the lp solve technology. Ideally, lp solve source code should probably be revised to identify which messages, links and intervals are the cause of infeasibility in generating schedules and should provide this information to the rest of the system.

Propagation of Results from Message Scheduling

In the current system results from the linear programming solver do not propagate back to the rest of the system. If a feasible schedule can not be found for a given configuration information from this module about which messages are responsible for the intractibility of

[14]which should utlimately be relaxed in the next generation of scheduled routing software developed for the NuMesh

the problem, and thus which message deadline times should be relaxed should propagate back up towards the higher level compilation modules. Communication can then be rearranged and schedules recomputed if necessary. In the time frame during which the current system was developed, code to isolate this information and communicate it to other parts of the system was not implemented. The linear programming solver was instead used as a black box. This should certainly be rectified in any revisions to the system since the freedom to move deadline times forward would provide an important relaxation step in the overall optimization problem.

Problems with Linear Programming Approach

There are a number of problems with the approach taken here to schedule messages onto network links. The current formulation of the problem allows messages to be fragmented across intervals. Messages are not constrained to remain contiguous. This assumption leads to an interesting breakdown when messages are broken up by the scheduler. Every message transmission time must be padded out with a 'drain time' to allow the message fragment transmitted during a given interval to reach its destination. If a message fragment relinquishes the network links to which it has been assigned before the entire message fragment has reached its destination message data will be lost. Thus every message transmission interval must be padded with enough cycles to allow for the worst case distance a message might travel which is typically taken to be the network diameter. If messages are long compared to the network diameter, this padding is not a large percentage of the total message transmission time in the network. However if messages are short they should not be fragmented across time intervals. Keeping messages contiguous leads to difficulties in expressing the problem as a simple linear program, however.

Refining the Approach to Scheduling Messages on Links

Assuming that messages are simultaneously active on all links used during transmission leads to potential performance loss. A refinement of the message scheduling system implemented here is needed. One approach is to build a more complete representation of communication operations. Currently the module which assigns release and deadline times to messages assumes that a message is released from its source at a particular network clock cycle and that it arrives at its destination by some deadline time. The DAG currently used to assign release and deadline times could be refined to include a cycle by by cycle representation of what links the message traverses from source to destination. This allows a message to get scheduled onto a link which has been relinquished by some other message before all the data in this other message has completely arrived at its destination.

In the limit this approach amounts to simulating message traffic cycle by cycle in order to more accurately schedule messages onto links. There is a tradeoff here between the size of the time interval in which message link usage is accurately known, and the space and time required to represent and make use of this information.

u

Chapter 5

Results

This chapter presents the results of applying the backend placement, routing and message scheduling software described in Chapter 4 to example communication graphs. As test examples I chose some simple problems which had previously been hand-coded for the NuMesh environment. An implementation of the *Multigrid* algorithm was programmed manually and simulated in the NuMesh simulation environment [Met91] when the simulator was first developed. The Jacobi relaxation problem which has similar, though less complicated, communication characteristics was expressed in TTDL and routed and scheduled by the backend. Because TTDL functions can be applied to network objects of varying sizes and topologies, it was a simple matter to generate results for different instances of the Jacobi problem. Thus routing and scheduling for an instance of the two dimensional Jacobi problem on two dimensional meshes of four, sixteen and sixty-four nodes and an instance of the three dimensional Jacobi problem on a three dimensional mesh of twenty-seven nodes were programmed and the resulting DAG structure input to the backend. The communication patterns present in the Jacobi relaxation problem are very simple. All nodes communicate repeatedly with their nearest neighbors. Thus it provided a nice test example for debugging purposes. It also allowed a test of the mapping function facility which calls a user-defined mapping function to place tasks for a given problem onto a particular toplogy. The mapping function supplied for this problem maps tasks defined in a logical torus to a physical mesh.

Results were also generated for a decomposition of the one dimensional FFT algorithm. Again it was straightforward to express the communication graph for the one dimensional fft in TTDL. This example is more characteristic of the types of applications for which the software developed here is targeted. Periodic real time applications for image and audio processing compute FFTs repeatedly and typically require high throughput from both the computation and communication hardware on which they execute. The backend software identified and extracted the maximum possible throughput for this simple problem.

Finally the communication graph for the lexical search module of a real time speech recognition application previously hand coded for the NuMesh was expressed in TTDL and input to the backend.

5.1 Jacobi Relaxation Problem

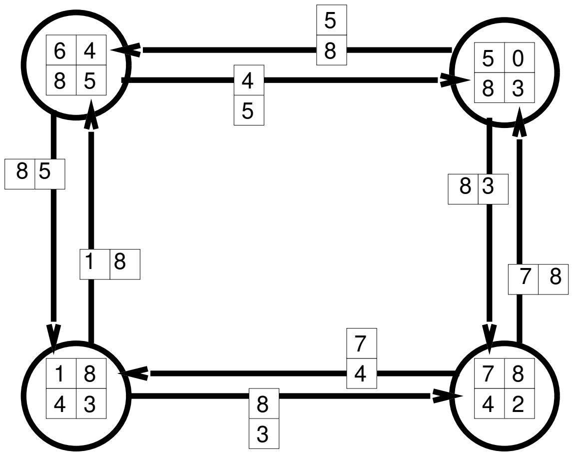
Jacobi relaxation is an iterative algorithm which, given a set of boundary conditions finds discrete solutions to differential equations of the form $\nabla^2 A + B = 0$. The difference equation

$$A_{i,j}^{k+1} = \frac{A_{i+1,j}^{k} + A_{i-1,j}^{k} + A_{i,j+1}^{k} + A_{i,j-1}^{k}}{4} + b_{i,j}$$

.

Jacobi differs from other iterative relaxation alogrithms in that the update of each point (at iteration step k+1) requires the *previous* values of the neighboring points. Figure 5-1 gives an example of a four node Jacobi relaxation problem. The circles representing processors are shown exchanging data during one iteration of the relaxation.

Figure 5-1: Communication Pattern for Jacobi Relaxation Problem.



5.1.1 TTDL Description of Jacobi Relaxation Problem

Figure 5-2 shows a program which expresses this same communication pattern in TTDL. A logical definition for a two dimensional Jacobi relaxation problem giving the communication pattern between a source task and its north, south, east and west neighbors is defined using `define-comm`. A definition for the three dimensional version of the problem, which is different only in that information is exchanged between (logical) neighbors in all three dimensions, is also shown.

A DAG representation of the communication is generated from an invokation of this definition of the two dimensional Jacobi problem, as shown in Figure 5-4. An embedding function was also provided in TTDL to map a logical torus to a physical mesh.

The backend was passed the file shown in Figure 5.1.1. This file contains:

1. A description of the physical network for which the problem is targeted.

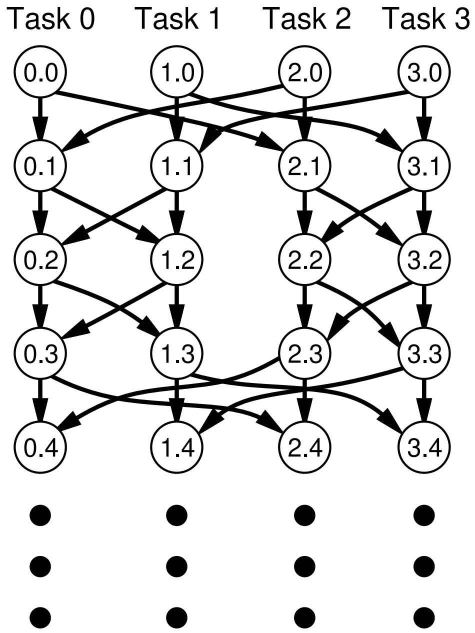
Figure 5-2: A Call to Define-comm Establishes the Definition of a Communication Pattern for the Two and Three Dimensional Jacobi Problem.

```
;; Two dimensional Jacobi problem

(define-comm jacobi-2d
  (nw source-task t-north t-east t-west t-south n)
  (let ((compute-time 10))
  (sequence
  (synchronize 0)
  (iterate
     (sequence
       (write-stream m0 t-north)
       (read-stream  m0 t-south)
       (write-stream m0 t-east)
       (read-stream m0  t-west)
       (write-stream m0 t-west)
       (read-stream   m0  t-east)
       (write-stream  m0 t-south)
       (read-stream  m0 t-north)
       (compute compute-time)
       )
     n)))))


;; Three dimensional Jacobi problem is very similar

(define-comm jacobi-3d
  (nw source-task t-north t-east t-west t-south t-up t-down n)
  (let ((compute-time 10))
  (sequence
  (synchronize 0)
  (iterate
     (sequence
       (write-stream m0 t-north)
       (read-stream  m0 t-south)
       (write-stream m0 t-east)
       (read-stream m0  t-west)
       (write-stream m0 t-west)
       (read-stream   m0  t-east)
       (write-stream  m0 t-south)
       (read-stream  m0 t-north)
       (write-stream  m0 t-up)
       (read-stream  m0 t-down)
       (compute compute-time)
       )
     n)))))
```

Figure 5-3: Invoking Jacobi-2d on a *Logical Mesh*, a *Physical Mesh*, a *Placement Strategy* and a File Name

```
;; two dimensional Jacobi problem mapped to two dimensional mesh
(jacobi-2d  *med-2d* *med-2d* *dont-place*
    "~milan/mathesis/bin/test-med-j-2d.tasks" )
;; two dimensional Jacobi problem mapped to diamond mesh
(jacobi-2d  *med-2d* *diamond* *do-place*
    "~milan/mathesis/bin/med-j-diamond.tasks" )
;; three dimensional Jacobi problem mapped to three dimensional mesh
(jacobi-3d  *med-3d* *med-3d* *dont-place*
    "~milan/mathesis/bin/test-med-j-2d.tasks" )
;; three dimensional jacobi problem mapped to two dimensional mesh
;; no mapping  was specified in definition of 3 dimensional Jacobi problem
(jacobi-3d *small-3d* *med-2d*  *do-place*
   "~milan/mathesis/bin/test-med-j-2d3d.tasks" )

;; a network object
(defvar *med-2d*
(make-network :topology "2d-mesh"
:routing "random" :n 16 :ndim 2 :nx 4 :ny 4))
```

Figure 5-4: DAG Generated by Evaluation of TTDL for 2d Jacobi Problem.



2. A description of message traffic in the network with message source and destination tasks, message size (which is 8 units for all messages) and release and deadline times.

3. A directive to the backend about which placement strategy to use. This can be a request to place (possibly providing an initial configuration) or an indication that a mapping is already specified in the file.

Since a mapping function was defined for this problem, a placement is already provided in this file.

5.1.2 Results of Routing and Scheduling

Figures 5-6 and 5-7 show snapshots of the communication pattern as scheduled and routed onto a physical two dimensional four by four mesh. Circles represent processor nodes and are labeled with the id of the resident task. A total of sixty-four messages, four from each processor, are transmitted during execution of the communication pattern. Transmission of sixteen messages exchanged along the x axis in the East-West directions is scheduled for nine network cycles, as shown in Figure 5-6 . Similarly transmission of sixteen messages along the y axis in the North-South direction is shown in Figure 5-7. A total of thirty-six network cycles are used to fully exchange data between all nodes in the mesh.

Schedules generated and assigned to communication finite state machines resident on processors 0 - 2 are shown in Figure 5-9.

An example of the three dimensional Jacobi relaxation problem mapped to a three dimensional grid is also shown, in Figure 5-8. The `define-comm` for Jacobi 3d is quite similar to the Jacobi 2d `define-comm`. The Jacobi-3d function was invoked on a three dimensional network object to generate a graph representing the communication pattern and this graph was passed to the backend.

5.2 One Dimensional FFT Problem

The Fast Fourier Transform (FFT) [Opp75] is a common operation in real time processing of video and audio signals. Figure 5-10 shows the communication graph for a decomposition of a one dimensional FFT. Computation times are shown in the circles representing tasks. Messages travelling between nodes are shown as arrows labeled with message lengths in units of network bandwidth.

The TTDL expression describing this decomposition of the FFT is shown in figure 5-11 the DAG produced by executing this code was passed to the backend.

Once the DAG is generated, the message release and deadline times are assigned. These times are computed using the recursive procedure described in Chapter 4. If the programmer has specified that the maximum thruput of the system should be computed, code to search the DAG for the maximum length computation or communication stage in the graph is executed. A relaxation step which moves deadline times forward for messages which have some slack relative to this execution time, is then performed. In this example, each node reads a 4 or 8 unit message from either one or two sources, computes for 12 units,

Figure 5-5: File Format Output by Execution of TTDL

```
#Model
TDL
#Ntasks
16
#Nphases
1
#nNodes
16
#nx
4
#ny
4
#nz
0
#nt
0
#****************************
# Message Traffic Description
#****************************
#Nth phase
0
#Nmsgs
64
#***********************************************
# Message format:
# msg := src-task release deadline size dest-task
#***********************************************
0 0 8 4 12
4 0 8 4 0
    .
    .
    .
15 24 32 4 14
#****************************
# Placement Description
#****************************
#Placement mode :
# 1 directs backend to place using simulated annealing
# 2 directs backend to use specified mapping which follows
2
#N tasks to read for specified placement for this problem
16
# Task format: task-id wired-down-or-not x-loc y-loc
# where wired-down-or-not identifies tasks which should
# not be moved by the placer.
0 1 0 0
1 1 0 2
   .
   .
   .
14 1 1 3
15 1 1 1
```

and writes a 4 or 8 unit message to either one or two destinations. Thus the maximum thruput for this example, if the task flow graph is to be invoked periodically is 12. Slack of 4 units is available to message `M16`, for example, as it travels from task 8 to task 12.

Routes for messages were allocated randomly and congestion was detected and eliminated using the rerouting algorithm outlined in Chapter 4. Snapshots of message traffic during the steady state execution of the FFT are shown in Figures 5-12 and 5-13. Figure 5-13 shows a simple example of two messages `M5` and `M7` which the offline router rerouted, avoiding a collision. `M5` is transmitted first in the **y** dimension and then in **x**, while `M7` is transmitted in x and then in y. In this case messages could also have been reordered at a higher level to avoid a collision. It's simpler here to rely on the routing software which is actually allocating the paths to avoid congestion when it can, rather than propagating information back up to the compiler about reordering messages.

Figure 5-14 shows the same problem routed and scheduled onto a three dimensional mesh. In three dimensions the path for message `M5` is two hops instead of three.

5.3 Viterbi Search Communication Graph

As a final example, a communication graph corresponding to the Viterbi search algorithm [1] was expressed in TTDL, the optimal thruput was computed, and the graph was placed and routed. Figures 5-15 and 5-16 show the TTDL description of this graph. This decomposition also uses algorithmic partitioning to implement a pipelined version of the Viterbi search used to perform lexical access as part of a real time speech recognition system. Communication and computation alternate in phases. First the host node, `Node 0`, distributes the lexical network to each processor. Each processor is then able to perform a Viterbi search for its own part of the lexical network. Figure 5-17 shows the execution of the first phase of communication, distribution of the lexicon data. Figure 5-18 shows the steady state execution of the pipeline, with each of nodes `1` through `7` receiving broadcast speech data from the host. As will be discussed in Chapter 6 there was not enough time to implement a multicast facility to route and schedule a single message destined for multiple locations as part of the current system. In future versions of this system the data shown here travelling thru a linear pipeline from host to nodes `1 - 7` would be simultaneously transmitted from the host node `0` to the destination Viterbi nodes.

[1] which had previously been manually implemented for the NuMesh environment [Duj92]

Figure 5-6: 2d Jacobi Problem on 2d Mesh East-West Transmission.

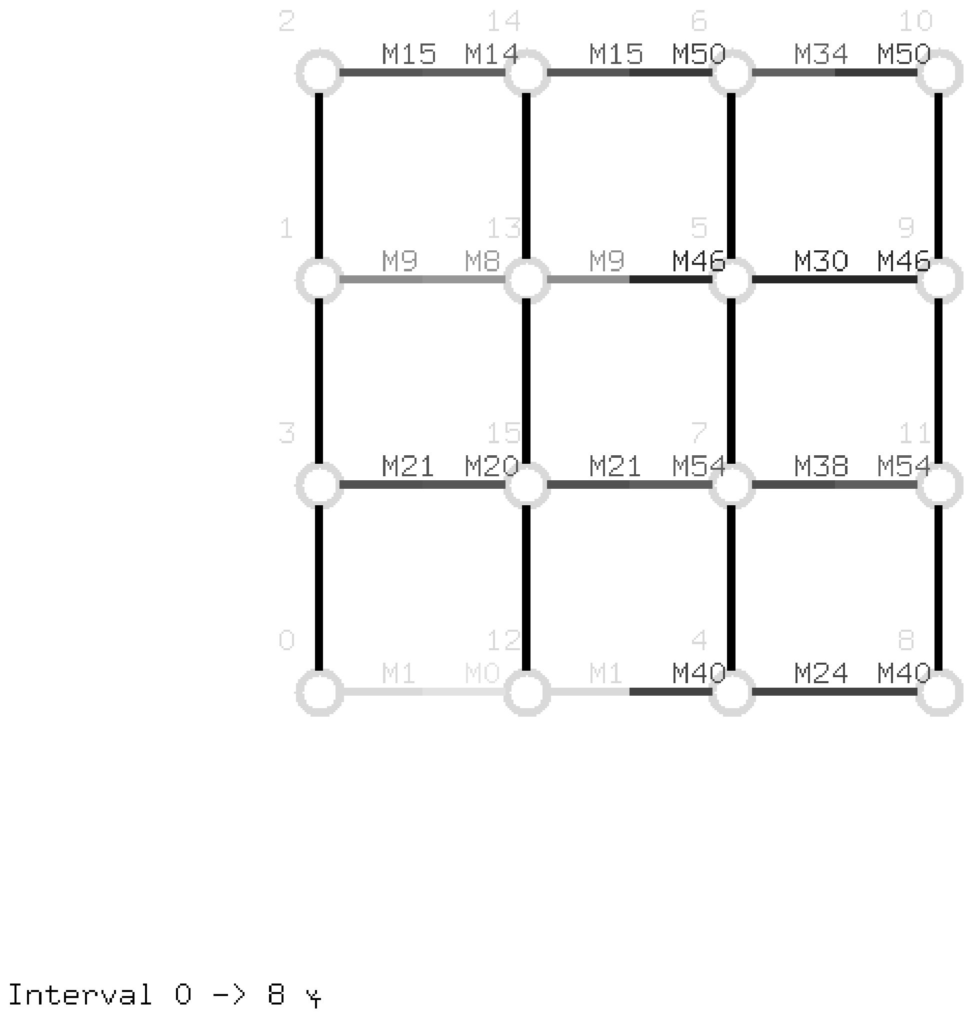


Figure 5-7: 2d Jacobi Problem on 2d Mesh North-South Transmission.

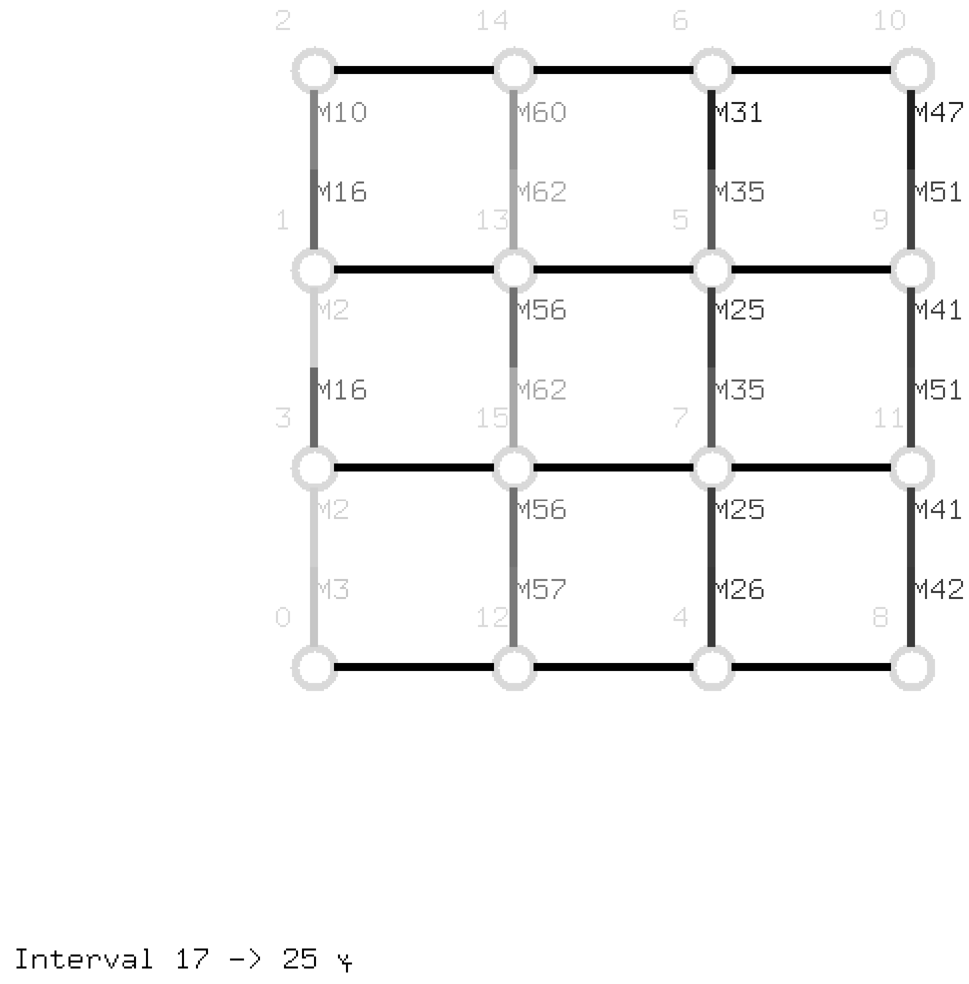


Figure 5-8: 3d Jacobi Problem on 3d Mesh North-South Transmission.

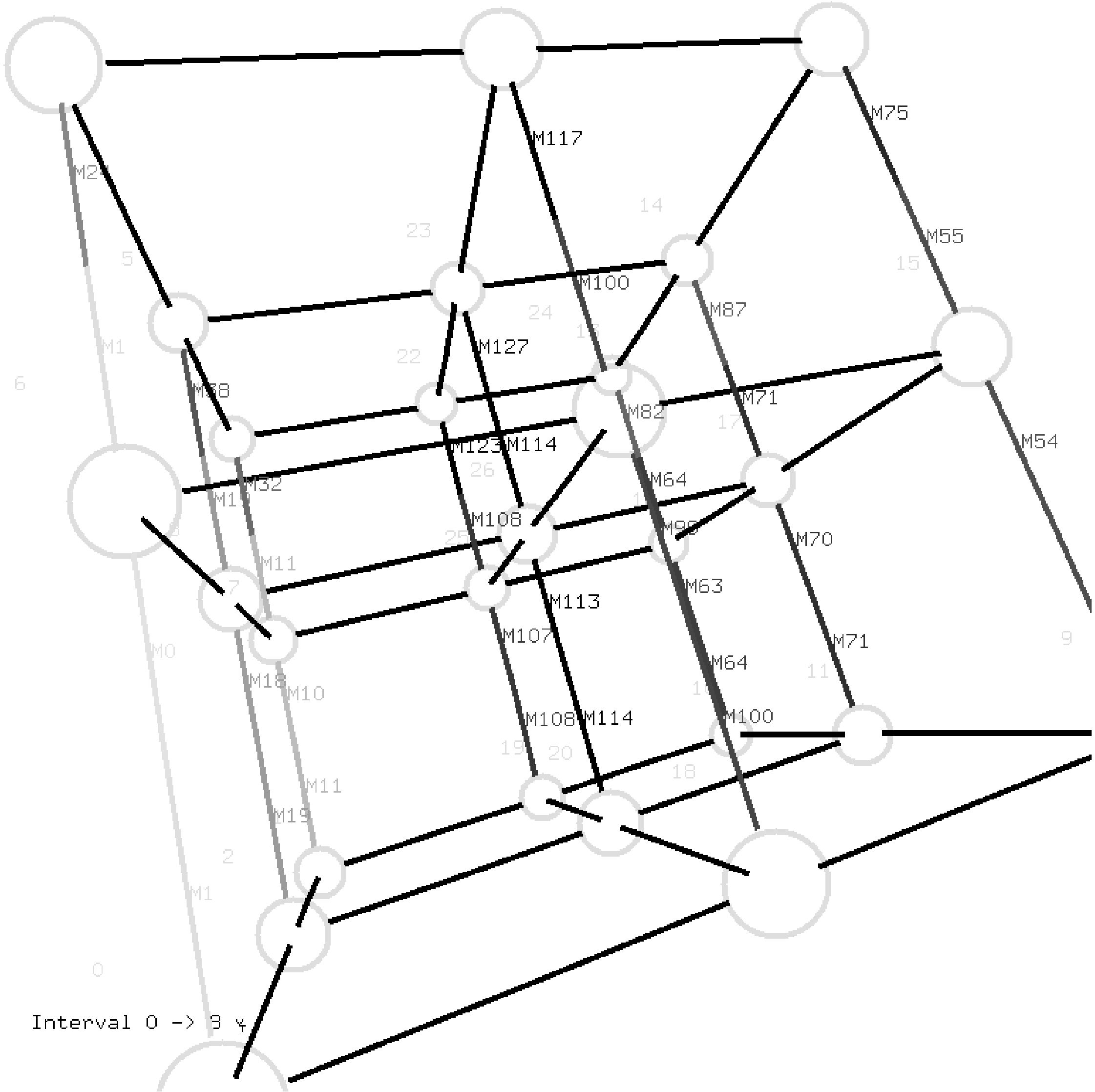


Figure 5-9: Switching Schedule Assigned to Communication Finite State Machine Resident on Processors 0-2.

```
[Schedule for node : 0]
#Instruction# WAIT-FOR-VALID-BIT
#Instruction# SWITCH Msg:0 Start: 0 End: 8 In: DownFifo Out: East
#Instruction# SWITCH Msg:1 Start: 0 End: 9 In: East Out: UpFifo
#Instruction# SWITCH Msg:2 Start: 17 End: 26 In: DownFifo Out: North
#Instruction# SWITCH Msg:3 Start: 17 End: 25 In: North Out: UpFifo
#Instruction# SWITCH Msg:4 Start: 34 End: 42 In: DownFifo Out: North
#Instruction# SWITCH Msg:5 Start: 34 End: 43 In: North Out: UpFifo
#Instruction# SWITCH Msg:6 Start: 51 End: 60 In: DownFifo Out: East
#Instruction# SWITCH Msg:7 Start: 51 End: 59 In: East Out: UpFifo

[Schedule for node : 1]
Instruction WAIT-FOR-VALID-BIT
Instruction SWITCH Msg:20 Start: 0 End: 8 In: DownFifo Out: East
Instruction SWITCH Msg:21 Start: 0 End: 9 In: East Out: UpFifo
Instruction SWITCH Msg:16 Start: 17 End: 26 In: North Out: UpFifo
Instruction SWITCH Msg:2 Start: 17 End: 26 In: South Out: North
Instruction SWITCH Msg:3 Start: 17 End: 25 In: DownFifo Out: South
Instruction SWITCH Msg:17 Start: 34 End: 43 In: DownFifo Out: North
Instruction SWITCH Msg:4 Start: 34 End: 42 In: South Out: UpFifo
Instruction SWITCH Msg:5 Start: 34 End: 43 In: North Out: South
Instruction SWITCH Msg:22 Start: 51 End: 60 In: DownFifo Out: East
Instruction SWITCH Msg:23 Start: 51 End: 59 In: East Out: UpFifo

[Schedule for node : 2]
Instruction WAIT-FOR-VALID-BIT
Instruction SWITCH Msg:8 Start: 0 End: 8 In: DownFifo Out: East
Instruction SWITCH Msg:9 Start: 0 End: 9 In: East Out: UpFifo
Instruction SWITCH Msg:10 Start: 17 End: 25 In: DownFifo Out: North
Instruction SWITCH Msg:16 Start: 17 End: 26 In: North Out: South
Instruction SWITCH Msg:2 Start: 17 End: 26 In: South Out: UpFifo
Instruction SWITCH Msg:11 Start: 34 End: 42 In: North Out: UpFifo
Instruction SWITCH Msg:17 Start: 34 End: 43 In: South Out: North
Instruction SWITCH Msg:5 Start: 34 End: 43 In: DownFifo Out: South
Instruction SWITCH Msg:12 Start: 51 End: 60 In: DownFifo Out: East
Instruction SWITCH Msg:13 Start: 51 End: 59 In: East Out: UpFifo
```

Figure 5-10: Communication Graph for 16 Node One Dimensional FFT

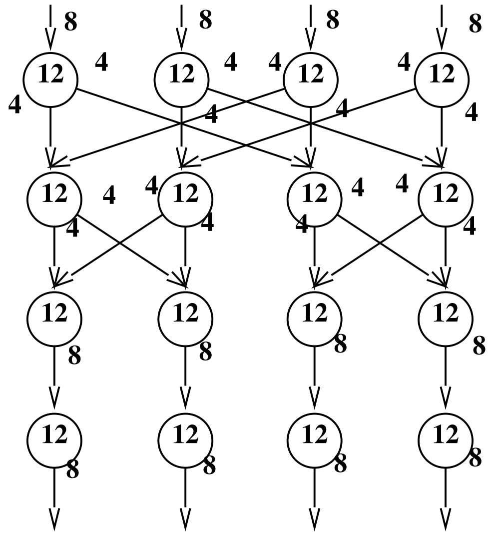


Figure 5-11: A Call to Define-comm Establishes Definitions of Communication Patterns for the One Dimensional FFT.

```
(defvar *compute-time* 12)

(define-comm fft-initial
    (source-task task-a task-b task-c task-d)
  (broadcast m1 task-a task-b task-c task-d))

(define-comm fft-first-row
    (source-task task-a task-b task-c n)
 (sequence
  (synchronize phase-zero)
  (iterate
   (sequence
    (read-stream m1 task-a)
    (compute *compute-time*)
    (write-stream m1 task-b)
    (write-stream m1 task-c)
   n))))

(define-comm fft-second-row
    (source-task task-a task-b task-c task-d n)
  (sequence
   (synchronize phase-zero)
   (iterate
    (sequence
     (read-stream m1 task-a)
     (read-stream m1 task-b)
     (compute *compute-time*)
     (write-stream m1 task-c)
     (write-stream m1 task-d)
    n))))

(define-comm fft-third-row
    (source-task task-a task-b task-c n)
  (sequence
   (synchronize phase-zero)
   (iterate
    (sequence
     (read-stream m1 task-a)
     (read-stream m1 task-b)
     (compute *compute-time*)
     (write-stream m1 task-c)
    n))))

(define-comm fft-fourth-row
    (source-task task-a n)
  (sequence
   (synchronize phase-zero)
   (iterate
    (sequence
     (read-stream m1 task-a)
     (compute *compute-time*)
    n))))
```

Figure 5-12: Pipelined FFT Problem on 2d Mesh.

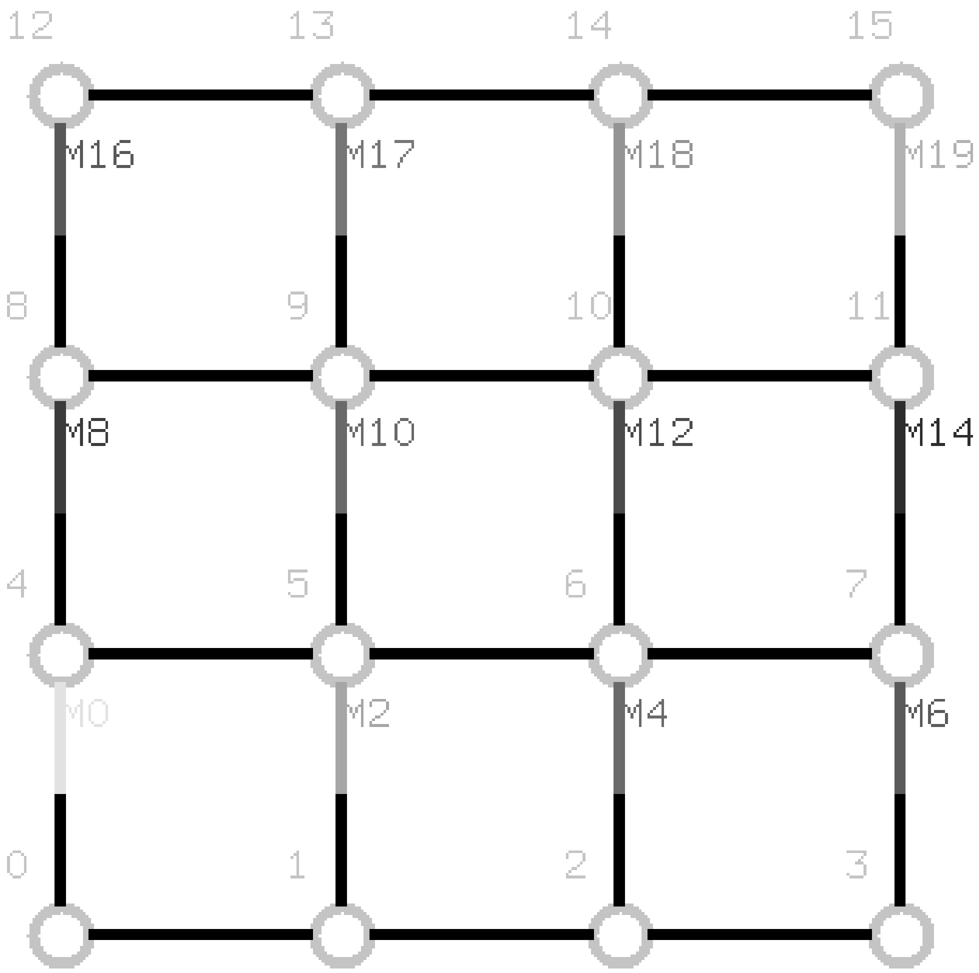


Figure 5-13: Pipelined FFT Problem on 2d Mesh.

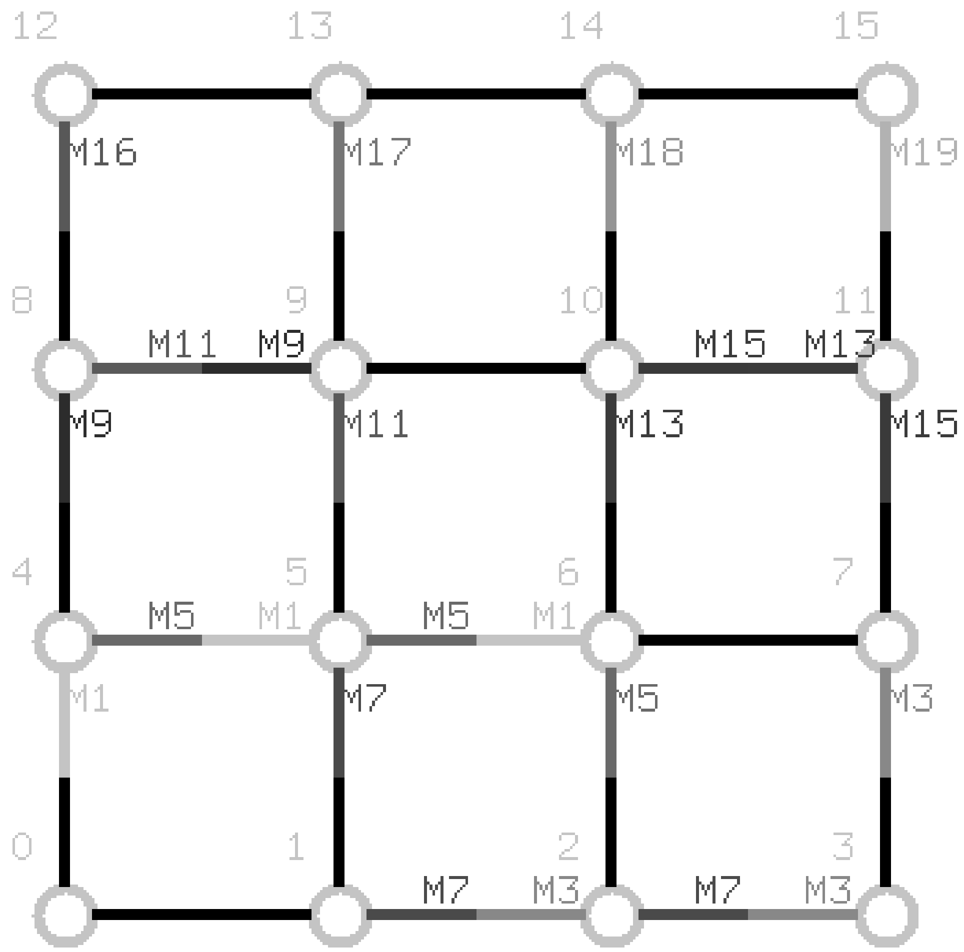


Figure 5-14: Pipelined FFT Problem on 3d Mesh.

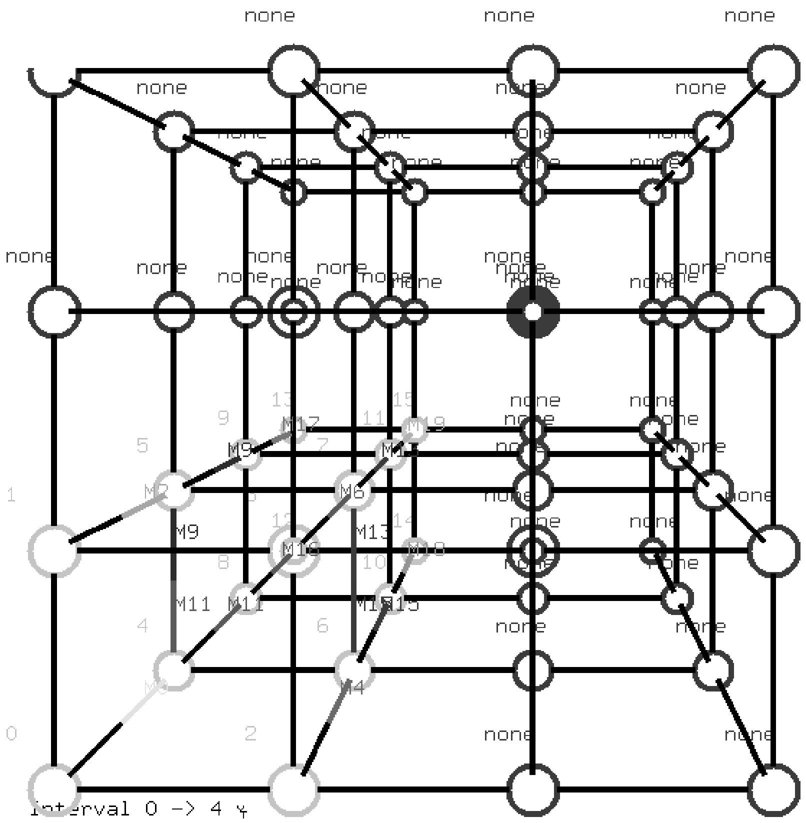


Figure 5-15: A Call to Define-comm Establishes Definitions of Communication Patterns for Host Node in Viterbi Search.

```
(defvar lexicon 1500)
(defvar speech-data 1500)
(defvar results 1500)

(define-comm vhost
    (source-task d0 d1 d2 d3 d4 d5 d6  n m)
  (sequence
   (synchronize phase-zero)
   ;; phase-zero  distribute lexicon amongst viterbi nodes
   (write-stream lexicon d0)
   (write-stream lexicon d1)
   (write-stream lexicon d2)
   (write-stream lexicon d3)
   (write-stream lexicon d4)
   (write-stream lexicon d5)
   (write-stream lexicon d6)
   (synchronize phase-one)
   ;; phase-one send speech data
   (iterate
    (write-stream speech-data d0)
    n)
   (synchronize phase-two)
   ;; phase-two  receive results
   (iterate
    (read-stream results d0)
    m)))
```

Figure 5-16: A Call to Define-comm Establishes Definitions of Communication Patterns for Nodes Executing Viterbi Search.

```
(define-comm viterbi
    (source-task host prev next n m)
  (let ((ct 3000))
    (synchronize phase-zero)
    (read-stream lexicon host)
    (synchronize phase-one)
    (iterate
     (sequence
      (read-stream speech-data prev)
      (compute ct)
      (write-stream speech-data next))
     n)
    (synchronize phase-two)
    (iterate
     (sequence
      (read-stream results next)
      (compute ct)
      (write-stream results prev)
     m)))

(define-comm viterbi-end
    (source-task host prev n m)
  (let ((ct 2000))
    (synchronize phase-zero)
    (read-stream lexicon host)
    (synchronize phase-one)
    (iterate
     (sequence
      (read-stream speech-data prev)
      (compute ct))
     n)
    (synchronize phase-two)
    (iterate
     (sequence
      (compute ct)
      (write-stream results prev))
     m)))
```

Figure 5-17: Distribute Lexicon.

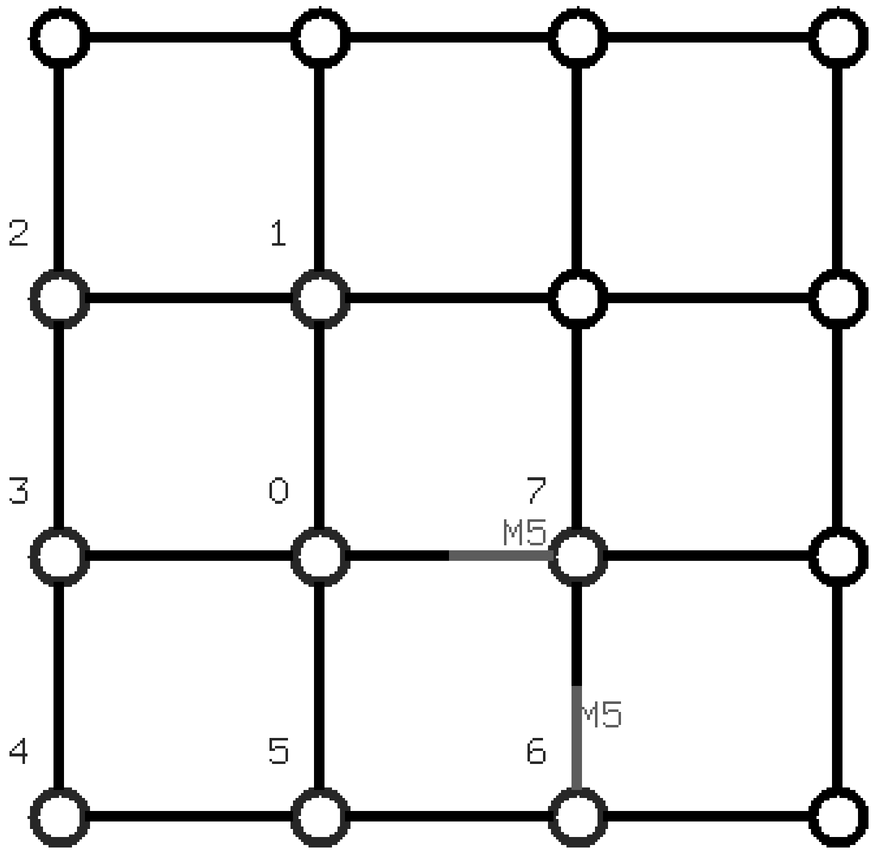
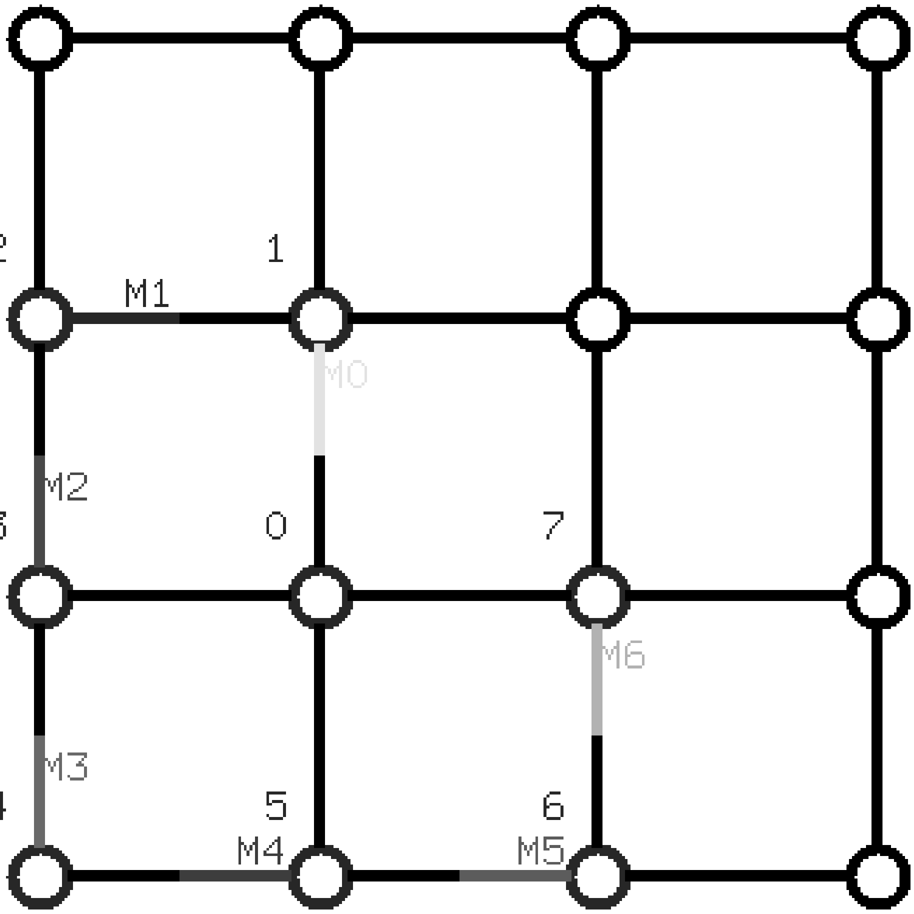


Figure 5-18: Broadcast Speech Data.



Chapter 6

Conclusions and Extensions

6.1 Conclusions

In a multicomputer, application communication patterns in *time* are translated by virtue of a particular network configuration into spatial and temporal traffic patterns. When these patterns are predictable and compact, compilation technology which solves optimization problems of the sort described in this thesis can be applied to the problem of allocating network bandwidth to network traffic.

One of the goals of the NuMesh project as a whole is to develop compilation techniques which analyze communication in a distributed system. A subgoal of this larger work, the use of these predictions to optimize the allocation of network resources, was the subject of this thesis. This work explored some simple techniques to distribute network traffic in space and time. An intermediate form to express communication patterns and a backend placement, routing and scheduling system to allocate network bandwidth tailored to these patterns was developed. Results from the system assigning schedules and paths to traffic generated by some well known problems showed that we can automate the process of allocating network bandwidth for applications with fairly static communication needs using well known optimization techniques. In particular, results from offline routing of the FFT example showed that by exploiting multiple available shortest paths from source to destination, a high throughput for a pipelined task flow graph partioning of the FFT problem was achieved.

The techniques used to allocate routes and schedules to messages used in this thesis, simulated annealing and global allocation of schedules using linear programming, worked well for the simple and rather small applications analyzed by the system. As problem sizes grow, more sophisticated methods will need to be applied. Since static routing of network traffic shares many of the characteristics of static placement and routing of VLSI components (with the added dimension of time), it may be worthwhile to harness some of the state of the art industrial technology already developed to solve such problems. A useful next step in developing more accurate and efficient network resource allocation modules for the next generation of scheduled routing software developed for the NuMesh would be to explore the extent to which algorithms used in commercially available CAD systems might be incorporated into the general framework for analysis setup in this thesis.

In general, network flow methods such as the multicommodity flow approach mentioned in Chapter 4 should be examined as possible approaches to solving the problems outlined in this thesis. These algorithms will probably yield the best solutions. They also have the advantage that unlike simulated annealing, which arrives at a solution iteratively, an upperbound exists on their running time.

6.2 Extensions

6.2.1 Compact Representation of Network Traffic

One of the most interesting problems which results from trying to statically route network traffic is the question of how to compactly represent at compile time, the traffic which will be present in the network at run time. Since placement, routing and scheduling algorithms all depend on the amount of traffic analyzed, it is important to identify traffic *patterns*. Patterns provide an efficient representation of data since they capture the structure of repeated information. Consider the problem of describing traffic present in the network when the components of a program distributed amongst the processors of a multicomputer send data repeatedly at some characteristic frequenc(ies). If the distributed components of the program send data at the same frequency ω and with the same phase ϕ (to within the accuracy of available clock synchronization technology) and for an infinite interval, there is no need to analyze all messages generated over the interval (which is after all infinite). It is sufficient to statically route message traffic present in an interval with period $1/\omega$. This traffic simply repeats over time and thus minimum-contention routes and schedules allocated for one period of data transfer will apply to each repeated iteration. A generalization of this technique might look for the least common multiple of traffic source repetition frequencies to determine an interval in which to analyze contention.

At a high level, the knowledge of repetition present in network communication patterns may be extracted from the source level language. The same compiler which computes upper bounds on block execution times will have access to iteration constructs which, when executed, result in repeated insertion or extraction of network traffic.

Thus, analysis of network transfer operations executed withing loop constructs executed on different processors to determine resultant traffic patterns should form an important component of compiler technology developed for the NuMesh. In fact the benefits of this analysis (and perhaps backend analysis which might use data compression techniques to extract pattern information) will probably be important at multiple levels in the NuMesh system, from reducing the computation requirements of placement routing and scheduling algorithms to minimizing the length of schedules stored at individual NuMesh Communication Finite State Machines.

6.2.2 Feedback Between Scheduling and Routing

In an ideal realization of this system, the backend modules allocating network resources should serve as advisors to a higher level scheduling compiler which actually analyzes programs written in a high level language. Congestion information should flow back from this analysis to the compiler. The compiler can then make effective decisions about whether or not to reorder writes and reads at communication sources and destinations.

If the compiler's initial ordering results in traffic congestion as detected by the backend and the congestion is such that the network becomes the bottleneck in an application's performance, the compiler can decide whether it is possible to reorder communication at a high level and still maintain the same computational throughput for the application while improving its communication performance.

6.2.3 Implementation of Multicast Service

Since all communication emitted from or destined for a processor is serialized at the processor/network interface, methods to broadcast data to multiple destinations and to combine data arriving at a single destination will improve system performance when they can be applied. Network bandwidth and communication latency in the current system can be improved if a service to allocate routes for messages which are *multicast* to several destinations is provided. Basically what is needed is a routine to allocate a set of routes for a statically specified multicast such that multicast data can travel along paths with maximum overlap.

6.2.4 Extensions to Support Applications Which are More Dynamic

Probably the most important set of extensions to be made to this framework will allow the analysis which takes place here for a very limited set of applications to be applied to a applications with a richer set of communication requirements. Many applications [WAD+93] require *flow controlled* communication, where message transmission times are not well defined at compile time but the order of communication is predictable. In such cases, it may be worthwhile to extract contention information from a dependency analysis of the communication. For example, if it is known at compile time from analysis of an application's communication code that message A is always transmitted during a time interval mutually exclusive of the time interval in which message B is transmitted, this information can be exploited by a router which assigns both messages to the same network link. The use of explicit barriers to establish a point of synchrony so that static routing techniques can be applied to route and schedule traffic in time intervals measured relative to such points should also be studied. The tradeoff which exists between the cost of the barrier and the performance gained by statically routing communication should be evaluated.

6.2.5 Integration With the NuMesh Simulator and Hardware

In the time frame during which this thesis was developed, it was not possible to integrate the current system with the NuMesh simulation environment. A valuable next step would be to combine the two systems so that testing of the entire environment from intermediate form to cycle by cycle simulation can be done and alternative strategies for offline routing evaluated for a wide range of applications. Testing on actual NuMesh hardware should follow.

Bibliography

[ACD+91] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[Aga91] Anant Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, Oct 1991.

[Ber] M.R.C.M. Berkelaar. Lp solve, a shareware program to solve mixed integer linear programming problems. Eindhoven University of Technology, Design Automation Section, michel@es.ele.tue.nl.

[D+89] W. J. Dally et al. The J-Machine: A fine-grain concurrent computer. In *Information Processing 89*, Elsevier, 1989.

[Dah90] E. Denning Dahl. Mapping and compiled communication on the connection machine. *IEEE*, pages 756–766, March 1990.

[DS87] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[Duj92] Rajeev Dujari. Parallel viterbi search algorithm for speech recognition. Master's thesis, MIT, 545 Technology Square, Cambridge MA 02139, 1992.

[Fet90] Michael A. Fetterman. Backend code generation with highly accurate timing models for numesh. Master's thesis, MIT, 545 Technology Square, Cambridge MA 02139, 1990.

[Fox88] Geoffrey C. Fox. *Solving Problems On Concurrent Processors: Volume I*. Prentice Hall, 1988.

[G+91] D. Gustavson et al. Scalable Coherent Interface: Logical, physical, and cache coherence specifications, January 1991. Preliminary draft, P1596 Working Group of the IEEE Microprocessor Standards Committee.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[Hil85] D.W. Hillis. *The Connection Machine*. MIT Press, Cambridge, Mass, 1985.

[HL87] N. Hasan and C.L. Liu. A force-directed global router. In *Advanced Research in VLSI*. The MIT Press, 1987.

[Hon92] Frank Honore. Redesign of a prototype numesh module. M.I.T. Bachelor's Thesis, EECS, February 1992.

[Hor71] W. A. Horn. Some simple scheduling algorithms. *Naval Journal of Operations Research*, February 1971.

[KJV83] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 200(4598):671–680, May 1983.

[KP93] Pekka A. Kauranen and John S. Pezaris. Graph embedding to the diamond lattice. In *Proceedings of the Student VLSI Conference, M.I.T.*, 1993.

[Kru83] Clyde P. Kruskal. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, Dec 1983.

[Kun82] H. T. Kung. Why systolic architectures. *IEEE Computer*, pages 37–44, January 1982.

[Kun88] H. T. Kung. Deadlock avoidance for systolic communication. In *Proc. 15th Symposium on Computer Architecture*, Honolulu, May 1988.

[Lam89] M. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, Boston, 1989.

[LMP+91] Tom Leighton, Fillia Makedon, Serge Plotkin, Clofford Stein, Eva Tardos, and Spyros Tragoudas. Fast approximation algorithms for multicommodity flow problems. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, pages 101–111, 1991.

[Lo88] Virginia Mary Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, 37(11):1384–1397, November 1988.

[LRea90] Virginia M. Lo, Sanjay Rajopadhye, and et. al. Oregami: Software tools for mapping parallel computations to parallel architectures. Technical Report CIS-TR-89-18, Dept. Of Computer Scienve, University of Oregon, Eugene, Oregon 97403-1202, 1990.

[Met91] Chris Metcalf. A NuMesh simulator. NuMesh Systems Memo 4, MIT Computer Architecture Group, April 1991.

[Nga89] John Y. Ngai. *A Framework for Adaptive Routing in Multicomputer Networks*. PhD thesis, California Institute of Technology, 1989.

[Ngu91] John Nguyen. A C interface for NuMesh. NuMesh Systems Memo 3, MIT Computer Architecture Group, February 1991.

[Opp75] Alan V. Oppenheim. *Digital Signal Processing*. Prentice Hall, 1975.

[Pez92] John S. Pezaris. Numesh cfsm revision 2: A comparative analysis of three candidate architectures. Master's thesis, MIT, 545 Technology Square, Cambridge MA 02139, 1992.

[PN93] Gill Pratt and John Nguyen. Distributed synchronous clocking. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[PSW91] C. Peterson, J. Sutton, and P. Wiley. iWarp: A 100-MOPS, LIW multiprocessor for multicomputers. *IEEE Micro*, pages 26–29, 81–87, June 1991.

[PW92] Gill Pratt and S. A. Ward. Rationally clocked computing. Internal Memo, MIT Computer Architecture Group, 1992.

[PWM+93] Gill Pratt, Stephen Ward, Chris Metcalf, John Nguyen, and John Pezaris. The diamond interconnect. In process, 1993.

[SA91] Shridhar B. Shukla and Dharma P. Agrawal. Scheduling pipelined communication in distributed memory multiprocessors for real-time applications. In *18th Annual ACM SIGARCH Computer Architecture News, Vol. 19 No. 3*, 1991.

[Sei85] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1), January 1985.

[Shu91] S. B. Shukla. *On Parallel Processing For Real-Time Artificial Vision*. PhD thesis, North Carolina State University, 1991.

[Ste90] Guy Steele. *Common Lisp The Language*. Digital Equipment Corp., 1990.

[Tho87] Clarke D. Thompson. Experimental results for a linear program global router. In *Computers and Artificial Intelligence, Vol. 6, No. 3.*, pages 229–242. 1987.

[Tro89] Sean Edwin Trowbridge. A programming environment for the numesh computer. Master's thesis, MIT, 545 Technology Square, Cambridge MA 02139, 1989.

[WAD+93] Steve Ward, Karim Abdalla, Rajeev Dujari, Michael Fetterman, Frank Honoré, Ricardo Jenez, Philippe Laffont, Ken Mackenzie, Chris Metcalf, Milan Minsky, John Nguyen, John Pezaris, Gill Pratt, and Russell Tessier. The numesh: A modular scalable communications substrate". In *The International Conference on Supercomputing*, 1993.