

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

VLSI-Based Hypermesh Interconnection Networks for Array Processing

by

Masoud Rostam Kafhesh

B.Sc., Sharif University of Technology, 1986


A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Masoud Rostam Kafhesh 1989

SIMON FRASER UNIVERSITY

July 1989

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-59355-5

Approval

Name:

Masoud Rostam Kafhesh

Degree:

Master of Science

Title of Thesis:

VLSI-Based Hypermesh Interconnection Networks for Array Processing

Examining Committee:

Dr. Warren Burton, Chairman

Dr. Rick Hobson
Director and Associate Professor
Senior Supervisor

Dr. Ramesh Krishnamurti
Assistant Professor
Supervisor

Dr. Art Liestman
Associate Professor
External Examiner

July 24, 1989

Date Approved

PAGINATION ERROR.

TEXT COMPLETE.

NATIONAL LIBRARY OF CANADA.

CANADIAN THESES SERVICE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

BIBLIOTHEQUE NATIONALE DU CANADA.

SERVICE DES THESES CANADIENNES.

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

VLSI-Based Hypermesh Interconnection Networks for Array Processing.

Author: _____

(signature)

Masoud Rostam Kafhesh

(name)

July 27, 1989

(date)

Abstract

Given the clear and pressing need for improved computer system performance, there are several means of achieving this end. In the simplest approach, current computer architectures are reimplemented using faster technologies. Although this approach will always be exploited, physical, technological, and economic limitations make it incapable of providing all the needed computational power.

Instead, *parallelism* must be exploited to obtain truly significant performance improvements. Parallelism is a two dimensional problem. Along one dimension we find pure data parallelism as might be found in typical array algorithms involving vectors and matrices. Along the other dimension we find concurrency where independent processes work on facets of an algorithm which may not lend themselves to array processing. Assuming the use of the fastest reasonable technology, any further increase in performance requires the efficient exploitation of parallelism in one form or another.

The performance of computers can be made incrementally extensible by exploiting VLSI technology to build concurrent/parallel computers, ensembles of processing nodes connected by a network. Low latency communication elements are required to support fine-grain or medium-grain parallel computation. Communication between nodes of a multicomputer need not be slower than the communication between the processor and memory of a conventional computer. A VLSI-Based network controller can provide node-to-node communication times that approach main memory access times of sequential computers. A VLSI chip is subject to several technological constraints. Whenever each node of a multicomputer system is implemented as a VLSI chip or a printed circuit board, packaging constraints limit the number of connections that can be made available for communication links. Some key issues which must be considered when designing a high performance network controller based on VLSI technology are also discussed.

New variations on the $2-d$ mesh interconnection computer which can be implemented

very efficiently using VLSI technology for chips and packaging are proposed. Direct connectivity along rows and columns reduces the diameter of an $n \times n$ 2- d mesh from $2n-2$ to 2. This technique permits the network communication bandwidth to be more balanced (or uniform) with the node processor data bus bandwidth. Simulation studies on several important permutation and matrix algorithms show that direct connectivity in hypermesh highly simplifies algorithm design and supports very efficient communication patterns.

To my parents

Acknowledgements

I would like to thank my supervisor, Dr. Rick Hobson, for his guidance throughout my research and development of this thesis and also for the financial support in the forms of research assistantships.

The VLSI lab and all its enthusiastic members provided a very inspiring work environment. John Simmons for helping me and his support deserves special mention.

Many of my colleagues in computing science were very helpful. Garnik Haftevani and Ranabir Gupta deserve many thanks.

I would also like to acknowledge the School of Computing Science and Simon Fraser University financial assistance in the forms of teaching assistantships and a Graduate Fellowship.

Table of Contents

Approval	i
Abstract	iii
Acknowledgements	v
Table of Contents	vi
1. Introduction	1
1.1. Multicomputer Networks: A Definition	2
1.2. Multicomputers Building Blocks	4
1.3. VLSI Constraints	5
1.4. Interconnection Networks	6
1.5. Multicomputer Programming	6
1.6. Thesis Objectives	8
2. Architectural Features and Design Considerations of Hypermesh Multicomputers for Array Processing	10
2.1. Issues in Designing Parallel Machines	11
2.1.1. General versus Fixed Communication:	12
2.1.2. Fine Grained versus Coarse Grained	12
2.1.3. Multiple versus Single Instruction Stream	13
2.2. The Communication Network	14
2.2.1. Choosing a Topology	14
2.2.2. Choosing a Routing Strategy and Mechanism	16
2.3. Enhancements on Mesh Structure	18
2.4. Proposed Network Structure	20
2.4.1. The Architecture	21
2.4.2. Routing Scheme in Hypermesh	22
2.4.3. Characteristics of Hypermesh	25
2.5. Characterizing the Computational Power of the Hypermesh	26
2.5.1. Hypermesh vs. Regular Mesh	26
2.5.2. Hypermesh vs. mesh with single broadcasting	27
2.5.3. Hypermesh vs. mesh with multiple broadcasting	28
2.5.4. Hypermesh versus Fully Connected Network	29
2.5.5. Divide and Conquer Strategy	29
2.6. Symmetry and Embedding	30
2.7. Applications	32
2.8. Diagonal Hypermesh	33
2.9. Sorting	34

3. VLSI Constraints and Hardware Support for Communication in Multicomputer Networks	37
3.1. Communication Paradigm and Hardware Support	38
3.1.1. Buffer Management	38
3.1.2. Flow Control	39
3.2. Communication Protocol in a VLSI-Based Multicomputer Network	39
3.3. Node Processor Considerations	41
3.3.1. A Streaming Memory Interface	43
3.4. Network Interface Considerations	43
3.5. Network Controller and I/O Embedding in the Proposed Mesh	44
3.5.1. Sample Operations	45
3.5.2. Network Input/Output	45
4. Evaluating Success and Benchmarking	49
4.1. Important Metrics of Network Performance and Properties of Hypermesh	50
4.2. Some Fundamental Permutations on Hypermeshes	51
4.2.1. A Simple Routing and message density for permutations	52
4.2.2. Exchange Permutation	53
4.2.3. Perfect Shuffle Permutation	54
4.2.4. Butterfly Permutation	55
4.2.5. Bit Reversal Permutation	55
4.2.6. Shift Permutation	57
4.2.7. Analytical Proof for 2-Step Routing on D-hypermesh	58
4.3. Environment for Multicomputer Simulation	60
4.3.1. Array Summation	62
4.3.2. Matrix Multiplication	64
4.3.3. Performance Study of Matrix Multiplication	67
4.3.4. Matrix Multiplication on Diagonal Hypermesh	69
5. Conclusions	73
Appendix A. Glossary of Acronyms	76
Appendix B. Uniprocessor Matrix Multiplication	77
Appendix C. MicroAPL Code for Matrix Multiplication on Hypermesh	79
References	84

List of Tables

Table 3-1:	A sample of network coprocessor instructions.	41
Table 4-1:	Routing complexity on hypermesh of size $(n \times n)$ with a set of indexing scheme for a variety of frequently used permutations.	58

List of Figures

Figure 1-1:	Structure of a hypercube multiprocessor.	3
Figure 1-2:	Multicomputer Node.	4
Figure 1-3:	Representative interconnection networks.	7
Figure 2-1:	Latency of Store-and-forward routing (top) vs. wormhole (bottom).	17
Figure 2-2:	The mesh with a single global mesh.	19
Figure 2-3:	Proposed bus structure.	21
Figure 2-4:	A hypermesh interconnection network (A) and companion processor plan (B).	21
Figure 2-5:	A message moving toward its destination in a hypermesh, (first step of two hop).	23
Figure 2-6:	A simple routing algorithm for hypermesh.	25
Figure 2-7:	An illustration of a mesh with single broadcasting.	27
Figure 2-8:	Mesh interconnection network with multiple broadcasting.	28
Figure 2-9:	(a) 4-d hypercube; (b) hypercube embedded in hypermesh.	31
Figure 2-10:	Diagonal hypermesh interconnection network.	33
Figure 2-11:	Some indexing Schemes, (a) row-major; (b) snake-like; (c) proximity; (d) shuffle row-major.	35
Figure 3-1:	Node processor functional components.	42
Figure 3-2:	Network controller signaling.	44
Figure 3-3:	Network input/output.	46
Figure 3-4:	A hierarchy of network controllers for I/O.	48
Figure 4-1:	Flow diagram of (A)perfect shuffle, and (B)bit reversal permutations on hypermesh.	52
Figure 4-2:	Perfect shuffle on hypermesh with snake like ordering and 2-step routing solution.	54
Figure 4-3:	Bit reversal on a hypermesh with a shuffle row ordering; and the overloaded pivots after first step of the data routing.	55
Figure 4-4:	Flow Diagram of bit reversal permutation on an 8x8 D-hypermesh.	56
Figure 4-5:	Perfect shuffle on hypermesh with proximity ordering; and 2-step routing solution.	57
Figure 4-6:	Tree reduction scheme.	63
Figure 4-7:	A typical slice algorithm for a scalar aggregation in synchronized array processing.	64
Figure 4-8:	Mapping 4x4 matrix items on 2x2 processor array.	67
Figure 4-9:	Another representation of a Diagonal Hypermesh.	69
Figure 4-10:	One step reordering in a D-hypermesh; horizontal links are elided.	71

Chapter 1

Introduction

Multicomputer networks consist of a large number of interconnected computing nodes that asynchronously cooperate via message passing to execute the tasks of parallel programs. Each network node, fabricated as a small number of VLSI chips, contains a processor, a local memory, and (optionally) a communication controller capable of routing messages without delaying the computation processor.

With the advent of fast, powerful microprocessors, a new branch of the computer industry has emerged. By using a large number of these cheap processors, each connected to a large private memory, it is possible to build a computing system with very impressive potential performance. If the processors are connected to each other so that they can exchange messages in a reasonably efficient manner and if the programmer can decompose his computation into a large system of communicating processes, such a multicomputer network can be a powerful supercomputer.

The appeal of multicomputer networks and their commercial emergence is based on their effective exploitation of VLSI technology, the availability of a high degree of "general purpose" parallelism, and moderate price [ReedFuji 87]. As Dally [Dally 87a] stated, a VLSI chip is subject to several technological constraints. VLSI systems (VLSI chips packaged together on modules and boards) are limited by wire density, not by terminal or logic density.

Multicomputer networks pose several important and challenging problems in network topology selection, communication hardware design, operating systems, fault tolerance, and algorithm design. This chapter summarizes recent results in some of these areas, with the following emphasis:

- models of interconnection networks
- VLSI constraints
- multicomputer building blocks

We begin this chapter by defining a multicomputer network. Given this definition, we examine the spectrum of interconnection networks and required building blocks for various models. The limitations of VLSI technology are also discussed. In section 1.5 multicomputer programming is considered. Finally, the thesis objectives are outlined.

1.1. Multicomputer Networks: A Definition

In much of the literature, multiprocessor systems and multicomputer systems are considered to be equivalent [Soucek 88]. However, they can be distinguished by the following considerations. According to Bell [Bell 86], the tightly coupled systems, called *multiprocessors*, have multiple processors and common or global memory. The processors and memories are connected by one or more high speed busses. Loosely coupled systems, called *multicomputers*, have local memories for each processor, although they sometimes have global memory for shared data.

A multicomputer network consists of tens or hundreds of *nodes* connected in some fixed *topology*. As Figure 1-2 shows, a multicomputer node minimally contains a microprocessor, local memory, and hardware support for internode communication. Special applications may dictate inclusion of specialized co-processors for floating-point, graphics, or secondary storage operations [ReedFuji 87].

Ideally, each node would be directly connected to all other nodes. Unfortunately, packaging constraints, hardware limitations and costs limit the number of connections. Because the node degree is limited, messages are often routed through a sequence of intermediate nodes to reach their final destination. In contrast to sequential computers and shared-memory computers which operate by sending messages between processors and memories, a *message-passing* parallel computer operates by sending messages between nodes that contain both logic and memory. As shown in Figure 1-1 message-passing

parallel computers such as the Caltech Cosmic Cube [TuaPet 85] and the Intel iPsc or Connection Machine [Hillis 85] consist of a number of processing nodes each containing both a processor and a local memory. The communication channels used for memory access are completely separated from those used for inter-processor communication.

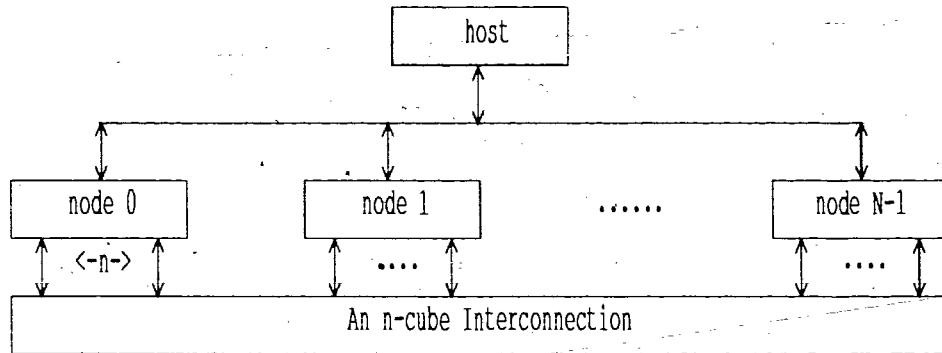


Figure 1-1: Structure of a hypercube multiprocessor.

Message-passing computers take a further step toward reducing the Von Neumann bottleneck by using a *direct*¹ network which allows locality to be exploited². A message to another process residing in a neighboring processor travels a variable distance which can be made short by appropriate process placement [Bokhari 87, LinMol 85, LeeAgg 87, Sinclair 88].

We will limit our attention to *message-passing* multicomputers. By combining a processor and memory and communication support in each node of the machine, this class of machines allows us to manipulate data locally. By using a *direct* network, message passing machines allow us to exploit locality in the communication between nodes as well.

¹point-to-point connections.

²Shared memory multicomputers (*indirect networks*) consist of a number of processors connected to a number of memories through a switch [Dally 87a].

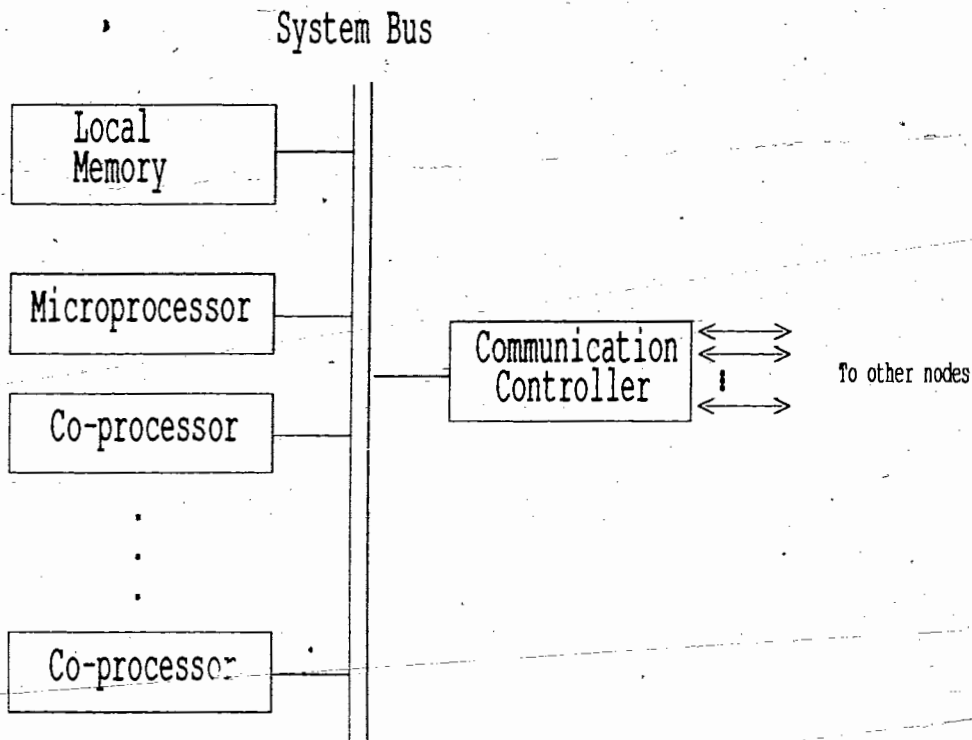


Figure 1-2: Multicomputer Node.

1.2. Multicomputers Building Blocks

The nodes of a multicomputer network each contain a processor with some locally addressable memory, a communication controller capable of routing messages without delaying the processor, and a small number of connections to other nodes.

Many realized that a universal building block would greatly simplify multicomputer network design and construction [Dally 87a, ReedFuji 87]. General purpose building blocks have been proposed, and in some cases implemented, for both computation and communication aspects of a multicomputer network node. Two such building blocks are "the Inmos Transputer" [Soucek 88] and the "Torus routing chip" [DalSei 86]. Design issues for a general purpose communication component will be discussed in Chapter 3.

1.3. VLSI Constraints

A general purpose VLSI communication component is envisioned that can be used as a building block for constructing large multicomputer networks. These components feature special purpose hardware to implement frequently used communication functions.

A VLSI chip is subject to a number of technological constraints. Several researchers have discussed these limitations [Seitz 84, Dally 87a, Dally 87b, FranDhar 86]. Violation of these constraints will result in a chip which cannot be manufactured in large quantities. Fuji [Fuji 83] describes the implications of some constraints, (*i.e* silicon area, power dissipation, and number of pins) on the design of a VLSI communication component. Closer examination of VLSI network implementation problems however show that pin limitations, rather than chip area or logical component limitation, are a major constraint in designing large networks. The number of interconnections to the chip's periphery is limited, and will increase much more slowly than the number of transistors per chip.

Whenever each node of a multicomputer system is implemented as a VLSI chip or a printed circuit board, packaging constraints limit the number of connections that can be made to the node, placing an upper bound on the I/O bandwidth available for communication link. As more links are added to each node, less bandwidth is available for each one. However, increasing the number of the links will usually reduce the average number of hops required to reach a particular destination. Therefore, a tradeoff exists between link bandwidth and average hop count as the number of links on each node is changed.

A packaging strategy based on Dense Interconnection Technology has been proposed [HobKaf 89], that can be used for efficient bottom layer of a parallel computer hierarchy.

1.4. Interconnection Networks

Interconnection networks for parallel computers have been studied intensely, and many different network topologies have been proposed [Feng 81, ReedFuji 87, ReedGrun 87]. Among the proposed interconnection networks several that serve as useful points of reference or have particularly attractive features are the single bus, the complete connection, the single ring, the chordal ring, the spanning bus hypercube, the dual-bus hypercube, the torus, generalized hypercubes, the cube-connected cycles, the R-ary N-cube, the lens, the X-tree, and B-ary tree. Figure 1-3, illustrates a subset of these networks.

The single bus network joins all nodes and uses a contention resolution protocol to resolve simultaneous requests for the bus. Although inexpensive, it can efficiently support only a modest number of nodes. In contrast, the complete connection network directly connects each node to all others. Its performance is the best achievable. These two networks, the bus and the complete connection, bound the spectrum of price and performance for all practical multicomputer networks.

1.5. Multicomputer Programming

Multicomputer networks are typically programmed using familiar sequential programming languages, augmented with message passing communication primitives [ReedFuji 87]. The application programs for multicomputer networks must be decomposed into a collection of parallel tasks that communicate using the message passing mechanisms provided by the machine. In the following, some possible approaches will be reviewed.

The Massively Parallel Processor (MPP) [Soucek 88] is programmed in a high level language Parallel Pascal [ReevGut 89]. Parallel Pascal is an extended version of the Pascal programming language which is designed for the convenient and efficient programming of MPP parallel processors. In Parallel Pascal all conventional expressions are extended to array data types.

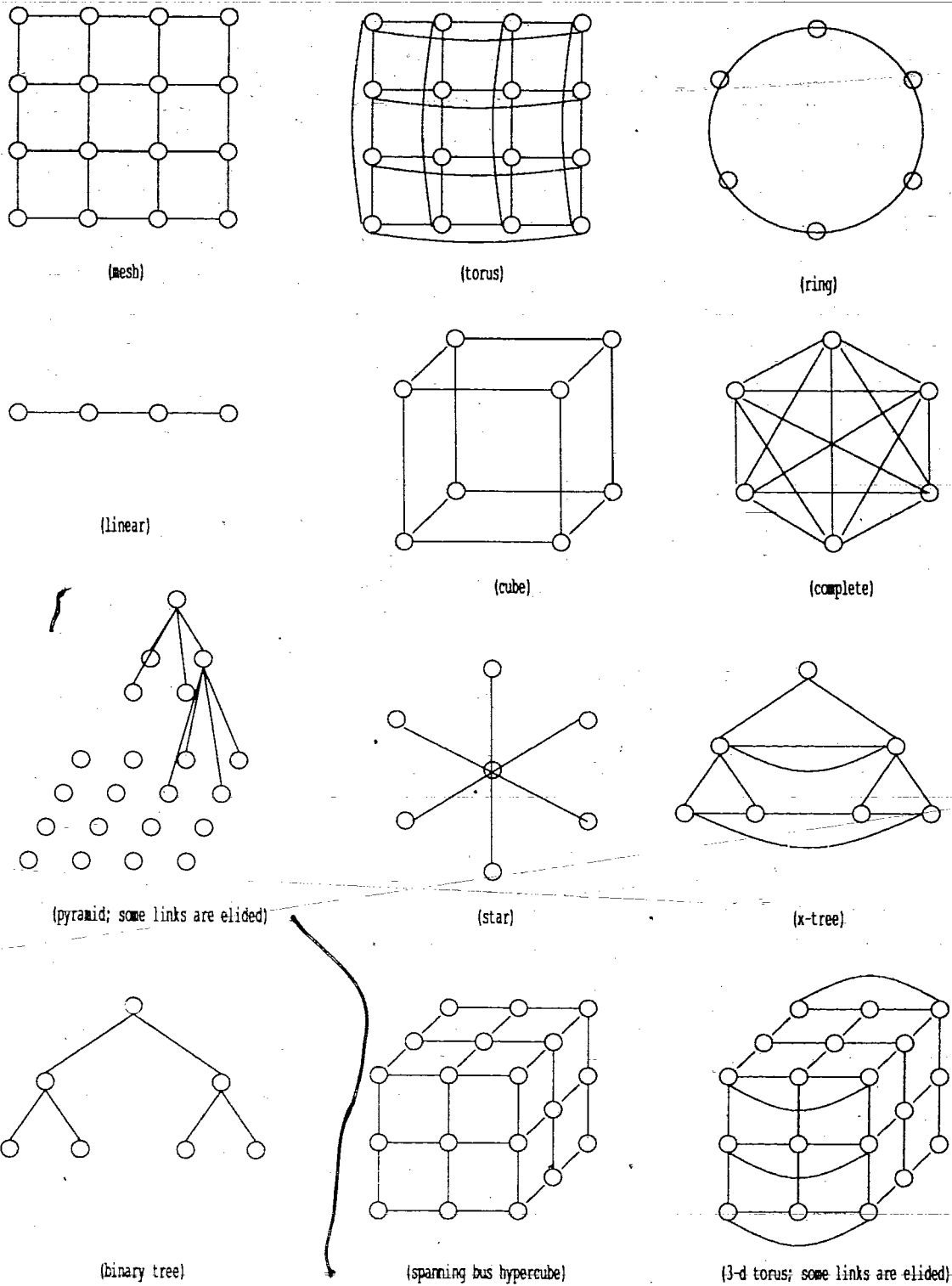


Figure 1-3: Representative interconnection networks.

Dally [Dally 87a] proposes the use of *Object-Oriented* programming approach, to program concurrent computers. The message-passing paradigm of object-oriented languages introduces a discipline into the use of the communication mechanism of message-passing computers. In an object-oriented language, computation is performed by sending messages to objects. Objects never wait for or explicitly receive messages. Instead, objects are reactive. The arrival of a message at an object triggers an *action*. The action may involve modifying the state of the object, transmitting messages that continue the control flow, and/or creating new objects. Since the actions on an object are ordered, simultaneous processing of messages is not consistent with the model of computation described above. Therefore, the concept of *distributed objects* has been proposed, which consists of a collection of all *constituent objects*, each of which can receive messages on behalf of the distributed object. Since many constituent objects can receive messages at the same time, the distributed object can process many messages simultaneously. In this thesis APL language has been used for programming a simulator of the proposed multicomputer system (Chapter 4). APL provides a syntax which is appropriate for the type of parallelism associated with array and vector processors.

1.6. Thesis Objectives

Within the context of multicomputer networks for array processing a summary of the objectives of this thesis are the following:

- Proposal of a new multicomputer network for array processing and study
 - minimum diameter and symmetry
 - routing algorithm
 - efficiency for some well known scientific computations
 - VLSI communication consideration.

- Simulation of multicomputer networks on a Unix-based Sun workstation, in APL (using C for linking multiple APL workspaces) and study
 - algorithms for some problems in array processing
 - a partial instruction set for a network controller chip.

The development of an architecture that applies VLSI technology to support parallel processing is approached in two steps. First we consider the interconnection network over which processing elements communicate. In addition to a topology a network requires a communication mechanism for routing messages. A VLSI-Based Communication component will be discussed in Chapter 3. Performance characteristics of the hypermesh and also the simulation results are presented in Chapter 4.

Chapter 2

Architectural Features and Design Considerations of Hypermesh Multicomputers for Array Processing

Parallel algorithms and the architectures used to execute them have been of great interest to computer researchers recently. This is due to the potential speedups they offer in solving important application problems. A well-studied method for interconnecting many different processors is the *mesh-connected* parallel computer, in which connections are made only to nearest neighbor processors in two dimensions.

Parallel machines such as ILLIAC IV have been built based on the mesh interconnection strategy. Also, parallel algorithms for important computational problems including sorting [NasSah 80], and linear algebra, image processing [Page 88], computational geometry [Lu 88, MilStou 89, MilStou 86, BoxMil 88], 2-d convolution [FaLiNi 89], and numerical computations [Modi 85], have been developed for the mesh-connected parallel computer. In a two-dimension mesh-connected parallel computer consisting of N processing elements³, (PE's) PE's at the extremes of the mesh are separated from each other by approximately $N^{1/2}$ intermediate PE's. For a parallel algorithm that starts with one input per PE and forms an output whose value depends on all the inputs, parallel time $O(N^{1/2})$ is required. Problems having this time requirement on the mesh include all of those listed above, and other important ones such as solving linear recurrences.

Gentleman [Gentleman 78] has conducted research into the data movement required for matrix multiplication, and for the inversion of a matrix on a lattice of interconnected processors (MCC). His analysis confirms that data movement and not arithmetic

³Throughout this thesis, $N=n^2$.

operations- is often the limiting factor in the performance of algorithms, and also that conventional complexity analyses for parallel computations commonly ignore the details of machine structure, which can often result in misleading conclusions. In brief, it suggests more attention should be paid to the hardware characteristics of a particular implementation.

Issues in designing parallel machines are described first. A survey of communication protocols in multicomputer networks are also described in section 2.3. Enhancements on the mesh structure are reviewed from the literature. In section 2.4 an architecture called the *Hypermesh* is proposed. In section 2.5 comparisons with other mesh-like organizations are described. Embeddability of some other important interconnection networks like hypercube and mesh on a hypermesh are also discussed. The complexity results of a set of algorithms [KumRag 87, Bokhari 84] which can be implemented efficiently using broadcasting features of the hypermesh are also shown. Another variations on the hypermesh structure called the *diagonal hypermesh* is proposed. Finally a quick review of existing parallel sorting techniques which may be used efficiently on the hypermesh wraps up the chapter.

2.1. Issues in Designing Parallel Machines

Three of the most important choices in designing any parallel machine are:

- general versus fixed communication
- fine versus coarse granularity
- multiple versus single instruction streams

Although each issue can be characterized by the extreme schools of thought, each offers a spectrum of choices rather than a binary decision.

2.1.1. General versus Fixed Communication:

Some portion of the computation in all parallel machines involves communication among individual elements. In some machines, such communication is allowed in only a few specific patterns defined by the hardware. For example, the processors may be arranged in a two-dimensional grid with each processor connected to four others. Proposed connection patterns for such fixed-topology machines include *rings, cubes, binary-cubes, etc.*

The alternative to a fixed topology is a general communication network that permits any processor to communicate with any other⁴. There are also many other intermediate possibilities, namely *dynamic reconfigurable systems* [Murakami 88] that can be reconfigured as either a shared-memory tightly coupled multiprocessor or a message-passing loosely coupled multiprocessor at run time, also as a hybrid of the two.

2.1.2. Fine Grained versus Coarse Grained

We first define the term *granularity*, which is used to classify parallel computers in terms of complexity and number of processors. Machines with a large number of elementary processors, each holding a small volume of data, are *fine-grained*; those with a small number of complex processors, each holding a large volume of data, are *course-grained*. In any parallel computers with multiple processing elements, there is a trade off between the number and the size of the processors. The conventional, single processor Von Neumann machine is the extreme case of this. The opposite approach achieves as much parallelism as possible by using a large number of small machines. In general, the ideal granularity of parallelism is application dependent [Brock 86].

The *fine* grained processor has the potential of being faster because of the larger degree of parallelism. But more parallelism does not necessarily mean greater speed. The individual processors in the small-grained design are necessarily, less powerful, so many small processors may be slower than one large one. For example, the Connection Machine

⁴*i.e.* through a shared memory.

[Hillis 85] and Massively Parallel Processors (MPP) [Soucek 88] are fine-grained machines.

Perhaps the most important issue here is one of programming style. Since serial processor machines are coarse grained, the technology for programming coarse grained machines is better understood.

2.1.3. Multiple versus Single Instruction Stream

A *Multiple Instruction Multiple Data* (MIMD) machine is a collection of connected autonomous computers, each capable of executing its own program. Usually an MIMD machine also includes mechanisms for synchronizing operations between processors when desired. In a *Single Instruction Multiple Data* (SIMD) machine, all processors are controlled from a single instruction stream that is broadcast to all the processing elements simultaneously. Each processor typically has the option of executing an instruction or ignoring it, depending on the processor's internal state. The correct choice depends on the application. For well-structured problems with regular patterns of control, SIMD machines have the edge, because more of the hardware is devoted to operations on the data. This is because the SIMD machine, with only one instruction stream, can share most of its control hardware among all processors. In applications in which the control flow requirements of each processing element is complex and data dependent, MIMD architectures have the advantage. The shared instruction stream in SIMD architectures can follow only one branch of the code at a time, so each possible branch must be executed in sequence, whereas the uninterested processors are idle. The result is that processors in an SIMD machine may sit idle much of the time.

The other issue in choosing between an SIMD and an MIMD architecture is one of programmability. There are arguments on both sides [Hillis 85]. There are also SIMD machines that allow varying amounts of autonomy for the individual processing element and/or small instruction streams, so basically this issue presents a spectrum of possible choices.

2.2. The Communication Network

The most difficult problem in the design of a multiprocessor network is the design of the general interconnection network through which the processors communicate. The building blocks from which the interconnection network is constructed are autonomous switching elements called *routers*. The routers are wired in some relatively sparse pattern, called the *topology* of the network. In other words, not every router is connected to every other. Processors communicate with one another through the routers, with the routers forwarding messages between processors just as the post office forwards mail from one branch to another. There are two issues in the design of such a system. One is choosing the topology for connecting the routers, and the other is choosing the algorithm for routing the messages.

2.2.1. Choosing a Topology

In choosing a topology, the goals can be divided roughly into two categories: *cost* and *performance*. On the performance side, we look for a combination of the following.

Small Diameter: The *diameter* is the maximum number of times that a message can be forwarded between routers when traveling from one processor to another. In other words, the diameter is the maximum of the minimum length path between any pair of nodes in a network. If this distance is small, then processors are likely to be able to communicate more quickly.

Uniformity: It is desirable that all pairs of processors communicate with equal ease or at least that the traffic patterns between all pairs or routes be reasonably balanced. This ensures that there are no bottlenecks. For example, in mesh connected computers, nodes are located at the corners of the network have less load in terms of the number of communication activities than other nodes in between. Intermediate nodes not only have to handle communication activities as a part of their communication patterns, but also have to take part in routing as an intermediary for their neighbors.

Extendability: It should be possible to build a network of any given size or, as a minimum, it should be possible to build an arbitrarily large version of the network.

Short Wires: If the network can be efficiently embedded in two or three-dimensional space such that all the wires are relatively short, then the physical distance between routers can be small. This means that information can propagate quickly between routers.

Redundant Paths: If there are many possible paths between each pair of processors, a partially defective network may continue to function. Also if a path is blocked because of traffic, a message can be directed along another route.

On the cost side we look for the following.

Minimum number of wires: Each physical connection costs money. Thus if the number of wires is small, the cost is likely to be small also.

Efficient layout: If the topology can be tightly and neatly packed into a small space, the packaging job becomes easier.

Simple routing algorithm: Because the routers can be locally controlled, this keeps down the cost of the routers.

Fixed degree: If each router connects to a fixed number of other routers, then one router design can serve several sizes of network.

Fit to available technology: If the network can be built easily with available components, it should be less expensive.

Notice that the wish list contains contradictions, for example, for minimum number of wires and redundant paths or for fixed degree, small diameter, and short wires. Any decision will be a compromise. Deciding which performance factors are most important is not easy. On the cost side most of the factors are difficult to measure and even more difficult to rationally trade off against one another. The fit to available technology often turns out to be one of the most important [Fuji 83].

2.2.2. Choosing a Routing Strategy and Mechanism

Along with choosing a topology for the network, we must choose an algorithm for moving information through it. This is called the *routing* algorithm. Often the performance of a parallel computer depends primarily on its data routing capability. The routing mechanism or transport mechanism provides a facility for moving data through the network. Several frequently used transport mechanisms and their distinguishing characteristics are discussed in [ReedFuji 87]. Briefly, these characteristics are:

- *Data Unit*: The indivisible unit of data transported through the network is either a variable-length message or a fixed-length packet.
- *Routing Overhead*: The overhead associated with message routing is incurred either on a hop-by-hop basis at each network node or only in the initial establishment of a circuit.
- *Bandwidth Allocation*: Bandwidth is allocated by the network either statically, e.g., when a circuit is established, or dynamically as messages are forwarded through the network.
- *Buffering Complexity*: The complexity of the buffering hardware varies with the sophistication of the chosen routing mechanism.

There are two major types of network, each one applicable to any of the above topologies [Shute 88]. However to compare them, consider the hypercube as an example. It is clear that, for an n -dimensional hypercube, the worst-case communication path involves passing the message along n edges. If it acts as a *store-and-forward network*, the source node sends its message, along with the destination address, to the neighboring node. The neighboring node, realizing that the message is not addressed to itself, sends it on to its neighbor in an appropriate direction. After n of these message-forwarding operations, the message arrives at the destination node. At each stage, the message is handled as a single unit (a packet). Once it has been passed to a neighbor, the node is free again to continue with the rest of its work, even though the message is still in transit. This is analogous to the process of sending messages through the postal system.

If it is a *circuit-switched network* instead, the analogy is with messages sent through the telephone system. The source node starts by setting up a route, first by contacting its neighbor and informing it of the address of the required destination node. Each of the

nodes along the path to the destination are set into a receptive mode, in much the same way as each of the intervening exchanges in a telephone network being configured when a telephone is dialed. Once the connection is made, the message can be transferred directly from the source node along the established path to the destination node. Just like the telephone exchange, each of the nodes in the path must maintain the connection until the source node signals that it has reached the end of its message. The primary disadvantage of this approach is the extensive bandwidth usage.

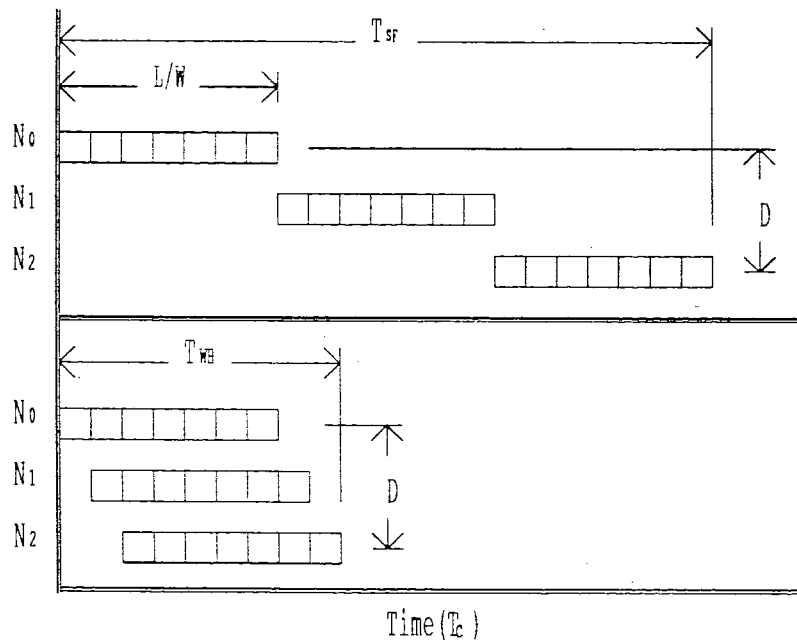


Figure 2-1: Latency of Store-and-forward routing (top) vs. wormhole (bottom).

According to [ReedFuji 87] three types of store-and-forward networks are common: *Datagram* networks are characterized by the unit of data sent through the network with variable length message. Clearly, buffer management is the primary disadvantage of this approach. In *packet-switched* transport mechanism each message is divided into fixed-sized packets that are routed separately through the network. Because packets can be relatively small, (eg. one byte) buffering requirements in each component are reduced. One of the disadvantages of the packet-switched approach is that the routing overhead

occurs on every packet rather than on every message sent into the network. It is possible to have one setup cost for one message consisting of several packets. This approach is called *virtual circuit* transport mechanism [ReedFuji 87]. A virtual-circuit is established between nodes that wish to communicate. All packets of one message sent on this circuit, travel along this path to reach their destination.

To reduce the latency of communications that traverse more than one channel, we can use *wormhole*⁵ routing rather than *store-and-forward* routing [DalSei 86]. Instead of reading an entire message into a network controller before starting transmission to the next node, the network controller forwards each *flit*⁶ of the message to the next node as soon as it arrives (Figure 2-1). Wormhole routing thus results in a message latency that is the *sum* of two terms, one of which depends on the message length, L and the other of which depends on the number of communication channels traversed, D . Store-and-forward routing gives a latency that depends on the product of L and D . Another advantage of wormhole routing is that communication does not use up the memory bandwidth of intermediate nodes.

2.3. Enhancements on Mesh Structure

Several multiprocessor architectures have been proposed for parallel processing [BeHeLa 87, Carlson 85, GoodSeq 81, Hillis 85, Hwang 89, Kale 86, LiMar 87, Page 88, Soucek 88, Stout 83, ThStSa 88, YounSing 88]. Of these, the *Mesh Connected Computers (MCCs)* have been widely used; their regular structure are particularly suitable for VLSI implementation. They seem to be a natural structure for solving many problems in matrix computation and image processing. In parallel and distributed computations the solution times of problems are constrained by information flow rather than processing times within PEs [Gentleman 78]. Moreover, even if the problem is not constrained by a large flow of information, the solution time can be constrained by the time required for moving a single

⁵This mechanism has been named *cut-through* in [ReedFuji 87].

⁶A FLOW control digIT, is the smallest unit of information that can be accepted by a communication channel or queue. One or more flits make up a message [Dally 87a].

piece of data over a long distance (as well as disturbing the other processors in between). For example, in a two-dimensional MCC with N PEs in which the PEs are placed at the grid points in a plane, moving a datum from one PE to another may take as much as $2\sqrt{N}-2$ time in the worst case.

Carlson [Carlson 85] proposes a modification to a regular mesh by adding one or more global mesh structures to the processor array (Figure 2-2). Modifying the mesh with multiple global busses can also be done (this is treated in a separate paper [JraHal: 87]). A

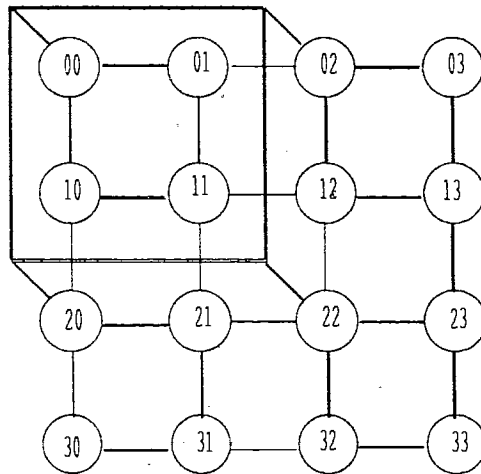


Figure 2-2: The mesh with a single global mesh.

clear disadvantage of this network is that the regularity of mesh is not maintained and some of the nodes do not have the same topology as the others. In the following, other modifications to the regular mesh will be discussed.

Given that a mesh connected computer is a natural and realistic parallel architecture for the efficient solution of many problems but solution times are constrained by long data movements, an obvious extension is to augment the network with a faster mechanism for moving data over long distances. Such a technique, called *broadcasting*, has been considered in [Gentleman 78, Stout 83]. In broadcasting, a single PE can broadcast data which are received by all the PEs simultaneously. Several such problems have been

considered [Bokhari 84, Stout 83] with substantial improvements in computation time compared to that required by MCCs without broadcasting.

2.4. Proposed Network Structure

In this section, we explore modifications of a mesh-connected parallel computer for the purpose of increasing the efficiency of executing important application programs. The modification is made by connecting each PE to all the other PE's in the same row/column. Such an extension to the 2-d mesh might be called a *Hypermesh*.

The approach taken here is similar in some ways to that of [KumRag 87]. They also propose an extension of the mesh connected computers, a mesh with multiple broadcasting, and presented several interesting algorithms running on this network quite efficiently. There are several differences between their network and the one proposed here. First, in this work, several communication features of the proposed network are not found in their network. Parallel message transfer at each node (both transmit and receive) provides a high data communication bandwidth for the entire network. The proposed bus structure (for one row/column of this network) is shown in Figure 2-3. Second, providing an insight in terms of actual design requirements and practical views in the communication support unit based on VLSI technology, which in fact is one of the main concerns of this thesis.

We will show how the hypermesh modification allows asymptotic improvements in the efficiency of executing computations having medium to high interprocessor communication requirements. Moreover, each PE can take advantage of pipelining which will be described later. We also compare our modified mesh-connected parallel computer to other similar organizations including the mesh, mesh with broadcasting, and hypercubes. We also need to select the routing algorithms carefully to avoid "traffic jams" when several messages are traveling through the network at once. These problems are discussed in detail in the next section.

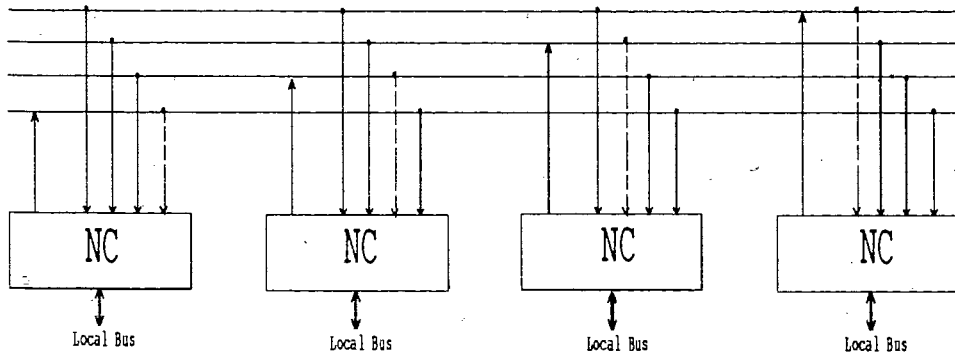


Figure 2-3: Proposed bus structure.

2.4.1. The Architecture

Figure 2-4 demonstrates the hypermesh connection pattern. Each row and column link has the structure shown in figure 2-3. The system consists of N processing nodes. Each node consists of a processor, a local memory and memory controller chip, and a communication controller. Each node has a separate communication processor to allow uninterrupted application processing.

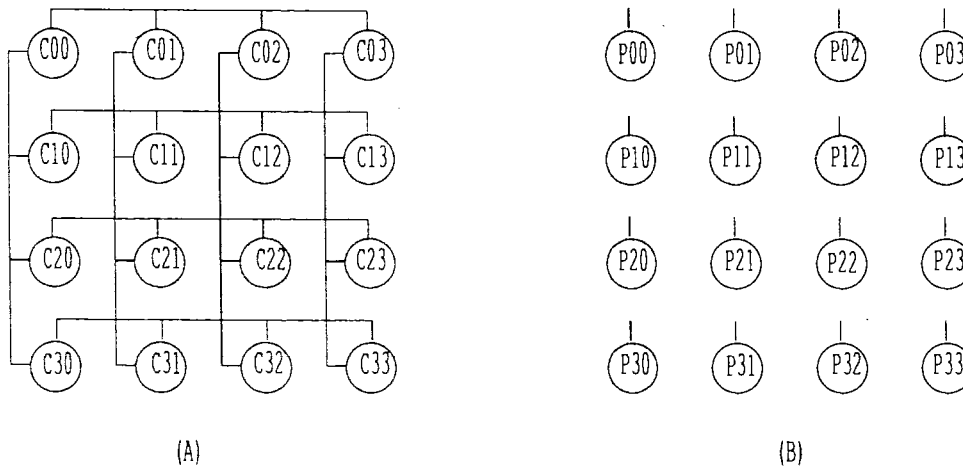


Figure 2-4: A hypermesh interconnection network (A) and companion processor plan (B).

We imagine our structure is composed of two separated $n \times n$ layers of PE 's: One layer is dedicated to processing, while the other is entirely dedicated to communication. Each processor PE in the processing layer connects to a corresponding distinct router PE in the communication layer. A processor uses the communication layer for efficiently routing data to other processors. The interconnections among the routers will determine the communication characteristics for this approach. This topology provides direct connectivity for each PE to all the other nodes along its row and column. Processors are identified by their two dimensional coordinates. Communication components are numbered similarly.

2.4.2. Routing Scheme in Hypermesh

All communication networks require some routing algorithm to build the paths between communicating nodes. A great deal of research has been done in the area of routing in multicomputer networks. In the context of the proposed communication domain, we will only consider *distributed routing* that does not rely on central authority. In regular networks (eg. mesh, torus, hypercube, etc.) routing can be performed in each node by a state machine or microprogrammed engine using a fixed algorithm based on the local and destination addresses.⁷ Routing algorithms are known for many standard topologies. In a square lattice, for example, the routing controller could forward the message in a direction that would reduce the difference between the X- or Y-coordinates of the current and the destination nodes. In the n-cube, links are selected which reduce the Hamming distance by one, until the target is reached.

⁷For irregular networks, routing must be based on suitable lookup tables [ReedFuji '87]. In such a system each node i has entries of the form:

$$NN = R_i(DN)$$

implying that messages destined for node DN are forwarded by node i to neighbor node NN . This table lookup, commonly called a **routing table**, can be defined statically, or it can be maintained dynamically using information exchanged between neighboring nodes. This technique could be of use even in hypermesh. Hierarchical techniques can be used to implement general lookup table mechanisms for message routing without excessively large memories. Memory size is minimized when many levels are used. •

In the following, we will show that many data routing functions which the mesh cannot perform well can be achieved by Hypermesh very efficiently. One way to evaluate the routing capability of an interconnection network is the communication time between any 2 processors. The communication for $node(i_1, j_1)$ to $node(i_2, j_2)$ requires $|i_1 - i_2| + |j_1 - j_2|$ steps on a mesh, which is $2(N^{1/2} - 1)$ in the worst case. By using Hypermesh, this can always be achieved in two steps. First $node(i_1, j_1)$ sends to $node(i_1, j_2)$, then $node(i_2, j_2)$ receives back from $node(i_1, j_2)$ ⁸. Figure 2-5 illustrates this process.

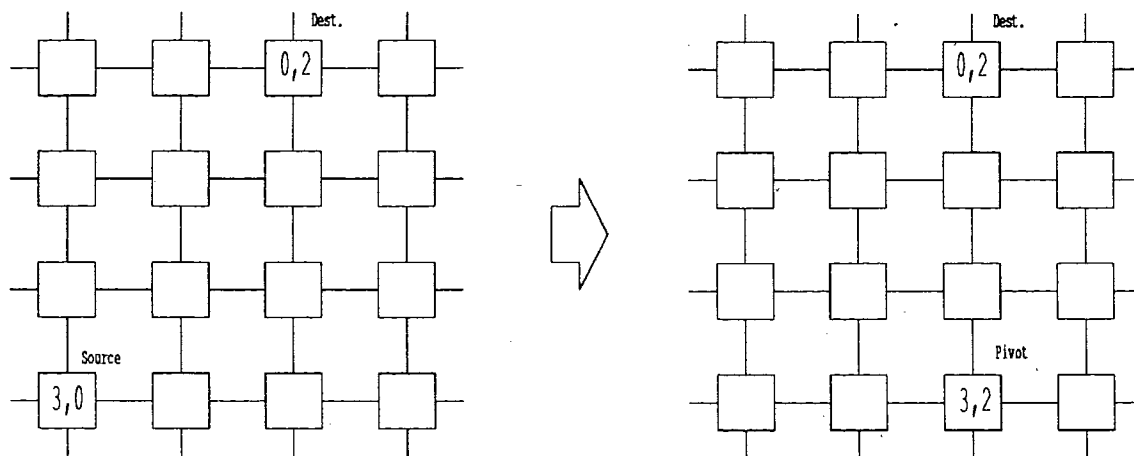


Figure 2-5: A message moving toward its destination⁹ in a hypermesh, (first step of two hop).

Row/column broadcasting can be performed as following: $node(i, j)$ sends a message to $node(i, *)$ ¹⁰ or $node(*, j)$ s. This can be done by first injecting message by $PE(i, j)$ to $Router(i, j)$, then through a handshaking mechanism, all the $Routers$ in the same row/column read it back simultaneously. Generally, broadcasting on hypermesh can be defined as following:

⁸ $node(i_2, j_1)$ acts as a pivot and forwards the received message.

⁹To achieve a bidirectional transaction flow, the proposed routing algorithm prevents congestion for the pivoting node and creating a cylinder effect; (all transaction flow occurs in a counterclockwise direction).

¹⁰ $A(i, *)$ and $A(*, j)$ denote all the nodes in the i^{th} row and j^{th} column of A, respectively.

$$PE\{k_i | k_i \in \{1, 2, \dots, N\}\} \longrightarrow PEs \subset \{1, 2, \dots, N\}$$

$i = 1$ single broadcasting
 $i > 1$ multiple broadcasting¹¹

Row and column broadcasting is a powerful communication mechanism. Suppose only PEs in a particular row, say row 0, each have a data item and we wish to compute, for example, the maximum of these numbers; then, we can use multiple broadcast busses to simulate a tree structure.

Performing permutations of data on $SIMD$ computers efficiently is important for high speed execution of parallel algorithms. For an efficient execution of parallel algorithms on $SIMD$ computers, an important objective is the fast rearrangement of intermediate results. The total execution time greatly depends on the time required to perform permutations of data. The classes of permutation usually considered are the permutations strongly suggested by the communication needs imposed by the existing parallel algorithms, and by the data storage schemes.

Simple algorithms for performing important permutations can be achieved for our proposed network. Many permutations can be done in a constant number of steps (Chapter 4), as opposed to $2(\sqrt{N}-1)$ the lower bound in mesh with wrap-around [RagKum 84], and $3(\sqrt{N}-1)$ in regular mesh [LinMol 85]. This approach is quite simple, and unlike previous approaches, makes efficient use of the special topology of the proposed network to realize these permutations using the minimum number of data transfer steps. Here, a very simple control algorithm on the hypermesh network is proposed. The control algorithm is actually based on a very simple idea. Assume permutation P maps $node(i,j)$ to $node(r,s)$. The routing algorithm would be the following:

This can be done concurrently for each pair of the permutation P . Applying this control

¹¹ An extreme case of multiple broadcasting called *flooding* [FriBa 87]. This data movement operation is used to achieve the all-to-all broadcast needed in some operations. Flooding is performed by broadcasting along rows (for all PEs) which leaves n items at each node. Then similar broadcasting operation along columns, that is n -step routing for all n items, results an $n+1$ routing steps. It is interesting to note that in a mesh with multiple broadcasting $O(n^2)$ (in fact n^2+n) steps are required for this operation.

```
if ( i=r or j=s ) then
    node(i,j) --> node(r,s)
else
    node(i,j) --> node(i,s)
    node(i,s) --> node(r,s)
fi

/* node(i,s) is called a "pivot" */
```

Figure 2-6: A simple routing algorithm for hypermesh.

algorithm on the hypermesh network, it turns out that many frequently used permutations can be realized with a constant number of passes through the network. This will be discussed, in detail, in Chapter 4.

2.4.3. Characteristics of Hypermesh

Some characteristics of the proposed mesh are:

- medium number of processors- the proposed architecture is shown in Figure 2-4. It contains 16 nodes. The readily available technology permits a single controller chip to serve both rows and columns of a 4 by 4 mesh.
- asynchronous execution- each node executes independently of all other nodes. Synchronization between nodes relies on message passing or instruction fetching primitives.
- message based communication- because it contains no shared memory, the cooperating task of a parallel algorithm relies solely on message passing. The message is, in fact, a raw fixed-length packet of data which contains a fixed number of flits.
- low communication overhead- hardware support in terms of a communication chip with a high bandwidth provides an efficient communication environment. Furthermore, by using the network to hold intermediate results, node processors can feed array data directly to arithmetic units rather than first moving them to local memory.
- small diameter- direct connectivity along rows and columns reduces the diameter of a n by n 2-dimensional mesh from $2n-2$ to 2. This technique permits the network communication bandwidth to be more closely matched to node processor data bus bandwidth.
- medium grained computation- which provides a well-balanced communication, computation over such a network.

The above mentioned properties and characteristics provide the following features:

- *efficient communication patterns, simplified algorithm design*- both direct connectivity and broadcasting simplify algorithm design and support very efficient communication patterns, which in fact enables one to efficiently emulate other important network topologies (*i.e.*, mesh and hypercube).
- *high throughput*- multiple broadcasting (wide bandwidth) supported by a communication controller chip provides high network capacity and in turn high throughput. These characteristics of the hypermesh will be analyzed in Chapter 4.

2.5. Characterizing the Computational Power of the Hypermesh

To explore the power of the modifications proposed here, we look at a fairly wide range of computational problems and show they can be solved algorithmically on a hypermesh-connected parallel computer. Our emphasis is more towards exhibiting the advantages of our new parallel organization rather than on the actual algorithms. The algorithms themselves are similar to previously known parallel algorithms for the problems considered, and thus shouldn't be thought of as the major contribution of this work. Another topic treated in this section is how our hypermesh compares to some other parallel computer organizations. The performance of hypermesh is studied by comparing it to other mesh related networks as follows.

2.5.1. Hypermesh vs. Regular Mesh

For problems requiring information transfer between remote nodes (not neighbors), hypermesh is much better than regular mesh as can be seen below.

- Communication between any two processors : This requires $2(N^{1/2}-1)$ steps on mesh of N nodes [NasSah 80], but just 2 steps on hypermesh.
- Broadcasting: On mesh, this requires $O(N^{1/2})$ step, but on hypermesh 2 steps are sufficient.
- Permutation: Hypermesh can perform permutations easier and faster. The lower bound for any permutation on regular mesh is $3(N^{1/2}-1)$ steps [LinMol 85].

On hypermesh, many permutations can be done in a constant number of steps. Moreover, the system overhead is different, although the upper bound for any arbitrary permutation is

\sqrt{N} . For mesh, performing different permutations may require different routing algorithms. In the case of hypermesh, our routing algorithm proposed in the previous section is universal, (i.e. independent of permutations).

2.5.2. Hypermesh vs. mesh with single broadcasting

Gentleman [Gentleman 78] was apparently the first to consider a supplemental mechanism called *broadcasting*. When a node broadcasts a value, it is simultaneously received by all other nodes (Figure 2-7). To avoid pandemonium, only one broadcast at a

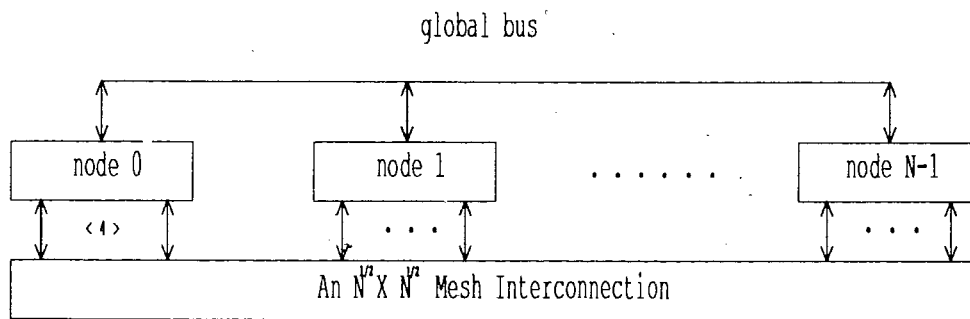


Figure 2-7: An illustration of a mesh with single broadcasting.

time is allowed. It is easy to see that a hypermesh can simulate a mesh with single broadcasting. For almost all classes of algorithms, hypermesh performs better. In fact, single broadcasting imposes a kind of sequentiality to the network [Bokhari 84]: since broadcasting is done over a shared global bus only one item can be communicated over the bus at any time. Thus, trying to cover many "long" distances using broadcasts will increase the solution time.

2.5.3. Hypermesh vs. mesh with multiple broadcasting

In contrast with the mesh with multiple broadcasting, where the broadcasting bus is superimposed on conventional node-to-node links (Fig. 2-8), our proposed network can act both as a neighbor to neighbor medium and a global broadcast medium. In fact mesh with multiple broadcasting is similar to the proposed hypermesh structure, and the available algorithms for the mesh with multiple broadcasting can be simulated on the hypermesh and in some cases with improvements in efficiency.

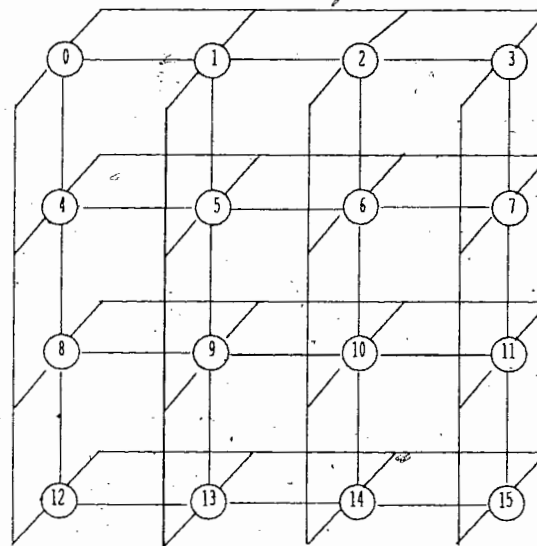


Figure 2-8: Mesh interconnection network with multiple broadcasting.

The comparison between the two architectures is done in terms of permutations. A mesh with multiple broadcasting feature does not perform better than a regular mesh in terms of permutations. In general $3(N^{1/2}-1)$ steps are required to perform permutation on mesh with multiple broadcasting. Also, the control overhead is large and the routing algorithms are complex [RagKum 84]. The upper bound for an arbitrary permutation on hypermesh is exactly $N^{1/2}$ (Chapter 4). However, for a wide class of permutations a constant number of steps are required. Also the overhead is low.

2.5.4. Hypermesh versus Fully Connected Network

The fully connected network is formed by placing a single link between every pair of nodes (Figure 1-3). Here, the number of nodes is equal to the number of tasks required by the parallel program and varies from application to application. This topology minimizes the number of hops between every pair of nodes at the expense of a larger number of branches to each node (n^2 vs. $2n$).

In some applications, a single node needs to send a data item to all other nodes in the network. If no broadcast mechanism is provided, the node must send a separate copy of the message to each destination. A queue rapidly develops in the node sending the message leading to long delays and poor performance. Therefore a significant performance improvement results from incorporating a broadcast mechanism. In a fully connected network, in practice, $O(\log N)$ time is taken to perform broadcast operations¹².

2.5.5. Divide and Conquer Strategy

The essence of the *divide-and-conquer* strategy is quite simple:

To solve a large instance of a problem, break it into smaller instances of the same problem, and use the solutions of these to solve the original problem.

The fact that the smaller problems are instances of the same problem is what distinguishes divide-and-conquer from the more general *top-down* strategy. This strategy is strongly encouraged in texts on data structures and algorithms. As Ullman notes, it is also a useful strategy in hardware design [Ullman 84].

When using parallel computers, there are several reasons why a divide-and-conquer approach may be particularly useful. First, there may be more data than can be obtained in the processors at one time, so the data must be analyzed piecemeal [Stout 87]. Second, in

¹²That's simply because of the required fanout.

some machines the individual processors may be quite large and powerful, holding many data items, in which case often a two-level strategy is needed. We can call such machines *medium-grained* machines, to distinguish them from *fine-grained* machines. In medium-grained machines, we can exploit the pipelining features of functional units more efficiently. Finally, many of the interconnection schemes being used or suggested for massively parallel processing (*i.e.* image processing) naturally suggest partitioning the machine into smaller submachines. Meshes (including hypermesh) can easily be partitioned into quadrants, each of which is a mesh. Meshes with broadcasting capabilities almost force one to use a divide-and-conquer approach, dividing the network into subnets in which a standard nonbroadcasting algorithm is used, with broadcasting used to combine results of the subproblems.

In networks of higher dimensions, using the hypermesh structure as a building block, the features of divide-and-conquer can be exploited more efficiently. In such a hypermesh hierarchy, the given problem can be decomposed into small pieces (the size of hypermesh) and distributed among the network components.

2.6. Symmetry and Embedding

Symmetric graphs, such as the ring, the n -dimensional hypercube, and the cube-connected cycles, have been widely used as processor/ communication interconnection networks. A special class of networks, called *symmetric interconnection networks*, has the property that the network viewed from any vertex of the network looks the same [Akers 89]. It is interesting to note that the hypermesh interconnection network is also a symmetric network. In such a network, congestion problems are minimized since the load will be distributed uniformly through all the vertices. Moreover, this symmetry allows for identical processors at every vertex with identical routing algorithms. It is also very useful in designing algorithms that exploit the structure of the network.

The hypermesh interconnection network, in addition to its symmetry properties, has the advantage of enabling many other interconnection networks to be embedded into itself.

The necessity for embedding arises when a programmer wishes to implement an algorithm A for which it is clear that a certain network G is most appropriate, but only another network H is available. The Fast Fourier Transformation algorithm, for example, ideally requires the perfect-shuffle interconnection, but the programmer may nevertheless wish to implement it on machines based on the lattice or the hypercube. The problem can be solved by utilizing part of H as a model of G in other words *embedding* G in H [Modi 85].

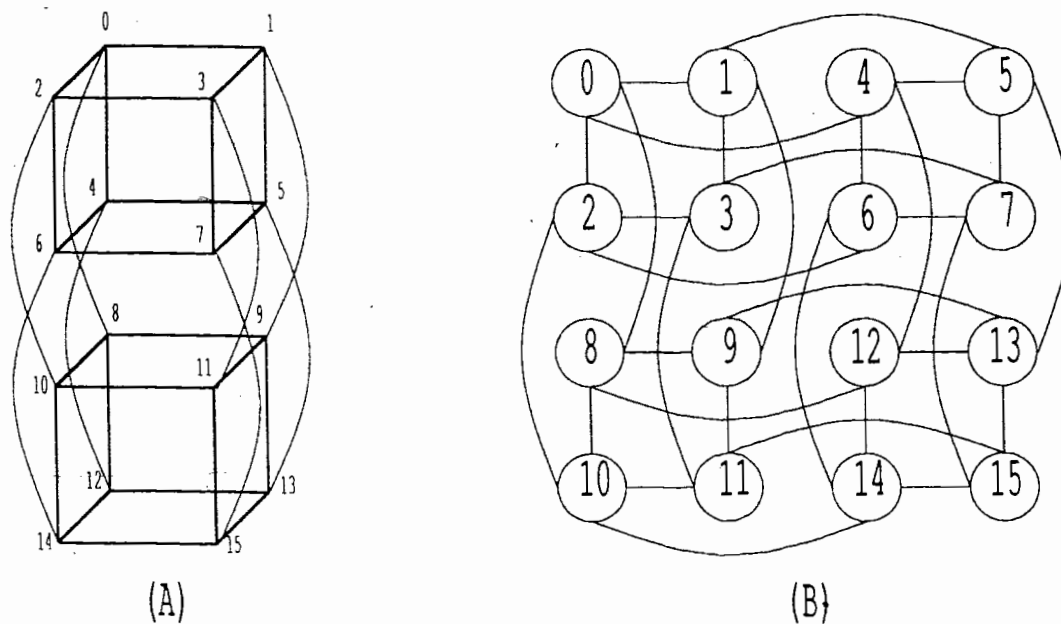


Figure 2-9: (a) 4-d hypercube; (b) hypercube embedded in hypermesh.

In order to minimize data movement cost, the indirect links (caused by mapping) should be short or avoided. In this respect the best embedding would be a *direct* one, which provides no extra data movement requirements on new network. One of the skills in parallel computing is to find a convenient embedding, involving a minimum use of long links of one network in another. This has been an interesting problem in parallel programming for a long time, and some researchers have tried to solve it. But finally it turned out to be an *NP* complete problem [NiKing 87, Berman 85].

Since hypermesh contains a mesh structure as a subgraph, the advantage of mesh is kept.

On the other hand, hypermesh is isomorphic to *Hypercube* network. As the Figure 2-9 indicates, hypermesh also contains hypercube structure as a subgraph, so the advantages of hypercube and its routing scheme are kept as well. Therefore, hypercube can be embedded in hypermesh directly. This ability has some direct advantages, in a sense that there are many applications and algorithms designed for hypercube, which can be executed on hypermesh with no extra effort, or possibly can be enhanced.

2.7. Applications

The hypermesh is an architecture with a variety of data routing capabilities. So, its potential is promising. Since the regular mesh and cube can be directly mapped into the hypermesh, any parallel algorithms designed for the mesh or cube can be adopted by hypermesh. However, if global communications are required, hypermesh is a much better architecture. In the following, we will discuss some application examples. Other applications can be probed by using the technique described here.

Semigroup computations, for example, include finding maximum, minimum, sum, etc. on N data items. On a regular *MCC*, a semigroup operation can be performed in $O(N^{1/2})$ time which is optimal. If a single broadcast bus is available then the time can be improved to $\Theta(N^{1/3})$ [Bokhari 84]. An algorithm for semigroup operations taking $O(N^{1/6})$ time in 2-dim *MCC* with multiple broadcasting has been proposed in [KumRag 87]. Since multiple broadcasting can be performed with hypermesh with no overhead, every step of that algorithm can be simulated here.

There are other parallel algorithms developed for many problems in the areas of linear algebra, image processing, computational geometry, and numerical computations, on *MCC* with multiple broadcasting, which can all be executed efficiently on hypermesh. An algorithm is given [KumRag 87] that finds the median value of N numbers distributed one per processor, in $O(N^{1/6}(\log N)^{2/3})$ time. Another set of algorithms on convex polygon computation of digitized pictures with complexity of $O(N^{1/6})$ time, and nearest neighbor in $O(N^{1/6})$ time, are proposed, which requires $\Omega(N^{1/3})$ on 2-*MCC* with single broadcast and

$\Omega(N^{1/2})$ on regular mesh. All of these algorithms can be efficiently simulated on proposed hypermesh with no degradation and in some cases with improvement in efficiency. A further reduction in the execution time is possible (for some communication patterns) by taking advantage of extra communication links and also broadcasting feature in a hypermesh. This aspect needs more investigation in the future.

2.8. Diagonal Hypermesh

Another variation on the hypermesh interconnection network is depicted in Figure 2-10.

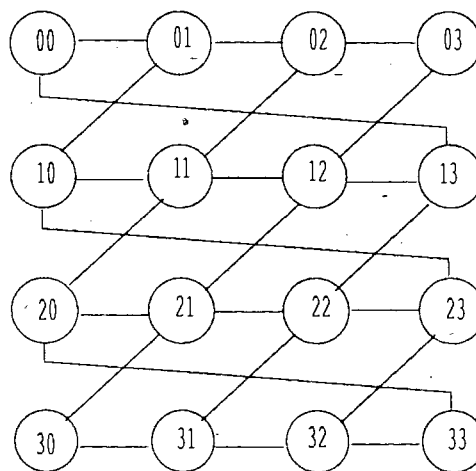


Figure 2-10: Diagonal hypermesh interconnection network.

In this network, the same direct connectivity along rows (just like in hypermesh) is maintained, but the column links are now replaced with diagonal links. This network reminds us of the *1-skewed storage technique*¹³ in memory systems [Lawrie 75, HarJum 87]. This topology turns out to have very interesting properties in terms of permutations. It can also be configured like the original hypermesh in just one simple communication step (along rows), maintaining all the features of the hypermesh.

¹³A skewing scheme is a method for assigning the elements of a vector to parallel memory modules. This technique is used to obtain conflict-free vector accesses for a subset of access patterns.

Studies on several important permutation functions on this topology showed that this topology can realize the most important class of permutations (*i.e* perfect shuffle, bit reversal, butterfly and exchange permutations,...) in at most 2 communication steps, regardless of the network size. An APL program has been implemented to verify this property, using the same data routing algorithm described for hypermesh. Experimental results are summarized in Chapter 4, for different sizes of the network.

2.9. Sorting

Sorting problem was not a research topic in this thesis. Nevertheless, reviewing of the possible techniques which possibly can be of use on hypermesh, will be presented. Extensive research into sorting techniques has been carried out during the last few years, and a large volume of literature is available [Akl 85, SchSen 89, NasSah 79, ThoKun 77, Thompson 83, SchSha 86, Han 85, SonKin 88, MarGaf 85] which provides an introduction to parallel sorting methods. This problem involves routing of each data item to a distinct position of the array predetermined by some indexing schemes. Three different schemes have been considered by Thompson and Kung [ThoKun 77]: row major, shuffled row major, and snake-like row major. Some of the standard indexing schemes are illustrated in Figure 2-11. Much of the attention has focused on restructuring well-known serial techniques such as quick sort, odd-even transposition sort [ThoKun 77], and bitonic sort [NasSah 79] in order to make them amenable to parallelism.

A serial algorithm based on, for example comparison-interchange, necessarily requires at least $O(k \log k)$ comparisons to sort k numbers. If k comparisons are carried out simultaneously at each stage, then clearly the lower bound on the number of parallel comparisons (or delays) is $O(\log k)$. However, it does not seem possible to achieve this lower bound by restructuring one of the well-known $O(k \log k)$ serial algorithms, (for example, the two-way merge sort), primarily because of lack of parallelism toward the end of the sorting process. On the other hand, it is possible by using odd-even transposition sort, using $O(N)$ processors to sort N numbers in $O(N)$ steps.

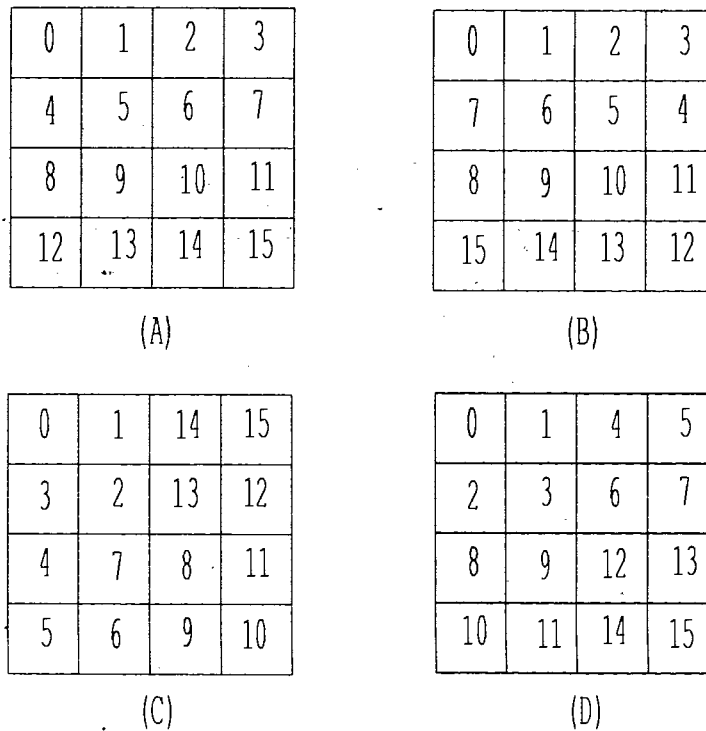


Figure 2-11: Some indexing Schemes,
(a) row-major; (b) snake-like;
(c) proximity; (d) shuffle row-major.

In the realm of sorting a two-dimensional array of numbers, a seemingly "nice" way would be to sort rows and columns (since it involves sorting on smaller problems of approximately \sqrt{N} size) and "hope" that somehow a combination of these two operations will terminate in a sorted sequence. Unfortunately, such a procedure doesn't seem to work when implementing in a straight-forward manner (row major ordering) [Leighton 85]. Paradoxically things fall into place when one sorts the rows in a *snake-like row-major* form without increasing the complexity of the procedure. A simple algorithm, called *shear-sort* has been introduced using this scheme in [SchSha 86]. It is worth noting that in a hypermesh we can get the snake-like ordering from the regular row major ordering in just one routing step (Chapter 4), rather than $\Theta(n)$ on a regular mesh. Since sorting along

row/column can be performed in $O(n)$ on an $n \times n$ hypermesh¹⁴, and also $\log n$ steps are required for this algorithm to converge [SchSha 86], therefore, the total complexity of the algorithm exploring this scheme achieves a bound within $O(n)$ of the optimal.

¹⁴Since flooding operation can be performed in linear number of communication steps on a row of hypermesh, an *enumeration sort* described in [YaTaYa 82] can be efficiently used to sort the row of the network in linear time.

Chapter 3

VLSI Constraints and Hardware Support for Communication in Multicomputer Networks

This chapter considers various physical constraints which influence the design of VLSI based interconnection networks used in multicomputer systems. Design expressions are presented for implementing a network controller for a mesh with direct connectivity along rows and columns.

The design of effective multiprocessor systems involves numerous interacting elements ranging from parallel algorithms to programming languages to computer architectures. This section focuses on the computer architecture question and, in particular, on the design of VLSI based electrical interconnection networks for use in multiprocessor systems. Due to their potentially critical effect on overall multiprocessor performance, interconnection networks have been widely studied. Various studies have focused on their functional properties (permutation, control algorithms), their complexity and performance, and their actual design.

In the following, some issues which must be considered when designing a high performance network controller (NC) based on VLSI technology is discussed. A set of useful NC instructions will also be proposed. In sections 3.3 and 3.4 considerations on a node processor and network interface chip along with an equation demonstrating the number of required pins for a typical NC in a hypermesh network will be presented. Finally, requirements for high bandwidth I/O subsystems will be discussed and a hierarchical solution will be proposed.

3.1. Communication Paradigm and Hardware Support

Communication functions have traditionally been implemented by software in loosely coupled communication networks. Workloads for such systems are generated by processes that communicate infrequently to perform high level functions such as file transfers [ReedFuji 87]. In contrast, multicomputer networks execute a collection of *closely coupled* tasks that communicate frequently. Therefore, although many of the same problems and issues that arise in loosely coupled networks also arise in multicomputer networks, the latter often require completely different solutions and implementation. In particular, rather than software implementation of communication protocols, hardware support is more appropriate.

Several key issues must be considered when designing a high performance communication controller: The routing issue has been discussed in a previous chapter, therefore in the following other issues are considered.

3.1.1. Buffer Management

Each message passed into the communication domain must be subdivided by the sender into some number of fixed length packets (flits). Packets form the indivisible unit of data transmitted through the communication network. Due to conflicts that arise when several packets simultaneously require the use of the same link, buffering is required in each node. The strategy for managing usage of these buffers can have a significant effect on performance. One simple solution gives each channel on each link a separate buffer. Allowing several channels to share buffers, is another approach. However, the control logic would be more complex. The first approach takes advantage of regularity in terms of VLSI design aspects. In order to support several activities simultaneously on different channels the first approach seems to be reasonable. Under these circumstances, better performance is obtained by having a separate buffer for each channel.

Now, what happens when a packet arrives at a pivot? It is placed at the end of the linked list corresponding to the output channel on which the packet is to be forwarded. It is

removed from the list after it has been successfully transmitted to the next node. The linked lists are managed as a FIFO queue to ensure that packets are forwarded in the same order in which they arrive. This queue management can be implemented in hardware so that packet forwarding can proceed as quickly as possible [ChuLeu 86]. Each link has an associated FIFO buffer that temporarily stores message packets.

3.1.2. Flow Control

Flow control is the mechanism that regulates the transaction of messages along circuits. The network must be able to "throttle" traffic on communication lines to prevent buffer overflow and handle other situations of that kind. There are 2 approaches to the flow control problem, *remote buffer management* and *send/acknowledge protocol* [ReedFuji 87]. The latter approach provides faster communication mechanism, which will be discussed here. A simple send/acknowledge protocol for data transmission over the link is the most straightforward example of *receiver controlled flow control*. Each node sends a packet and waits for the receiver (or receivers) to return a control signal (ack). It is assumed that each link has a separate control line to carry the ack signal. Because the receiver can generate an acknowledgement after only the header (first flit) is received, a direct connection to the sender (or receiver) offers the unusual feature that the sender will receive acknowledgement before it has finished sending the packet. This allows a "pipelined" stream of flits through the links. Flow control is built into our slice algorithms (Chapter 4), *i.e.* we produce and consume data at exactly the proper rate.

3.2. Communication Protocol in a VLSI-Based Multicomputer Network

A general purpose VLSI *communication component* is envisioned that can be used as a building block for constructing large multicomputer networks. These components feature special purpose hardware to implement frequently used communication functions. Each router handles messages for one PE, allowing it to communicate with all other PE's in its same row/column. A typical communication network of the hypermesh is formed by 16 routers connected by unidirectional wires (Figure 2-4). The routers are wired in the pattern

of the hypermesh. The address of the routers within the network depends on their relative position within the mesh. Networks with more than 16 nodes would require a larger router (i.e., 2 for 8x8 mesh). The operations of the router can be divided into following categories [ReedFuji 87]:

injection, delivery, buffering, forwarding.

The *injection* process involves simple handshaking between processor and router. The process by which a router removes a message from the network and sends it to the node for which it is destined is called *forwarding*. When a message finally reaches its destination router, it is *delivered* to the appropriate processor by writing into the processor's memory (register), which involve a simple handshaking between the processor and router. Clearly the router is hardware limited to a fixed buffer capacity. The number of buffers is large enough so that the router almost never runs short of storage, but an additional mechanism could be provided for dealing with the overflow case should it occur. This mechanism uses two FIFO queues at both sides of the communication link. Between the processor and router is a pair of first-in/first-out buffers (FIFOs) that buffer bytes going to the router and data returning to the processor. These buffers allow the router to operate asynchronously with the processor.

To support array algorithms, the network data interface should permit some automatic sequencing from one processor port to another. Some of the coprocessor strategy which was used for SJMC [HobSim 87] can be used for a Network Coprocessor. These coprocessors receive source and destination instructions over the system bus, 1 cycle in advance of when they are needed. The system bus is thus used for one data transfer and one instruction transfer during each cycle. For this reason 2 network data transfers can occur in 1 system cycle.

An instruction set for Network Coprocessor will evolve as a variety of array algorithms are studied. A small collection of useful NC instructions are outlined in Table 3-1. These primitives are executed by the NC firmware in accordance with instructions submitted to it

by the node processor. The list of proposed primitives cover the functionalities which are required for programming higher-level communication and synchronization protocols. They have been designed in order to keep the NC simple and fast and, on the other hand, to provide the higher-level network modules and the applications in the hosts with a powerful set of communication instructions.

-
- NRBC:** Network row broadcast initialize. Subsequent data transfer will go to all row processors.
 - NCBC:** Network column broadcast initialize. Subsequent data transfer will go to all column processors.
 - NRWA:** Network read word alternate. The number of words to read before changing from row to column is provided on the data bus as a parameter.
 - NRRW:** Network read row word. Initialize for automatically cycling through the row ports. The number of words to read before changing from one row processor to another is provided on the data bus as a parameter. This parameter also selects which processor to start with.
 - NRCW:** Network read column word. See NRRW.
 - NSN:** Network source next. Data are place on the system bus and the next source of data is readied. This may be the same processor, the next row processor, or the next column processor.
 - NDN:** Network destination next. Data are sent from the system bus into the network and the next destination is readied. This may be the same processor, the next row processor, or the next column processor.

Table 3-1: A sample of network coprocessor instructions.

3.3. Node Processor Considerations

A central requirement for efficient array processing is to match data transfer bandwidth with arithmetic processing bandwidth. If the Arithmetic Processor (AP) is pipelined, the data links to memory or a network must also be pipelined.

State-of-the-art floating-point AP's are available with cycle times ranging from below 50ns up to 250ns depending upon the technology and the amount of pipelining. Data ports can be either 32 bits or 64 bits wide. Vector data registers and highly interleaved memory

banks are required to keep such chips maximally busy [HobKaf 89]. When networking is brought in, it is unlikely that data paths in the network itself will be as wide as 32 or 64 bits. If we are going to strive for maximum connectivity (the largest number of direct connections), wide data paths are only practical for a small number of processors, say a 2x2 hypermesh. For the following discussion, we choose 8-bit-wide network data paths. This number has been chosen only because it will permit hypermesh sizes of up to 16x16 with at most 2 network communication chips per node. Hypermeshes of size 8x8 or less will only require 1 communication chip per node. Let us also assume that 2 network data transfers can occur in 1 system cycle. This is reasonable if the network busses are unidirectional, short, and only moderately loaded. Such a network can keep a 16-bit processor continuously supplied with data for bursts of computation. Node processors will

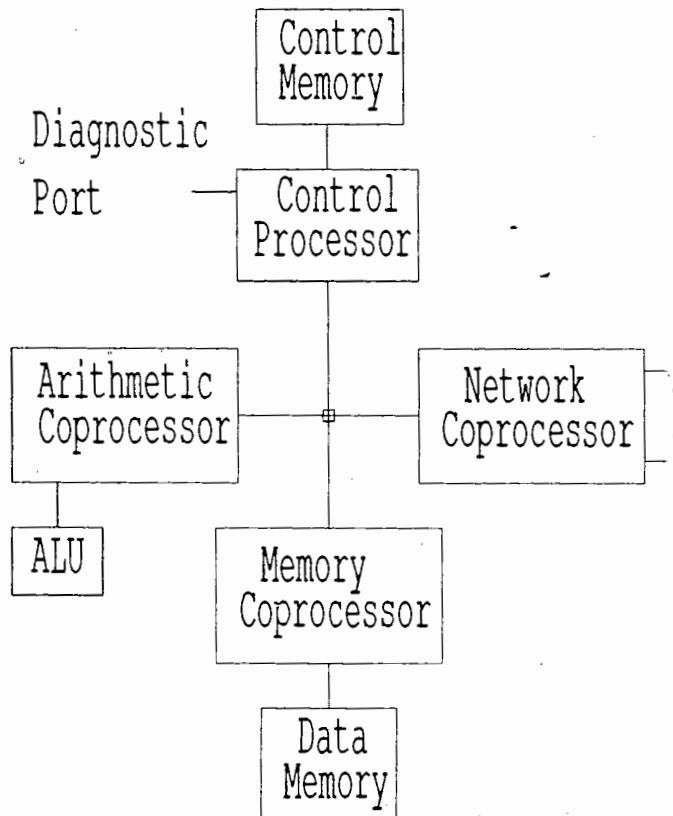


Figure 3-1: Node processor functional components.

therefore have a 16-bit data path between the network, memory, AP, and control processor. Wider data paths exist from the memory controller into the memory, and the AP controller into the arithmetic unit. These functional components of Node Processor, are shown in Fig. 3-1. With submicron technology, many of the functional components will fit onto one chip [HobKaf 89].

3.3.1. A Streaming Memory Interface

Desirable memory interface features have been identified in previous work [HobSim 87]. The SJMC memory coprocessor supports up to 8 data streams for array processing. Systems can be designed with memory cycle times 0, 2, or 4 times slower than the processor cycle time, so low cost DRAM technology can be used. It is desired to retain as much of this *lookahead* capability as possible in the expanded system.

Since arrays are our primary data structure, it is proposed [HobKaf 89], to map data from 1 processor into a network of $n \times n$ processors by interleaving data uniformly amongst the processors. Thus a vector element V_i will be stored with processor P_k where $k = i \bmod n^2$. Large data structures will wrap around many times, while small data structures will not cover the hypermesh. We assume for this discussion that all data structures can be extended to cover the hypermesh uniformly.

3.4. Network Interface Considerations

As mentioned previously, network data paths are 8 bits wide. Each node processor can communicate directly with any processor in its row or column. We also assume the existence of an I/O link for each row and column.

Since network data paths are unidirectional, each processor only needs one output bus which can be used to broadcast data to one or more of the processors in its row or column. Each processor must have $2n$ input busses for receiving data from one or more processors in its row or column (Figure 3-2). The total number of data lines is thus $8 \cdot (2n+1) + 16$, where the '16' comes from a local (bidirectional) data bus. If we assume that 2 control

lines are needed for each communication channel, there will be $4n$ for outputs, $4n$ for inputs, and 2 for the local bus, giving a total of $8n+2$. Not counting clock and power, the total number of pins for a network interface chip is about:

$$IO(n) = 24n + 26$$

$IO(4)$ is quite modest at 122. $IO(8)$ is quite demanding, but feasible at 218.

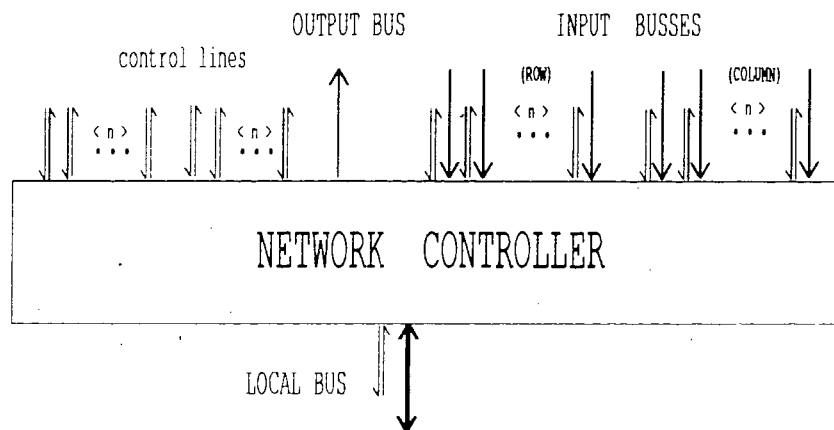


Figure 3-2: Network controller signaling.

An 8x8 hypermesh communication network on a single board would thus be a good commercial target for the near future. Unless one can fit a 16x16 hypermesh onto a single board, the two dimensional hierarchy should be investigated for larger systems. The loading on a single broadcast bus with 32 ports may also be significant.

3.5. Network Controller and I/O Embedding in the Proposed Mesh

The host talks to the network cells through a *network controller*. The purpose of the network controller is to act as an arbiter for the entire network, in terms of initialization and receiving the results. Another thing it does is to act as a bandwidth amplifier between the host machine and the processors. It is not surprising that the *Host Machine* is in fact similar to one of the nodes of the network (Figure 3-3), which has been featured with the same communication controller.

3.5.1. Sample Operations

A typical macro-instruction sent from the host to the network controller is a matrix addition instruction, which specifies the addition of two matrices with one element within each processor. Another macro-instruction could be a matrix multiplication (Appendix C).

3.5.2. Network Input/Output

As in a conventional machine, it is important that a multicomputer machine implementation support a balance of processing and input/output. In some applications the input/output bandwidth may actually dominate the performance of the machine. For example, in the hypercube multiprocessor of dimension n , each node consists of a computation processor, a communication handling mechanism, and a local memory. This communication handling mechanism is in charge of the communications between the host as well as between the n neighboring nodes. The host, having a communication path to each of the nodes, usually performs program development, program and data downloading, and peripheral control (Figure 1-1). Under such a structure, the *host-to-node* interconnection tends to be the system bottleneck, especially at initialization and summing-up stages of the computation. The success of an implementation depends on how well it fits all aspects of the applications, not just the processing. The input/output performance can become extremely important, particularly if this portion of the machine is poorly designed. The objective is to minimize I/O overheads by maximizing parallel I/O capability. Fortunately, the hypermesh machine architecture provides two natural possibilities for high-bandwidth input/output ports, through the communications network and directly to the individual communication controller co-processors. However, having a diameter 2 in this topology, I/O can efficiently be handled solely through the communication network.

Many multiprocessor systems based on the *Mesh* and *Hypercube* topologies have been built recently [TuaPet 85, LinMol 86, ShihIr 87, GeAbGu 88]. In such systems, I/O processors are used to handle the data transfers between the processors and the outside world or the *Host*. In some systems each processor is connected to an I/O processor and

the I/O processor handles all the data transfers between that processor and the outside world. For example, the Intel iPSC system uses I/O hardware within each processor for I/O communication using the ethernet protocol [NaBaAb 88]. In the NCUBE system, an I/O processor is connected to a subcube of 8 processors and the I/O processors are themselves partially interconnected [Hayes 86].

A close look at our topology, gives us another idea. Since we can send a data item to each node of the network in at most two routing steps (2 hops), the communication links between nodes can handle that without any requirement for a dedicated I/O channel. This approach uses the system links efficiently for both the I/O and node-to-node communication. The network itself should be connected to a host node, which can be the same node as the other network's node. One scheme would be the following: The host

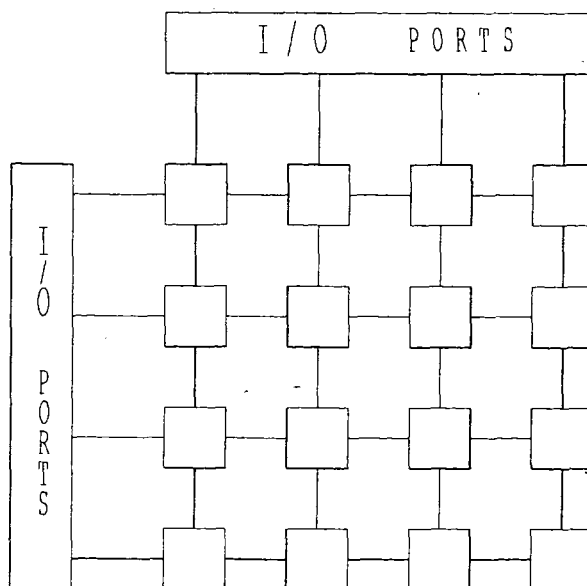


Figure 3-3: Network input/output.

computer is hooked up to the first row and column of the network through a network controller node quite similar to other nodes of the structure (Figure 3-3). By placing the I/O links along first row and column of the network, we do not require explicit I/O system

for the network¹⁵, as they are required in several other networks [Kale 86, GeAbGu 88]. Thus, we can use one of the communication components as a network controller to handle network I/O to/from the outside world.

Since having only one row or column of the network connected to the Host, provides the required communication links for the whole network in just one routing step; this scheme implies a good tolerance of I/O failures. In other words, we can feed the data into the network along two separate path, which will provide high tolerance of I/O failures. As soon as the first column (or row) receives the data/instruction it will broadcast it along the other dimension, using the regular system links.

Utilizing the system links for I/O transfer requires some consideration. We might create congestion along the links when I/O and interprocessor communication have to take place along the same link at the same time. There are two reasons to believe that sharing the links for I/O and interprocessor communication does not lead to congestion. Most problems are solved on multiprocessor systems in the following manner:

- Distribute the data and code to each processor
- carry out the computation in a cooperative manner, and
- combine the results together.

Step 1 and 3 are I/O communications and step 2 requires computation and interprocessor communication. With such a model of solving a problem we can see that the I/O communication and interprocessor communication do not overlap in time. And this leads us to conclude that the system links can be efficiently shared for both I/O communication and interprocessor communication. An obvious problem with this approach is that it forces the first column/row of the network to have a different topology.

In order to provide higher I/O bandwidth for the hypermesh and also to relax the problem just mentioned, the following scheme is proposed. In this scheme the I/O requirement can

¹⁵except the first row/column.

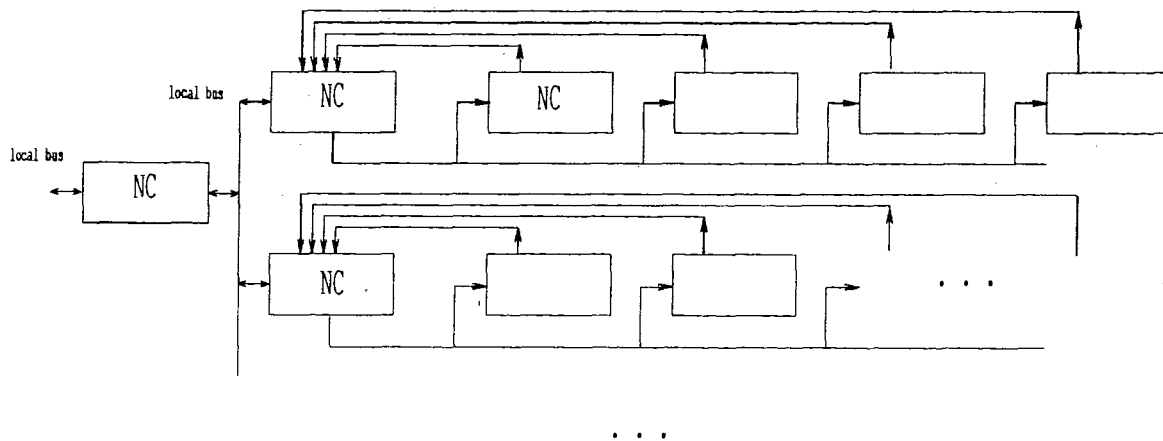


Figure 3-4: A hierarchy of network controllers for I/O.

be handled in a hierarchical manner. This means that the required I/O to this two dimensional network can be provided through a one dimensional array of the nodes requiring exactly similar communication components. In this array, each node is in charge of one row¹⁶ of the network (Figure 3-4). Since the described communication component features two sets (one for each dimension) of $n-1$ input channels for inter node communications, therefore the n th link can be treated for I/O.

The next layer of this hierarchy will probably require a higher bandwidth for more efficient I/O activities. In that respect, local bus links (16-bits wide) can then be assigned and used effectively to provide the required bandwidth, between a single node and the mentioned array of network controllers.

Note that the whole system (including I/O) is quite symmetrical. In general, the node architecture used to implement communication nodes in the base hypermesh (communication layer) can also be used for nodes in the I/O subsystem, thereby reducing the hardware variety in the system. Figure 3-4, shows part of a 16 node configuration. In this figure, a 16 node base hypermesh is controlled by a 4 node NC which, in turn is controlled by a single NC. Thus, there are a total of $1+4+16=21$ NC nodes in the system.

¹⁶could be at most for two rows in hypermesh.

Chapter 4

Evaluating Success and Benchmarking

The appearance of any new computer system raises many questions about its performance, both in absolute terms and in comparison to other machines of its class. Multicomputer networks are no exception.

Repeated studies have shown that a system's performance is maximized when the components are *balanced* and there is no single system bottleneck [ReedFuji 87]. Optimizing multicomputer performance requires a judicious combination of node computation speed, message transmission latency, and operating system software. For example, high speed processors connected by high latency communication links restricts the classes of algorithms that can be efficiently supported [Page 88].

In this chapter expressions for important metrics of network performance for hypermesh will be derived first. Another important performance characteristic of a parallel processor is its ability to perform data permutations. In section 4.2 this issue will be discussed and an upperbound for performing any permutation function on the proposed network with the variety of indexing schemes will be derived. An interesting feature of the D-hypermesh in performing a set of most important permutation functions in constant time will be followed by an analytical proof. Section 4.3 explains our approach to parallel programming (slice concept) and describes the implementation of two applications on a simulator of the proposed multicomputer system.

4.1. Important Metrics of Network Performance and Properties of Hypermesh

Three important metrics of network performance are *latency*, *capacity*, and *throughput* [Dally 87a]. Latency, T_l , is the sum of the latency due to the network and latency due to the processing node.

$$T_l = T_{net} + T_{node} \quad (4.1)$$

Network latency depends on the time required to drive the channel, T_c , the number of channels a message traverses, D , and the number of cycles required to transmit the message across a single channel, L/W , where L is message length.

$$T_{net} = T_c(D + L/W) \quad (4.2)$$

Other important evaluative measures of an interconnection network is the *average distance* [AgJap 86]. This is the distance messages must travel, on an average, in the network. It is advantageous to make this as short as possible. The average distance (in terms of the number of links) is defined as:

$$AvDist = \frac{\sum_{d=1}^r d \cdot N_d}{N-1} \quad (4.3)$$

where N_d is the number of PEs at a distance d links away, r is the diameter and N is the total number of computers. If we select two processing nodes, P_i, P_j at random, the average number of channels that must be traversed to send a message from P_i to P_j is given by the following equation for a hypermesh.

$$T_{av}(n) = \frac{2n}{n+1} \quad (4.4)$$

$T_{av}(4)$ is 1.6. $T_{av}(8)$ is 1.78.

Throughput, another important metric of network performance, is defined as the total number of messages the network can handle per unit time. One method of estimating throughput is to calculate the *capacity* of a network, the total number of messages that can

be in the network at once. Typically the maximum throughput of a network is some fraction of its capacity [Dally 87a]. The network capacity per node is the total bandwidth out of each node divided by the average number of channels traversed by each message [Dally 87a]. For an $n \times n$ hypermesh, the bandwidth out of each node is $(2n-1)W$, and the average number of channels traversed is given by (4.4), so the network capacity per node is given by

$$\Psi_{cp}(n) = \frac{(2n-1)W}{\frac{2n}{n+1}} \approx nW \quad (4.5)$$

Throughput will be less than capacity because not all channels can operate at the same time. In hypermesh either row or column can be used for data transfer operations at each time. This will make the throughput $\frac{nW}{2}$. A typical value for throughput is about 1 in a torus (mesh with wraparound connections) [Dally 87a].

4.2. Some Fundamental Permutations on Hypermeshes

In order to analyze the performance of a multicomputer system it is necessary to characterize its data permutation ability [ReevGut 89]. A permutation on an ordered set of N nodes can be defined by a one-to-one function $\pi(x)$, where¹⁷ x and $\pi(x)$ are integers in the range $0 \leq x, \pi(x) \leq N-1$ [HoJess 81]. It is often found that a simple way of defining a permutation can be obtained by looking at the binary representation of x . Thus

$$x = \{b_n, b_{n-1}, \dots, b_1\} = b_n 2^{n-1} + b_{n-1} 2^{n-2} + \dots + b_1 2^{n-1} \quad (4.6)$$

represents the binary address of an element in the set. Permutations of the set of inputs can now be defined by operations or permutations on their binary address (Figure 4-1).

In this section the performance of the hypermesh for a number of important data permutations is described in detail. These permutations occur in many scientific problems and knowledge of their performance may also be useful in guiding a programmer to develop efficient programs.

¹⁷ x and $\pi(x)$ represent the addresses of the elements before and after the permutation, respectively.

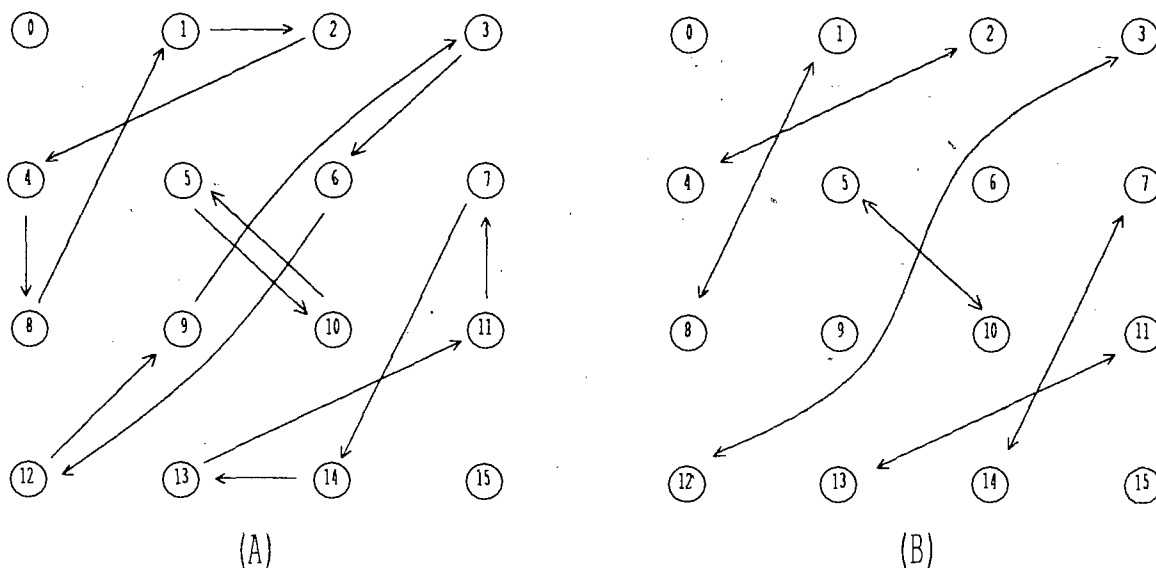


Figure 4-1: Flow diagram of (A) perfect shuffle, and (B) bit reversal permutations on hypermesh.

A simple routing algorithm for performing permutation functions on hypermesh is described first. Then an upper-bound for any permutation using this algorithm will be derived. In the rest of this section studies on some fundamental permutations on varieties of hypermeshes (in terms of indexing schemes) will be presented. This will be followed by an analytical proof verifying an interesting feature of the D-hypermesh in performing the set of studied permutations in constant number of communication steps.

4.2.1. A Simple Routing and message density for permutations

We perform this in a two phase algorithm. In phase I of the algorithm, we displace the messages between columns. This initial displacement ensures that there is no congestion in routing the message to its target, if two nodes intend to exchange messages. Phase I moves data to the same column as the destination is located (along the rows). Also, in phase I data is moved between adjacent nodes, (if the source and destination are on the same row/column). At the end of phase I, each node is holding at most $\sqrt{N}-1$ messages. In phase II data is moved within columns to its destination row. Since a node can play at

most $\sqrt{N}-1$ times as a pivot, and the largest column distance to be covered is I , the number of data transfers required in phase II to reach the target row is at most $\sqrt{N}-1$. Thus, the total number of data transfers executed by the algorithm is $I + \sqrt{N}-1 = \sqrt{N}$.

This simple algorithm will result in a path in which one node acts as an intermediary, we call it *pivot* here, for each pair of nodes. Clearly, whenever a pair of nodes swap their data this simple routing algorithm is optimal in the sense that they always utilize different pivots for each direction. For certain permutations some nodes are involved more than others in message transmission, and not all the nodes carry an equal amount of traffic. However, in this case, the maximum number of queued messages at those heavily loaded pivots will never exceed \sqrt{N} , which gives the upper bound for any set of permutations to be exactly \sqrt{N} .

Some enhancements are possible. For example, if we use a proximity ordering for the network nodes, a similar routing algorithm can perform perfect shuffle and bit reversal permutations both in 3 steps, using hypermesh of size 8×8 . A hypermesh with proximity ordering has an interesting property that $node_i, node_{i+1}$ are neighbors. Also this mesh may be recursively subdivided into sub-meshes such that each sub-mesh contains consecutive indexed nodes. Results on performing the set of fundamental permutations on hypermeshes will follow their quick definition.

4.2.2. Exchange Permutation

The exchange permutation can be defined in terms of the binary representation of x .

$$\varepsilon_k(x) = \{b_n, \dots, \bar{b}_k, \dots, b_1\} \text{ where } 1 \leq k \leq n \quad (4.7)$$

The bar denotes the complement of a given bit. Thus the K^{th} exchange permutation can be defined by complementing the kth bit of the binary representation of x .

4.2.3. Perfect Shuffle Permutation

The perfect shuffle is so called as it can be performed by cutting the set in two and interleaving the two sets obtained, as in the perfect card shuffle. This permutation corresponds to a unit circular left shift of the binary representation of x .

$$\sigma(x) = \{b_{n-1}, b_{n-2}, \dots, b_1, b_n\} \tag{4.8}$$

In terms of row and column indices this can be written as

$$\sigma(r,c) = (2r \bmod \sqrt{N} + E_c, 2c \bmod \sqrt{N} + E_r) \tag{4.9}$$

where

$$E_c = \begin{cases} 1 & c \geq \sqrt{N}/2 \\ 0 & \text{otherwise} \end{cases}$$

$$E_r = \begin{cases} 1 & r \geq \sqrt{N}/2 \\ 0 & \text{otherwise} \end{cases}$$

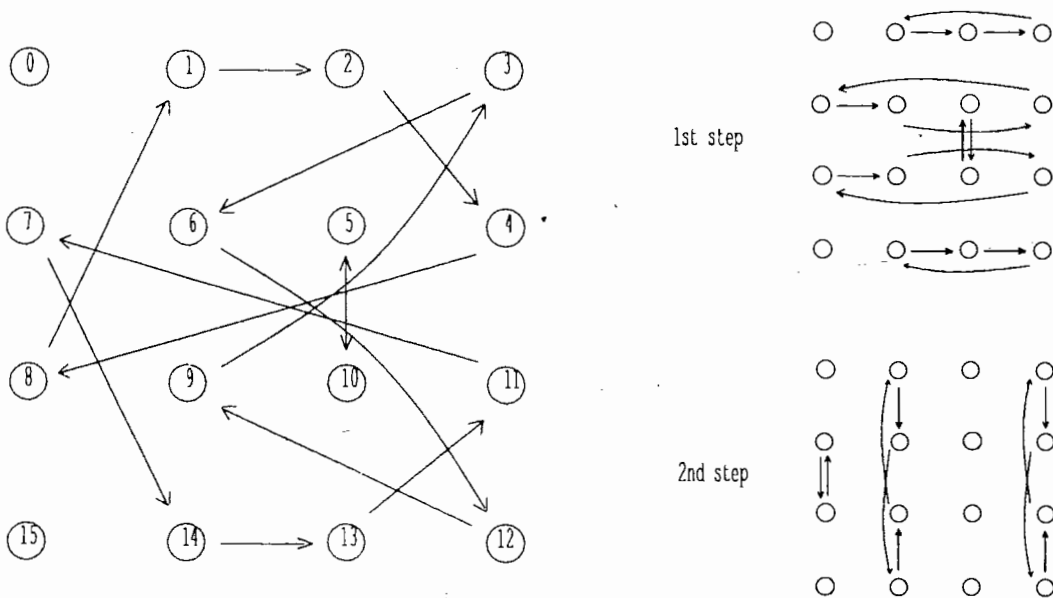


Figure 4-2: Perfect shuffle on hypermesh with snake like ordering and 2-step routing solution.

4.2.4. Butterfly Permutation

The butterfly permutation is defined over the binary representation of x by exchanging the first and last bits.

$$\beta(x) = \{b_1, b_{n-1}, \dots, b_2, b_n\} \tag{4.10}$$

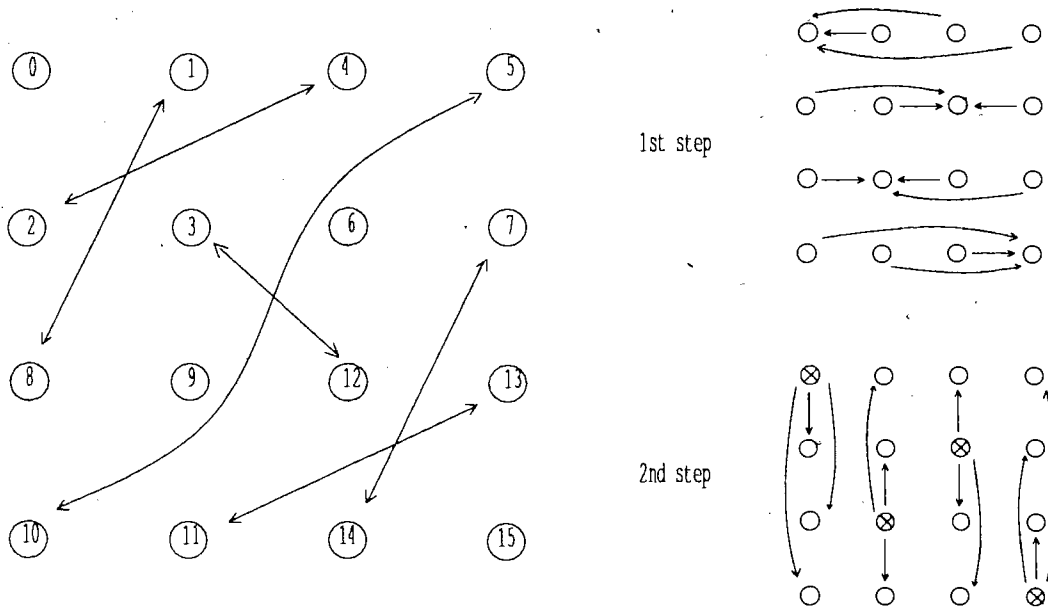


Figure 4-3: Bit reversal on a hypermesh with a shuffle row ordering; and the overloaded pivots after first step of the data routing.

4.2.5. Bit Reversal Permutation

The bit reversal permutation, as its name suggests, is defined over the binary representation of x by reversing the order of bits (Figure. 4-3).

$$\rho(x) = \{b_1, b_2, \dots, b_n\} \tag{4.11}$$

In terms of row and column indices this can be written as

$$\rho(r,c) = (c^R, r^R) \tag{4.12}$$

$$x^R = \text{reversal of } \{x_{n-1}x_{n-2} \dots x_0\}$$

$$= \{x_0x_1 \dots x_{n-1}\}$$

It is interesting to note that, the matrix transpose algorithm can be defined as a bit reversal permutation on a hypermesh network. One application where this permutation occurs is in the Fast Fourier Transform algorithm [ReevGut 89].

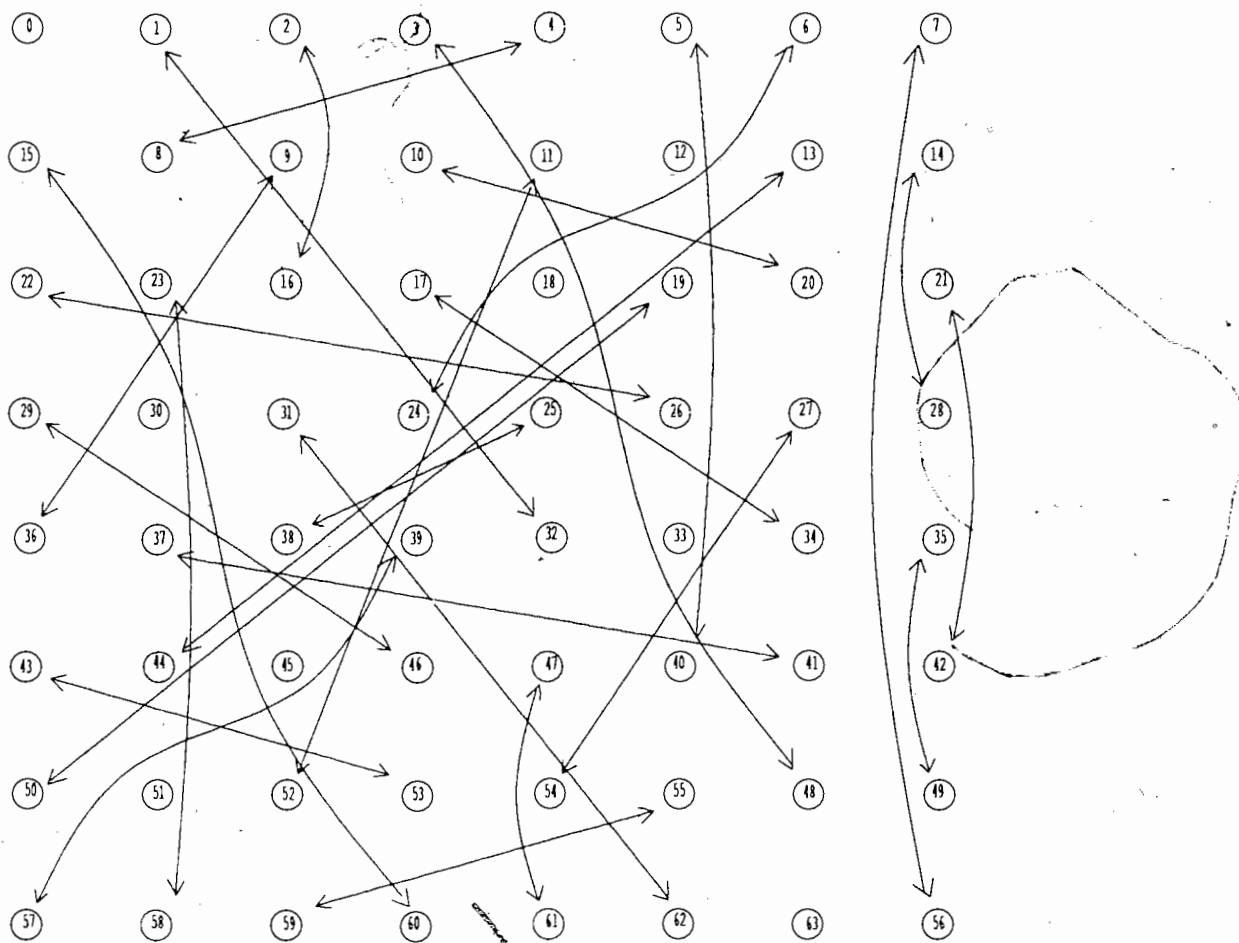


Figure 4-4: Flow Diagram of bit reversal permutation on an 8x8 D-hypermesh.

4.2.6. Shift Permutation

The near-neighbor interconnection network of the MCC can only directly implement the shift permutation. Any other permutations can only be achieved through the shift permutation. Clearly, this is not the case in hypermesh network. The shift permutation can be defined as following. In terms of the binary representation of x , the following equation defines the binary addition over the n -bit field, ignoring overflow .

$$\alpha(x) = |x+1|_{2^n} \tag{4.13}$$

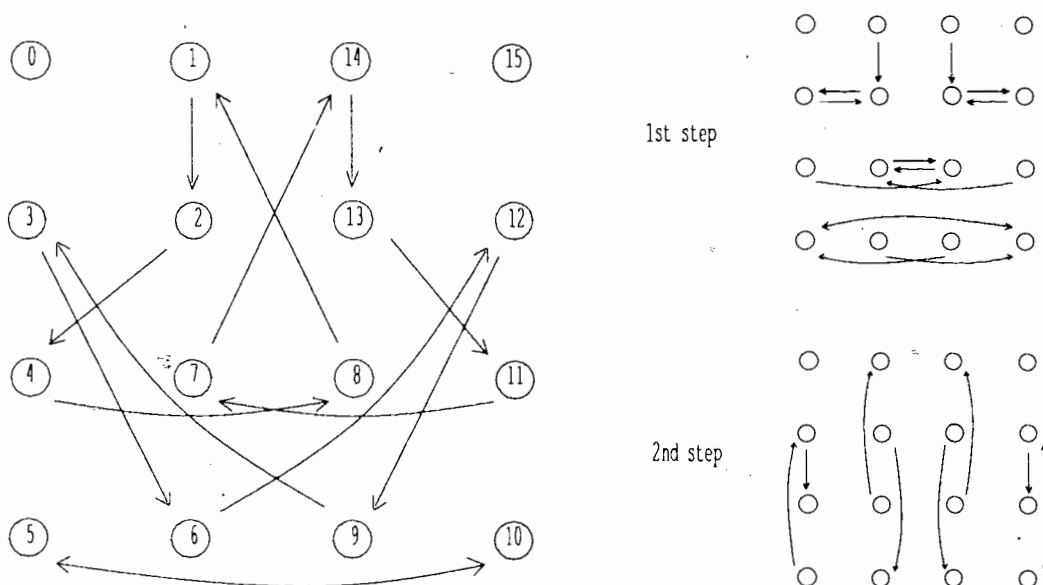


Figure 4-5: - Perfect shuffle on hypermesh with proximity ordering; and 2-step routing solution.

It is important to note that some of these permutation functions can be realized with fewer routing steps on other types of ordering in hypermesh (Figures 4-2, 4-3, 4-5).

A summary of the complexity results of performing these permutations on hypermesh networks ($n \times n$) is given in Table 4-1. Each entry in this table indicates the number of routing steps required to perform the corresponding permutation function for the specified

ordering of the nodes. As the table shows, D-hypermesh turns out to have very interesting properties in terms of permutations. For these permutations a 2-step routing solution using the routing algorithm presented in Chapter 2, exists. For example, the bit reversal permutation can be performed on this network, in just 2 communication steps, regardless of the network size. Figure 4-4 shows the flow diagram required to perform the bit reversal permutation on an 8x8 D-hypermesh (in here network horizontal and vertical links are elided).

	row major	snake like	proximity (n=4,8)	shuffle row	d-hypermesh
bit reversal	n	n/2	3	n	2
perf. shuffle	3	2	log n	n	2
exchange	1	2	3	1	1
butterfly	2	2	2	2	1

Table 4-1: Routing complexity on hypermesh of size (nxn) with a set of indexing scheme for a variety of frequently used permutations.

4.2.7. Analytical Proof for 2-Step Routing on D-hypermesh

Experimental results (using an APL program) for performing permutation functions show a 2-step routing solution on D-hypermesh with up to 256x256 in size. In order to generalize this property to an arbitrary size of network, an analytical proof is given here. Here the bit reversal permutation is used to demonstrate this proof. Clearly, this can be done for some other permutation functions as well. In an nxn network of processors¹⁸, the

¹⁸Define $n=2^k$; c, r are k -bit binary encoding of column and row numbers.

bit reversal permutation¹⁹ function in terms of row and column indices can be written as

$$\rho(r,c) = (c^R, r^R) \text{ where } 0 \leq r, c \leq n-1 \text{ and } 0 \leq c^R, r^R \leq n-1 \quad (4.14)$$

The pivot node for each data transmission can be found using the function

$$\delta(r,c) = (r, (c^R + r^R - r) \bmod n) \quad (4.15)$$

In order to prove that this network can perform permutation functions in 2-steps, we have to demonstrate that in the first step of the routing algorithm, none of the pivots receives more than one message to forward.

To do that, it is sufficient to prove that the pivot function, δ , is an injective function (i.e. if $\delta(x_1, x_2) = \delta(y_1, y_2) = (z_1, z_2)$, then $x_1 = y_1$, and $x_2 = y_2$).

Suppose (x_1, x_2) and (y_1, y_2) are the coordinates of 2 nodes. Then

$$\begin{aligned} X &= \delta(x_1, x_2) = (x_1, (x_2^R + x_1^R - x_1) \bmod n) \\ Y &= \delta(y_1, y_2) = (y_1, (y_2^R + y_1^R - y_1) \bmod n) \end{aligned} \quad (4.16)$$

are the pivot nodes of (x_1, x_2) and (y_1, y_2) respectively.

Two tuples are equal if corresponding terms are equal. So,

$$\text{if } X = Y \text{ then } \begin{cases} (a) & x_1 = y_1 \\ (b) & (x_2^R + x_1^R - x_1) \bmod n = (y_2^R + y_1^R - y_1) \bmod n \end{cases} \quad (4.17)$$

Since ρ is an 1-to-1 function, from Eq.(4.17)-a, we get

$$x_1^R = y_1^R \quad (4.18)$$

Substituting $x_1^R - x_1 = y_1^R - y_1 = C$ into Eq.(4.17) yields

$$(x_2^R + C) \bmod n = (y_2^R + C) \bmod n \quad (4.19)$$

Since $a \bmod n = b \bmod n \rightarrow (a+d) \bmod n = (b+d) \bmod n$. Therefore

$$(x_2^R + C + (-C)) \bmod n = (y_2^R + C + (-C)) \bmod n$$

¹⁹Define c^R , to be a bitreversal of c .

can be simplified to:

$$x_2^R \equiv y_2^R \pmod{n} \quad (4.20)$$

Finally since $x_2^R, y_2^R < n$ (from Eq.(4.14)) implies that

$$x_2^R = y_2^R \xrightarrow{20} x_2 = y_2$$

the proof is complete. \square

4.3. Environment for Multicomputer Simulation

For effective use of parallel systems, it is essential to obtain a good match between algorithm requirements and architecture capabilities. Information which captures the relationships between parallel algorithms and parallel architectures can be investigated using a simulation. Moreover, task (application) level modeling of multiprocessor architectures may produce some good insight into the trade-offs between computation versus communication, low versus large granularity, alternate mapping, scheduling and both static and dynamic routing strategies.

The lack of adequate system software is currently the largest hindrance to parallel program development for multicomputer networks [ReedFuji 87]. This is partly because of the requirement for a network based operating system to support message-based communication features at the software level, on top of primitive communication support at the hardware level. Another reason perhaps is that no matter what the interconnection network looks like, the communication patterns required by some algorithms will be inefficient or difficult to formulate.

One of the objectives of this thesis was to study a possible simulator for a multicomputer

²⁰Again, because bitreversal is an 1-1 function.

system based on the same language as the real version²¹. Then, since the simulation and implementation languages are identical, the overhead of transporting software to the real implementation is negligible. This enables us to consider the correctness issue of the programs outside the structure of the real system.

Simulations of the hypermesh were performed to evaluate various design options and to validate that our design of a message transmission could meet the objectives. A hypermesh simulator has been set up using the APL and C languages. The essence of this approach is that each line of APL code represents one real microinstruction [Hobson 87]. A matrix multiplication algorithm has been implemented on the multiprocessing simulator, to verify the correctness of the algorithm and also to verify the required communication primitives on such a network. Some other sample algorithms using a *tree reduction technique* and also using a *centralized control algorithm* for array summation has been implemented. The system is written in APL and C language and run on a Sun workstation. The code is divided into two parts.

- An APL program which is the implementation of the matrix-to-matrix multiplication algorithm, specifically designed for the hypermesh network. The node program is exactly the same for all nodes. Another simple program is also required (for a network manager) to set up and configure the network.
- A C code, implementing the communication primitives required for the communication between APL processes. The role of the C code, is in fact, to facilitate the communication between tasks (node programs) running on different APL environments.

Work on a more realistic hypermesh multicomputer simulator using APL and C on a network of Sun workstations has been initiated, but because of the problem with the existing APL-C interface, the Inter-net communication primitives cannot be implemented directly in C as a routine²². One of the aspects that needs to be considered in any

²¹In a very recent work by Olsen *et al.*, [BaOlSo 89] "Occam" language has been used for a simulator of a network of transputers.

²²An internet communication activity was causing a crash on APL-C interface. However, that was taken care of by adding more complexity in Internet communication and implementing it as interrupt handler routines.

simulation is to ensure that the simulator is free from properties like deadlock and various time-dependent errors. In this respect, more work needs to be done in terms of the handshaking requirements.

The algorithms developed in this chapter have a great deal in common:

- node processors are synchronized by passing messages,
- messages are short, containing constant length,
- routing decisions are solely based on local information.

To implement a good data parallel algorithm on the hypermesh multicomputer, one has to consider the number of processors required, an efficient way to partition the data, an efficient way to map partitions into processors, and the role of the network controller must be determined. We use two simple examples, array summation and matrix multiplication, to demonstrate various techniques to solve these problems. In this technique, each processor knows exactly what to expect from the network as part of its algorithm *slice*. There is no data interpretation overhead.

4.3.1. Array Summation

Given a vector of numbers, a_1, a_2, \dots, a_l , we want to compute their sum, $A = a_1 + a_2 + \dots + a_l$. Each number, a_i , is stored on different node. One approach, the *centralized accumulation method*, is to partition the vector into k subvectors, each having a size of x_i ($0 \leq i \leq l/N$)²³. One subvector is assigned to a node to calculate the partial sum. All partial sums are collected by the host to evaluate A [NiKing 87]. The host then may initiate another step, by redistributing the partial sums among half of the active nodes of the previous step, and carry on this strategy until a single result gets collected by the host. An implementation of this model has been done.

Another approach, the *tree structured accumulation method*, is to use a tree reduction among the nodes to accumulate the partial sums. The host then receives the final sum A

²³ N is the network size.

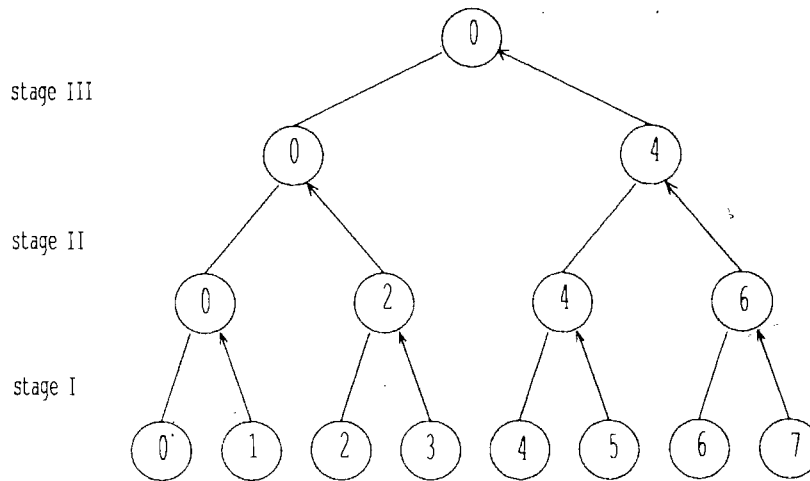


Figure 4-6: Tree reduction scheme.

from the root of the reduction tree (Fig. 4-6). There are other tree numbering conventions which could be used in tree reduction algorithm. For example, a tree reduction algorithm based on the *balancing tree* [Dally 87a] model has also been implemented (in a simulator).

To describe the concept of the algorithm *slice* in a synchronized array processing a scalar aggregation on a hypermesh will be discussed. Here we consider the addition of a vector of elements (or partial results), each resident in the local memories of each processor in the hypermesh. First, all the nodes compute their local aggregate values. Next, all the local aggregate values need to be combined to determine the global aggregate value. The global aggregation phase takes $\log n$ steps for one row of an $n \times n$ hypermesh²⁴.

In the k^{th} step, $k=1$ to $\log n$, nodes (p_i) whose rightmost k address bits are equal to the rightmost k bits of the host address (the root), read the aggregate value from the nodes which differ in address (from p_i) in the k^{th} bit. Clearly $2 \log n$ steps would be required to get the final result at the root (i.e. node(0,0)). A distributed routing algorithm of low complexity has been implemented using a simulator. Each node in the network, has a

²⁴Clearly, this process is running simultaneously for all rows of the network, in which nodes along the first column act as temporary roots for their row correspondingly.

binary number of length $2\log n$ corresponding to its position in the mesh. All nodes are programmed equally, and the routing algorithm is based on the node ID. A simplified algorithm for one row of the network is given in 4-6. In this algorithm n processors are employed, each initially holding one input value.

```

for all PEs do; 0 ≤ ID ≤ n-1
  for step=1 to log n do
    bit=myid(step)
    if bit=0 then
      receive-from (myid ⊕ bit)
    else
      send-to (myid ⊕ bit)
    exit
  fi
od
od
/* myid(i) is the ith bit in binary
   encoding of the node ID. */

```

Figure 4-7: A typical slice algorithm for a scalar aggregation in synchronized array processing.

The addition across a set of eight elements is shown in Fig. 4-6. The figure shows the stages and the binary tree structure control in the operation. The architecture is initially partitioned into clusters of two adjacent nodes with one active processor in each cluster.

4.3.2. Matrix Multiplication

Let A and B be matrices of size $(n \times n)$, the network size. In forming the matrix product $C=A \times B$ with elements

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad (1 \leq i \leq n, 1 \leq j \leq n) \quad (4.21)$$

there are n^3 products $a_{ik} b_{kj}$ to be calculated. There are various strategies for forming this product on a parallel computer with $n \times n$ processors [JagKai 89]. The matrix $C=AB$ has n^2 entries, each in the sum of products of n pairs of numbers.

If * denotes term-wise multiplication respectively on objects such as matrices and vectors; for vector v and ω :

$$v * \omega = (v_1, \dots, v_n) * (\omega_1, \dots, \omega_n) = (v_1 * \omega_1, \dots, v_n * \omega_n)$$

Then all multiplications can be performed (notationally with a single application of *), by multiplying positions of the data entries, at each node (n -steps). For instance, for a 2x2 case, we have the following

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

which is now more conveniently arranged as

$$\begin{bmatrix} (a_{11}, a_{12}) & (a_{11}, a_{12}) \\ (a_{21}, a_{22}) & (a_{21}, a_{22}) \end{bmatrix} * \begin{bmatrix} (b_{11}, b_{21}) & (b_{12}, b_{22}) \\ (b_{11}, b_{21}) & (b_{12}, b_{22}) \end{bmatrix}$$

Now various additions must be made (after performing the termwise operation "*"), and sums assigned to the corresponding position in C (at each node). In general there are n^2 results in the result matrix and each of the entries consists of the addition of n numbers (which takes $\log n$ steps using tree-reduction technique). However, since a multiply-accumulate operation can be done in just one operation (pipelined), there is no reason for that extra addition step.

A brief description of the program is the following: The Control Node program configures the network²⁵. It initially broadcasts the size of the network. Then it receives a vector operation by interacting with the user, and then broadcasts the vector operation to the entire network. Each node then starts executing its own program, (all nodes are programmed equally). Each node is assigned an *ID* associated with its position in the hypermesh structure. All the decisions making during the execution of the node program

²⁵Control Node is acting like a Cube Manager in hypercube architecture.

are based on the node ID. The algorithm consists of 2-step broadcasting operations (one in A, and one in B), followed by multiply-accumulate operation at each node. In the first step, the elements in each column of the matrix A, is broadcast to all others. Similarly, in the second step, the same algorithm will take place for the matrix B (in each row). These steps leave $2n$ data at each node ready to be consumed by the node processor for n local multiply-add (pipelined) operations. Moreover, all the routing decisions are solely based on local information.

Up to this point we have assumed that there are enough cells in the network to hold the entire problem. Of course, there will always be problems too big to hold on a physical machine. One is that the size of matrix is larger than the size of the network. In this case, the matrices need to be partitioned into smaller units, where each unit is dealt with in parallel; for example, if the matrices A, B, \dots, H are all of order $n \times n$, and with n^2 processors available, the most obvious method of multiplying matrices of order $(2n)^2$ is as follows:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

Where each product in the right-hand side is computed in parallel. In general case, it is possible that the given matrix is not a complete permutation of the network size. One simple solution to this problem is to augment the matrix with extra zero elements (along rows/columns) in order to get a complete permutation of the network size. Then split it into small sizes each one the same size as the network. Figure 4-8 shows one example of such partition. This approach has been taken in [HobKaf 89].

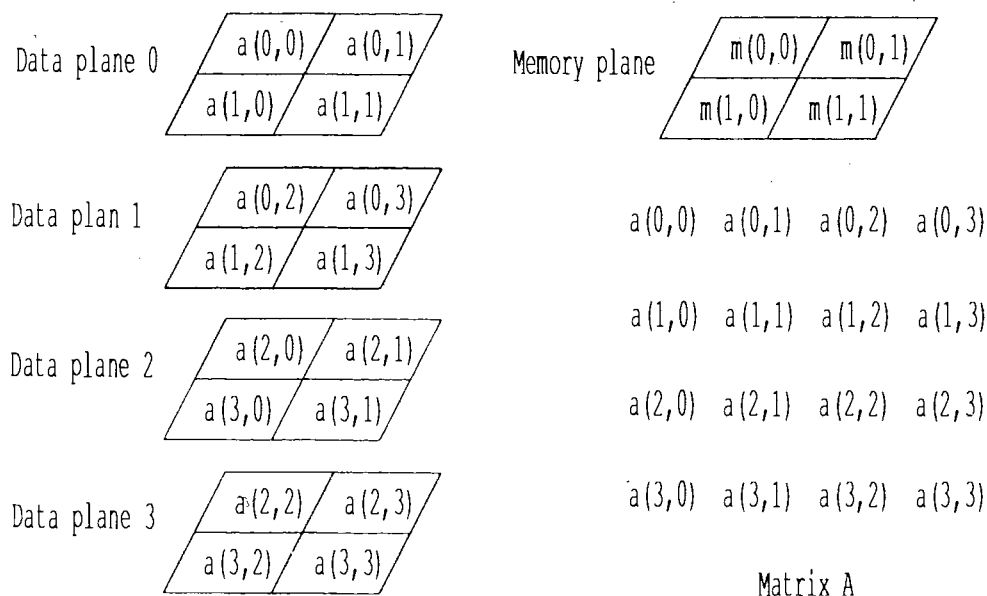


Figure 4-8: Mapping 4x4 matrix items on 2x2 processor array.

4.3.3. Performance Study of Matrix Multiplication

All hypermesh computations combine both communication and computation; hence, a single number such as MIPS, MFLOPS, or bits/sec will not accurately reflect communication and computation or the performance for different applications.

In evaluating a parallel system, two performance measures of particular interest are *speedup* and *efficiency* [EaZaLa 89]. Speedup is defined for each number of processors p as the ratio of the elapsed time²⁶ when executing a program on a single processor (the single processor execution time) to the execution time when p processors are available. In the notational form,

²⁶The cost metric could be a throughput, which is an appropriate cost measure if one has many such computations to be performed and the computations may be overlapped [Whelan 88].

$$S(p) = \frac{T_1}{T_p} \quad (4.22)$$

Efficiency is defined as the average utilization of the p allocated processors. Ignoring I/O, the efficiency of a single processor system is 1. Speedup in this case is of course 1. In general, the relationship between efficiency and speedup is given by

$$E(p) = \frac{S(p)}{p} \quad (4.23)$$

The theoretical maximum value of $S(p)$ appears to be p (and of $E(p)$ to be 1), attained when the algorithm is fully parallel and the calculation is distributed equally among all processors (processing elements). This time may be thought of as measured in clock periods. An efficiency study has been done [HobKaf 89] for matrix multiplication algorithm. MicroAPL techniques have been used to demonstrate how $n \times n$ matrix multiply may be broken into *outer* and *inner* routines for execution on the hypermesh [HobSim 87, HobTho 81]. A copy of the code with explanations can be found in Appendix C (from [HobKaf 89]). Uniprocessor version of this algorithm (from [HobGud 86]) can also be found in Appendix B. For matrices of size M ; ($m \times m$), and hypermesh of size N ; ($n \times n$), the efficiency function reveals that for $m \geq 16$ and $n \geq 8$, the efficiency is ≥ 1 [HobKaf 89]. This interesting result is due to a more efficient inner loop in the hypermesh algorithm than in the uniprocessor algorithm. After processors broadcast row/column data, the network co-processor can deliver this data for computation without the same startup penalty as the local memory system.

A significant advantage that synchronized array processing algorithms have over message passing concurrent algorithms is that data exchanges through the network are very precise. Each processor knows exactly what to expect from the network as part of its algorithm *slice*. There is no data interpretation overhead.

4.3.4. Matrix Multiplication on Diagonal Hypermesh

Excellent features of diagonal hypermesh in performing most important classes of permutations have been discussed earlier. Now we are interested in performing matrix multiplication on D-hypermesh. It is clear that the communication pattern required for matrix multiplication is not directly matched to the D-hypermesh structure. However, several approaches can be taken into consideration. One could look for another algorithm. For example, each column of matrix B (say col. i) can be projected into the row i , simply in one routing step. Then using the horizontal links and broadcast operation and finally reprojection of these data from row i to column i , it will end up in the same data setup requirements as the previous scheme for matrix multiplication on regular hypermesh, (of course the elements of matrix A must also become available to all nodes along rows which needs a single broadcast operation). In this scheme, the reprojection step requires n data transmission steps, which is disappointing (Figure 4-9).

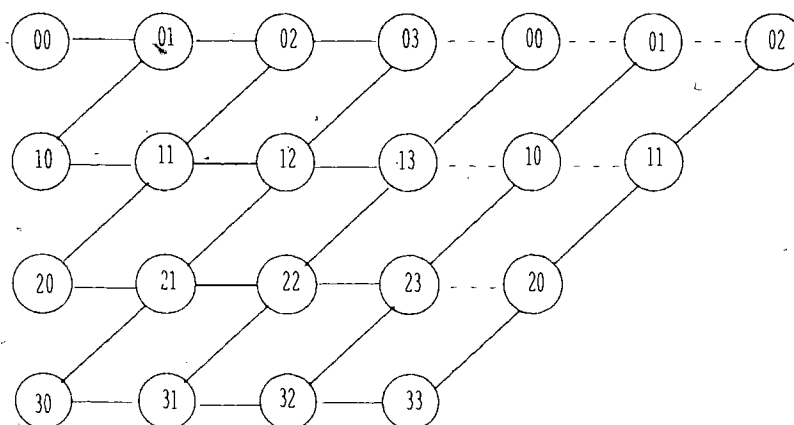


Figure 4-9: Another representation of a Diagonal Hypermesh.

In terms of time complexity, this scheme requires $O(n)$ steps communication and $O(n)$ times computation. Therefore, the total complexity stays unchanged (as compared to the same operation on regular hypermesh). However, a close look at the APL implementation (hardware execution) of this algorithm, and an asymptotic analysis of its execution time (in

terms of the number of cycles), will result a degradation in speed over 25 percent, for a 4x4 D-hypermesh. For a network of 64 processors (8x8 D-hypermesh) this degradation is over 30 percent²⁷. Another approach would be a technique by Winograd [JagKai 89] to compute the matrix multiplication using the following formula:

$$c_{ij} = \sum_{k=0}^{(n_k/2)-1} (a_{i,2k+1} + b_{2k,j})(a_{i,2k} + b_{2k+1,j}) \quad (4.24)$$

$$- \sum_{k=0}^{(n_k/2)-1} a_{i,2k+1} a_{i,2k} - \sum_{k=0}^{(n_k/2)-1} b_{2k+1,j} b_{2k,j}$$

The advantage of this procedure is that only the first summation, which requires half as many multiplications as the straightforward algorithm, need to be computed for each value of the pair i,j . The second summation need just be evaluated once for every value of i ²⁸, and the last summation for every value of j ²⁹. This means that these two sums can be evaluated first at each row and column (using fastest technique, namely tree reduction along row/column) and then the final result can be broadcast to other nodes along row, or column accordingly. Then these two terms can be combined together (add operation) locally, and form a constant number as a initial value for further multiply-add operations. The effort required for communication and then computation of the first term dominates the final elapsed time in D-hypermesh, and also the pipelined multiply-add operation which is a single cycle operation in current arithmetic units cannot be used efficiently. On hypermesh class of networks this approach turns out to be no better than the previous approach.

It is possible to rearrange the initial data at each node, in order to derive a more efficient solution to this problem. This can be done by reordering the initial data along rows of the

²⁷This is simply because of the extra number of communication based operations required in this approach.

²⁸all the nodes along each row will have the same value.

²⁹all the nodes along each column will have the same value.

network (Figure 4-10). The crux of this algorithm is a data routing operation which we shall now define. All the data permutation operations are cyclic shifts on rows or columns

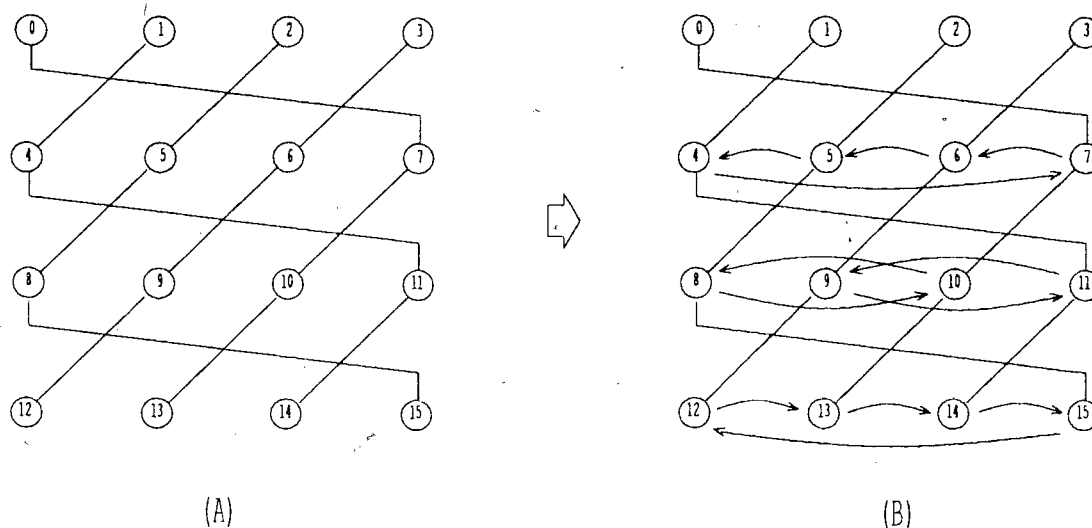


Figure 4-10: One step reordering in a D-hypermesh; horizontal links are elided.

and may be broadly categorized under the following:

$RRotate(+/-x)$, $CRotate(+/-x)$

As the name suggests, there are cyclic shifts in the horizontal (Row) and vertical (Column) directions, respectively. The amount of shift in each row(column) is determined by a parameter (x). Figure 4-10 illustrates an instance of $RRotate(+myrow)$. The elements in a row are cyclically shifted. It is clear that shift on rows can be carried out in just one routing step in D-hypermesh. Therefore, any data permutation operations defined by $Rotate()$ operation can be achieved in one parallel routing step if data under shift is in the router, otherwise, a memory access time must be added to the required time.

Simply applying this Rotate operation along rows of the D-hypermesh, with the amount of shift equal to node's row index, will result a regular hypermesh-like network. Then, the same matrix multiplication algorithm on hypermesh can be carried out. Finally the end results should be shifted back to the right places. Therefore, only small degradation in performance over regular hypermesh will be caused.

Performing a permutation from the set of most important permutation functions (Section 4.2; Table 4-1) on regular hypermesh seems to be possible to benefit from the current results on transforming D-hypermesh to regular hypermesh. Since the mentioned permutation functions can be performed on a D-hypermesh in at most 2 routing steps for any network size, therefore performing one transformation from regular ordering to D-hypermesh ordering before the permutation task, and another rearrangement (fix-up) step right after the permutation task, will give a 4-step routing solution for the set of mentioned permutation functions regardless of the network size. This gives, for example, a 4-step routing solution for performing bit reversal permutation on a regular hypermesh of size 256, (16x16), instead of 16 steps required otherwise. Thus, any algorithm which requires the class of the studied permutations can always be achieved in constant data routing time.

Chapter 5

Conclusions

Parallel architectures and the way that they support the efficient execution of parallel algorithms is an important area of current research related to high-performance computer systems. The choice of an appropriate architecture for any electronic system, is very closely related to the implementation technology. This is especially true in VLSI computer systems whose computational goal is the implementation of compute-bound algorithms rather than I/O-bound computations. VLSI technology can provide us with a novel set of building blocks for the construction of high performance point-to-point networks for closely coupled multicomputer systems.

One of the objectives was to get topologies with minimum diameter and minimum average distance simultaneously. We have shown an appropriate design choice of the adjacency pattern between network elements, yielding a network satisfying this constraint. This thesis has presented different modifications that can be made to a standard mesh-connected parallel processor organization, and has shown how they support efficient parallel algorithms for performing an important set of computational problems. The proposed system is suitable for the large class of scientific applications which involve regular operations on data arrays. Many of these applications involve matrix operations such as the Fast Fourier Transform (FFT), in which data permutation is the basic functional primitive, and matrix multiplication. The problem of efficiently performing permutations on a hypermesh system has also been considered. Here, a very simple control algorithm on the hypermesh network has been proposed, which can realize many frequently used permutations in constant number of steps. However, the upper bound for any arbitrary permutation has been shown to be $N^{1/2}$. Another variation of hypermesh named diagonal hypermesh has been introduced. An APL implementation of permutations on Diagonal

hypermesh revealed that a 2 step routing solution exists, independent of the network size. This interesting result has been accompanied with an analytical proof. Results about the performance attained by hypermesh network have been presented here and comparison with some other mesh-type networks are provided.

Another important advantage of the hypermesh is its ability to map other communication topologies onto itself. In this regard a direct mapping of hypercubes onto the hypermesh has been discussed. This property highly simplifies algorithm design and allows the exploitation of very efficient communication patterns. For a wide class of problems, the organization offers significant performance advantages over regular mesh-connected computers, or other mesh modifications that have been proposed previously. The strong connectivity, regularity, and symmetry of the hypermesh and also its versatility in embedding many other networks in linear complexity makes it a good candidate for a more general-purpose parallel processors. Many classes of algorithms can be naturally mapped onto the hypermesh, and distributed routing and broadcasting can be implemented efficiently. A hypermesh's full features can be exploited in array processing operations where the slicing concept is used by implementing a synchronized array processing algorithm. However, this architecture may not be well suited for regular message passing systems with asynchronous communication requirements.

The hypermesh is not easily expandable. This hampers a modular growth of the network. We should thus seek a hierarchical solution to parallelism in the same sense that we have hierarchical memory systems. At the bottom of the hierarchy we have modestly sized and very efficient arrays of processors. Above that layer one must tolerate gradual degradation of efficiency due to inherent physical constraints. Network topology in the bottom layer may be different from network topology in higher layers. This is an open problem. In fact, hypermesh system's expandability is due predominantly to the design of the NC chip. The wiring complexity of the hypermesh grows at the rate of \sqrt{N} , where N is the network size. This penalizes the hypermesh seriously under the packaging constraint. One solution to this problem is reducing bandwidth which will allow us to have more communication links for the NC chip.

Simulations of the hypermesh were performed to evaluate various design options and to validate that our design of a message transmission could meet the objectives. Hypermesh simulator has been set up using APL and C languages. The essence of this approach is that each line of APL code represents one real microinstruction. A matrix multiplication algorithm has been implemented on a multiprocessing simulator, to verify the correctness of the algorithm (slice) and also to verify the required communication primitives on such a network.

Another goal of this thesis was to investigate hardware support for data passing that obviates the need for software control. Some key issues which must be considered when designing a high performance network controller based on VLSI technology has been discussed. Technological considerations in the design of a communication component have also been examined. By offloading communication to a separate processor, the node processor is potentially free to overlap computation with communication (just the communication needs to be set up). Special instructions have been provided to support communication between nodes. Unlike many other multiprocessor networks the connectivity of the host and the hypermesh nodes is considerably richer, through a hierarchy of network controllers, providing increased flexibility and greater I/O bandwidth.

Overall, the objectives of this thesis have been met. Further research is necessary to determine the practical significance of the hypermesh in the commercial world.

Appendix A

Glossary of Acronyms

ACRONYM	EXPANSION
AP	Arithmetic Processor
D-hypermesh	Diagonal hypermesh
DRAM	Dynamic RAM
FLIT	FLow control digIT
MIMD	Multiple Instruction Multiple Data
MCC	Mesh Connected Computers
NC	Network Coprocessor
PE	Processing Element
SIMD	Single Instruction Multiple Data
SJMC	SAMjr's Memory Coprocessor
VLSI	Very Large Scale Integration

Appendix B

Uniprocessor Matrix Multiplication

MicroAPL techniques are used to demonstrate how matrix multiply may be broken into special *outer* and *inner* routines [HobSim 87, HobTho 81]. Some simplifications are made for the sake of readability. It is assumed that operands are pipelined to a floating-point processor based upon Weitek's chip set used in flow through mode, cf. [Weitek 84]. Action codes are placed in, FPCTRL, while an execution is triggered by an FPEXEC. ERROR is one of 7 directly testable message (interrupt) flags. Each nonempty line of microcode takes one system cycle, T.

The outer routine receives matrix dimensions, M, K, N in registers R[M], R[K], R[N]. Register R[RINDEX] keeps track of columns in the right operand. R[LINDEX] keeps track of rows in the left operand. R[T] holds the column step size in bytes. Data streams for LEFT, RIGHT, and DEST are also passed to MATMUL by the format routine. Data streams are started by SWW (segment write word) or SRW (segment read word). Data streams are advanced by SSN (segment source next), or SDN (segment destination next). These memory coprocessor instructions are defined in [HobSim 87].

Comments are preceded by •:

▽ MATMUL

[1] •start destination and create column step:
 [2] DEST SWW D'0' Δ R[T] ← 2 XSHIFT R[M],ZEROS
 [3] R[LINDX] ← NOP D'0' Δ
 FPEXEC Δ FPCTRL ← D'clear-accumulator-code'
 [4] LOOP1:
 R[RINDX] ← NOP D'-8' •initialize column index.
 [5] LOOP2:
 LEFT SRW R[LINDX] •start current row in left.
 [6] SBN LEFT Δ COUNTER ← NEGATE R[K] •initialize hardware counter.
 [7] CALL 'INNERPRODUCT' Δ R[RINDX] ← R[RINDX] PLUS D'8'
 [8] → BAD IF ERROR Δ SF R[RINDX] MINUS R[T] Δ DEST SDN ABUF[D0]
 [9] DEST SDN ABUF[D1]
 [10] DEST SDN ABUF[D2]
 [11] → LOOP2 IF → ZERO Δ DEST SDN ABUF[D3]
 [12] R[LCOUNT] ← SF R[LCOUNT] MINUS D'1' •SF = sample flags.
 [13] → LOOP1 IF → ZERO Δ R[LINDX] ← SAR[LEFT]
 [14] •SAR contains autoincremented row-offset.
 [15] → 0 Δ SR ← D'0' •clear status.
 [16] BAD: •processerror.
 ▽

Inner-product proceeds as a 10 microinstruction loop using pipelined multiply-accumulate:

▽ INNERPRODUCT

•64 bit data.

[1] COUNT ▽ FPCTRL ← 'multiply-accumulate-code'
 [2] LOOP:
 RIGHT SRW R[RP] •start right data stream.
 [3] SBN RIGHT
 [4] ABUF[L0] ← SSN LEFT
 [5] ABUF[L1] ← SSN LEFT
 [6] ABUF[L2] ← SSN LEFT
 [7] ABUF[L3] ← SSN LEFT Δ R[RP] ← R[RP] PLUS R[T]
 [8] ABUF[R0] ← SSN RIGHT
 [9] ABUF[R1] ← SSN RIGHT
 [10] ABUF[R2] ← SSN RIGHT
 [11] → LOOP IF → COUNT Δ FPEXEC Δ ABUF[R3] ← SSN RIGHT
 [12] FPEXEC Δ FPCTRL ← 'unload-and-clear-accumulator-code'
 [13] → 0 •delay 1 for output to catch up.
 ▽

Appendix C

MicroAPL Code for Matrix Multiplication on Hypermesh

MicroAPL techniques are used to demonstrate how ($m \times m = M$) matrix multiply may be broken into *outer* and *inner* routines for execution on the hypermesh [HobSim 87, HobTho 81]. The approach taken is to divide the operand matrices into $n \times n$ submatrices which fit the hypermesh exactly. The inner routine computes an $n \times n$ piece of the result, which requires, $K = m/n$, $n \times n$ matrix multiplies. The outer routine effectively sequences a smaller matrix multiply problem of size, $K \times K$, where each result element is computed by the inner routine.

Some simplifications are made for the sake of readability. It is assumed that operands are pipelined to a floating-point processor based upon AMD or Weitek chips. Floating-point data fifo-buffers for right and left arguments are FPR, and FPL. Floating-point instructions are placed in FPCTRL, while an execution is triggered by an FPEXEC. FPERROR and NETERROR are directly testable message (interrupt) flags. Each nonempty line of microcode takes one clock cycle, T .

The outer routine receives modulo matrix size in $R[SIZE]$ ($=K = m/n$). This is the actual matrix size, $R[MAT]$ ($=m$), divided by the network diameter, $R[NET]$ ($=n$). Register $R[BINDX]$ keeps track of columns in the right operand. $R[AINDX]$ keeps track of rows in the left operand. $R[STEP]$ holds the column step size in bytes. Data streams for $AMAT$, $BMAT$, and $CMAT$ are also passed to $MATMUL$ by an outer control routine. Memory coprocessor instructions are defined in [HobSim 87]. Network coprocessor instructions are defined in table 3-1. Comments are preceded by •:

▽ MATMUL

[1] • start result and clear A offset.
[2] CMAT SWW R[AINDX] ← NOP D'0'
[3] R[INC] ← D'8' • data size.
[4] • create column step:
[5] R[T] ← 2 XSHIFT R[SIZE],R[ZEROS] • mult by 8 for bytes.
[6] R[STEP] ← NOP R[T] Δ
 FPEXEC Δ FPCTRL ← D'clear-accumulator-code'
[7] LOOP1:
 R[BINDX] ← NOP D'0' • initialize column index.
[8] R[BITTR] ← R[SIZE] • initialize column counter.
[9] R[ATEMP] ← R[AINDX]
[10] R[BTEMP] ← R[BINDX]
[11] LOOP2:
 CALL 'INNERPRODUCT' Δ
 R[ATEMP] ← R[INC] PLUS AMAT SRW R[ATEMP]
[12] → EXIT IF ERROR Δ R[BITTR] ← SF R[BITTR] PLUS R[ONES] Δ
 CMAT SDN FPSN
[13] CMAT SDN FPSN Δ R[BINDX] ← R[BINDX] PLUS R[INC]
[14] CMAT SDN FPSN Δ R[BTEMP] ← R[BINDX]
[15] → LOOP2 IF → ZERO Δ CMAT SDN FPSN Δ R[ATEMP] ← R[AINDX]
[16] R[T] ← SF R[T] MINUS R[INC] • SF = sample flags.
[17] → LOOP1 IF → ZERO Δ R[AINDX] ← R[AINDX] PLUS R[STEP]
[18] → 0 Δ SR ← NOP D'0' • clear status.
[19] EXIT: • process errors...

▽

▽ **INNERPRODUCT**

```

[ 1] •start col memory stream:
[ 2] R[BTEMP]← R[STEP] PLUS BMAT SRW R[BTEMP]
[ 3] R[ITTR]← R[SIZE] Δ
           FPCTRL ← D'multiply-accumulate-code'
[ 4] SBN AMAT •fill row stream buffer.
[ 5] OLP:
           SBN BMAT •fill col stream buffer.
[ 6] NRBC •initilize row broadcast.
[ 7] NDN SSN AMAT •feed next 64 bit row word to net.
[ 8] NDN SSN AMAT
[ 9] NDN SSN AMAT
[10] NDN SSN AMAT
[11] NCBC •initialize col broadcast.
[12] NDN SSN BMAT
[13] NDN SSN BMAT
[14] NDN SSN BMAT
[15] NDN SSN BMAT Δ COUNTER ← NEGATE R[NET]
[16] NRWA D'4' Δ COUNT •setup alternate read.
[17] ILP:
           FPLN ← NSN
[18] FPLN ← NSN •left arg.
[19] FPLN ← NSN
[20] FPLN ← NSN
[21] FPRN ← NSN •right arg.
[22] FPRN ← NSN
[23] FPRN ← NSN
[24] → ILP IF ¬COUNT Δ FPEXEC Δ FPRN ← NSN
[25] → EXIT IF NETERROR
[26] → EXIT IF FPERROW Δ R[ITTR] ← SF R[ITTR] MINUS D'1'
[27] → OLP IF ¬ZERO Δ R[BTEMP] ← R[STEP] PLUS BMAT SRW R[BTEMP]
[28] FPEXEC Δ FPCTRL ← D'unload-and-clear-accumulator-code'
[29] → 0 •one cycle delay for output.
[30] EXIT: •error exit:
[31] SR ← NOP D'error-code'
▽

```

Ignoring constant overhead, the above prototype algorithm executes in the following number of cycles:

$$NP(n,K) = (((8 \times n + 15) \times K) + 10) \times K + 6 \times K$$

In [HobKaf 89], this is compared with matrix multiply on a single processor like NP16, as determined from similar microAPL code (Appendix A):

$$NP(1,m) = ((10 \times m + 9) \times m + 3) \times m$$

An important issue here is balancing between the system components. Several comments on this program are in order. At a glance, a further reduction in the execution time of this algorithm seems possible by employing a high bandwidth local bus. Having in mind the limitation on the number of pins available to a NC chip, increasing the local bus bandwidth will require the decrease of the communication links bandwidth. For simplicity of the discussion, suppose we can afford going from 16-bits to 32 or even 64, without any influence to the rest of the NC chip. A 64-bits local bus provides a one single cycle transaction between node processor and NC. However, the required time to perform the internode communication dictates a few waiting cycles (NOPs) to the node program. It is not difficult to see that in case of 64-bits wide local bus, the number of inserted NOPs, will not increase the efficiency of this sample algorithm. Also, in the ILP loop, the floating point unit may not be able to keep up with the incoming operands. In general, these issues are the matter of technology being used. But it should be pointed out that these issues must be considered in a hardware implementation.

Another issue here is the bandwidth, against possible network size, with the assumption of a single NC chip with fixed number of pins. Varying the bandwidth of the internode communication links in a NC will influence the network size supported by that chip. In order to support a larger network with the same NC chip, one has to decrease the links bandwidth. For example, reducing the bandwidth (bw) of the communication links from 8 to 4 bits (byte to nibble), allows the network size which can be supported by one NC chip to be doubled. The above prototype program, (with $bw=4$ and therefore $n'=2n$, $K'=\frac{K}{2}$), executes in the following number of cycles:

$$NP(n',K') = NP(2n, \frac{K}{2}) = (((8 \times 2n + 15 + 6) \times \frac{K}{2} + 10) \times \frac{K}{2} + 6) \times \frac{K}{2}$$

Where the first '6' comes from the number of NOPs inserted inside the OLP loop, in order to let NC to perform the required data communication. Comparing this results with the case $bw=8$, reveals that reducing the bandwidth, from 8 to 4, will offer an asymptotic

speedup of about the order of 4.4 to 4, for different matrix sizes (from 64x64 to 1Mx1M). The point is that increasing the number of communication links with less bandwidth, requires more control links. Therefore, a more complete analysis must consider the control links as well as the links bandwidth.

References

- [AgJap 86] D. P. Agrawal, V. K. Janakiram and G. C. Pathak.
Evaluating the Performance of Multicomputer Configurations.
IEEE Computer :pp. 23-37, May, 1986.
- [Akers 89] Sheldon B. Akers and B. Krishnamurthy.
A Group-Theoretic Model for Symmetric Interconnection Networks.
IEEE Transactions on Computers 38(4):pp. 555-566, Apr, 1989.
- [Akl 85] S. C. Akl.
Parallel Sorting Algorithms.
Academic Press Inc, 1985.
- [BaOlSo 89] K. K. Bogchi, O. Olsen, A. Christensen and L. Sorensen.
Simulation and Design of Message Routing Systems for Network of Transputers.
In *Proc. of the 1989 Eastern MultiConference- the 22nd Annual Simulation Symposium*, pages 69-80. Apr, 1989.
- [BeHeLa 87] Ramon Bevide, Enrique Herrada, J. Balcazar, and Jesus Labarta.
Optimized Mesh-Connected Networks for SIMD and MIMD Architectures.
* *1987 ACM* ():pp. 163-170, 1987.
- [Bell 86] C. Gordon Bell.
Expert Opinion.
IEEE Spectrum 23(1):pp. 36-38, Jan, 1986.
- [Berman 85] Francine Berman and Michael Goodrich.
PREP-P: A Mapping Preprocessor for CHiP Computers.
In *Proc. of the 1985 International Conference on Parallel Processing*, pages 731-733. Aug, 1985.
- [Bokhari 84] S. H. Bokhari.
Finding Maximum on an Array Processor with Global Bus.
IEEE Transactions on Computers c-33(2):pp. 133-139, Feb, 1984.
- [Bokhari 87] S. H. Bokhari.
Assignment Problems in Parallel and Distributed Computing.
Academic Publisher, 1987.
- [BoxMil 88] L. Boxer and R. Miller.
Dynamic Computational Geometry on Meshes and Hypercubes.
In *Proc. of the 1988 International Conference on Parallel Processing*, pages 323-330. Aug, 1988.

- [Brock 86] J. Dean Brock, A. R. Omondi, and D. A. Plaisted.
A Multiprocessor Architecture for Medium-Grain Parallelism.
In *Proc. the 6th International Conference on Distributed Computing Systems*, pages 167-174. May, 1986.
- [Carlson 85] David A. Carlson.
Performing Tree and Prefix Computations on Modified Mesh-Connected Parallel Computers.
In *Proc. of the 1985 International Conference on Parallel Processing*, pages 715-718. Aug, 1985.
- [ChuLeu 86] T. Chu and C. K. Leung.
Design of VLSI Asynchronous FIFO Queues for Packet Communication Networks.
In *Proc. of the 1986 International Conference on Parallel Processing*, pages 397-400. Aug, 1986.
- [Dally 87a] William J Dally.
A VLSI Architecture for Concurrent Data Structures.
Kluwer Academic Publisher, 1987.
- [Dally 87b] William J. Dally.
Wire-Efficient VLSI Multiprocessor Communication Networks.
1987 Stanford Conference on Advanced Research in VLSI.
The MIT Press, 1987, pages 391-415.
- [DalSei 86] William J. Dally and Charles L. Seitz.
The TORUS Routing Chip.
Distributed Computing (1):187-196, , 1986.
- [EaZaLa 89] Derek L. Eager, John Zohorjan and E. D. Lazowska.
Speedup Versus Efficiency in Parallel Systems.
IEEE Transactions on Computers 38(3):pp. 408-423, Mar, 1989.
- [FaLiNi 89] Z. Fang, X. Li and L. M. Ni.
On the Communication Complexity of Generalized 2-D Convolution on Array Processors.
IEEE Transactions on Computers 38(2):pp. 184-194, Feb, 1989.
- [Feng 81] T. Feng.
A Survey of Interconnection Networks.
IEEE Computer 20:, Dec, 1981.
- [FranDhar 86] M. A. Franklin and S. Dhar.
On Designing Interconnection Networks for Multiprocessors.
In *Proc. of the 1986 International Conference on Parallel Processing*, pages 208-213. Aug, 1986.

- [FriBa 87] Ophir Frieder and C. K. Baru.
Data Distribution and Query Scheduling Policies for a Cube-Connected Multicomputer System.
In Proc. Supercomputing 87, Vol 1, pages 376-388. , 1987.
- [Fuji 83] Richard M. Fujimoto.
VLSI Communication Components for Multicomputer Networks.
Technical Report TR-137, UCB/CSD Berkeley California, 1983.
- [GeAbGu 88] Edward F. Gehringer, Janne Abullarade and Michael H. Gulyan.
A Survey of Commercial Parallel Processors.
Computer Architecture News 16(4):pp. 75-107, Sept, 1988.
- [Gentleman 78] W. Morven Gentleman.
Some Complexity Results for Matrix Computations on Parallel Processors.
Journal of the Association for Computing Machinery 25(1):pp. 112-115, Jan, 1978.
- [GoodSeq 81] James R. Goodman and Carlo H. Sequin.
Hypertree: A Multiprocessor Interconnection Topology.
IEEE Transactions on Computers c-30(12):pp. 923-933, Dec, 1981.
- [Han 85] Y. Han.
A Family of Parallel Sorting Algorithms.
In Proc. of the 1985 International Conference on Parallel Processing, pages 851-853. Aug, 1985.
- [HarJum 87] David T. Harper and Robert Jump.
Vector Access Performance in Parallel Memories using a Skewed Storage Scheme.
IEEE Transactions on Computers c-36(12):pp. 1440-1449, Dec, 1987.
- [Hayes 86] John P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley and J. Palmer.
Architecture of a Hypercube Supercomputer.
In Proc. of the 1986 International Conference on Parallel Processing, pages 653-660. Aug, 1986.
- [Hillis 85] W. Daniel Hillis.
The Connection Machine.
The MIT Press, 1985.
- [HobGud 86] R. F. Hobson, J. Gudaities, and J. Thornburg.
A New machine Model for High-Level Language Interpretation.
In Proceedings. 19th Hawaii International Conference on System Science, IEEE Press, pages 132-139. Jan, 1986.

- [HobKaf 89] R. F. Hobson and Masoud R. Kafhesh.
A Mesh-like Array Processor with Fully Connected Rows and Columns.
In *IEEE 1989 Pacific Rim Conference on Communications, Computers
and Signal Processing*, pages 165-168. Jun, 1989.
- [HobSim 87] R. F. Hobson, R. W. Spilsbury, W. Strange, J. Hoskin and J. Simmons.
Design Considerations for a New Memory Controller Chip.
In *1987 Canadian VLSI Conference*, pages 149-154. Oct, 1987.
- [Hobson 87] Richard F. Hobson.
Microprogramming Tools in an APL Environment.
Technical Report TR87-14, LCCR, Computing Science Dept,
SFU, 1987.
- [HobTho 81] R. F. Hobson, P. Hannon and J. Thornburg.
Microprogramming with APL Syntax.
In *14th Annual Microprogramming Conference*, pages 131-139. Oct,
1981.
- [HoJess 81] R. W. Hockney and C. R. Jesshope.
Parallel Computers: Architecture, Programming, and Algorithms.
J. W. Arrow Smith Ltd, Bristol, 1981.
- [Hwang 89] K. Hwang, P. Tseng and Z. Kim.
An Orthogonal Multiprocessor for Parallel Scientific Computations.
IEEE Transactions on Computers 38(1):pp. 47-61, Jan, 1989.
- [JagKai 89] H. V. Jagadish and T. Kailath.
A Family of New Efficient Arrays for Matrix Multiplication.
IEEE Transactions on Computers 38(1):pp. 149-155, Jan, 1989.
- [JraHall 87] A. M. Jrad and R. W. Hall.
Orthogonal Fast Channels: An Enhanced MESH Architecture.
In *Proc. of the 1987-International Conference on Parallel Processing*,
pages 828-831. Aug, 1987.
- [Kale 86] L. V. Kale.
Optimal Communication Neighborhoods.
In *Proc. of the 1986 International Conference on Parallel Processing*,
pages 823-826. Aug, 1986.
- [KumRag 87] V. K. Prasanna Kumar and C. S. Raghavendra.
Array Processor with Multiple Broadcasting.
Journal of Parallel and Distributed Computing 4(3):pp. 173-190, Feb,
1987.
- [Lawrie 75] Duncan H. Lawrie.
Access and Alignment of Data in an Array Processor.
IEEE Transactions on Computers c-24(12):pp. 1145-1155, Dec, 1975.

- [LeeAgg 87] S. Y. Lee and J. K. Aggarwal.
A Mapping Strategy for Parallel Processing.
IEEE Transactions on Computers c-36(4):pp. 433-441, April, 1987.
- [Leighton 85] T. Leighton.
Tight Bounds on the Complexity of Parallel Sorting.
IEEE Transactions on Computers c-34(4):, Apr, 1985.
- [LiMar 87] H. Li and M. Maresca.
Polymorphic-TORUS Network.
In *Proc. of the 1987 International Conference on Parallel Processing*,
pages 411-414. Aug, 1987.
- [LinMol 85] T. Lin and Dan I. Moldovan.
Tradeoffs in Mapping Algorithms to Array Processors.
In *Proc. of the 1985 International Conference on Parallel Processing*,
pages 719-726. Aug, 1985.
- [LinMol 86] T. Lin and D. I. Moldovan.
M²-Mesh: An Augmented Mesh Architecture.
In *Proc. of the 1986 International Conference on Parallel Processing*,
pages 308-315. Aug, 1986.
- [Lu 88] M. Lu.
Solving Visibility Problems on MCC's.
In *Proc. of the 1988 International Conference on Parallel Processing*,
pages 95-102. Aug, 1988.
- [MarGaf 85] J. M. Marberg and E. Gafni.
Sorting and Selection in Multi-Channel Broadcast Networks.
In *Proc. of the 1985 International Conference on Parallel Processing*,
pages 846-850. Aug, 1985.
- [MilStou 86] Russ Miller and Quentin F. Stout.
Mesh Computer Algorithms for Line Segments and Simple Polygons.
In *Proc. of the 1986 International Conference on Parallel Processing*,
pages 282-285. Aug, 1986.
- [MilStou 89] Russ Miller and Q. F. Stout.
Mesh Computer Algorithms for Computational Geometry.
IEEE Transactions on Computers 38(3):pp: 321-340, Mar, 1989.
- [Modi 85] Jagoish J. Modi.
Parallel Algorithms and Matrix Computations.
Oxford Publications, 1985.
- [Murakami 88] K. Murakami, A. Fukuda, T. Sueyoshi and Shinji Tomita.
An Overview of the Kyushu University Reconfigurable Parallel
Processor.
Computer Architecture News 16(4):pp. 130-137, Sept, 1988.

- [NaBaAb 88] A. L. Narasimha Reddy, P. Manerjee, and Santosh G. Abraham.
I/O Embedding in Hypercubes.
In *Proc. of the 1988 International Conference on Parallel Processing*,
pages 331-338. Aug, 1988.
- [NasSah 79] David Nassimi and Sartaj Sahni.
Bitonic Sort on a Mesh-Connected Parallel Computer.
IEEE Transactions on Computers c-27(1):pp. 2-7, Jan, 1979.
- [NasSah 80] D. Nassimi and S. Sahni.
An Optimal Routing Algorithm for Mesh-Connected Parallel
Computers.
J. of the ACM 27(1):pp. 2-29, 1980.
- [NiKing 87] L. M. Ni, C. Ta. King and Philip Prins.
Parallel Algorithm Design Considerations for Hypercube
Multicomputers.
In *Proc. of the 1987 International Conference on Parallel Processing*,
pages 717-720. Aug, 1987.
- [Page 88] Ian Page.
Parallel Architectures and Computer Vision.
Oxford Science Publications, 1988.
- [RagKum 84] C. S. Raghavendra and V. K. Prasanna Kumar.
Permutations on ILLIAC IV Type Networks.
In *Proc. of the 1984 International Conference on Parallel Processing*,
pages 59-62. Aug, 1984.
- [ReedFuji 87] D. A. Reed and R. M. Fujimoto.
Multicomputer Networks Message-Based Parallel Processing.
The MIT Press Publications, 1987.
- [ReedGrun 87] D. A. Reed and D. C. Grundwald.
The Performance of Multicomputer Interconnection Network.
IEEE Computer 20:pp. 63-73, June, 1987.
- [ReevGut 89] Anthony P. Reeves and Maria Gutierrez.
On Measuring the Performance of a Massively Parallel Processor.
NASA Grant 5-403 (), , 89.
- [SchSen 89] Isaac D. Scherson and S. Sen.
Parallel Sorting in Two-Dimensional VLSI Models of Computation.
IEEE Transactions on Computers 38(2):pp. 238-249, Feb, 1989.
- [SchSha 86] Issac D. Scherson, S. Sen and Adi Shamir.
SHEAR SORT: A True Two-Dimensional Sorting Technique for VLSI
Networks.
In *Proc. of the 1986 International Conference on Parallel Processing*,
pages 903-908. Aug, 1986.

- [Seitz 84] Charles L. Seitz.
Concurrent VLSI Architectures.
IEEE Transactions on Computers c-33():pp. 1247-1265, Dec, 1984.
- [ShihIr 87] Y. Shih and Keki B. Iarni.
Large Scale Unification using a Mesh-Connected Array of Hardware Unifiers.
In *Proc. of the 1987 International Conference on Parallel Processing*, pages 787-794. Aug, 1987.
- [Shute 88] Malcolm J. Shute.
Fifth Generation Wafer Architecture.
Prentice Hall International Ltd, 1988.
- [Sinclair 88] James B. Sinclair.
Optimal Assignments in Broadcast Networks.
IEEE Transactions on Computers 37(5):pp. 521-531, May, 1988.
- [SonKin 88] J. Song and L. Kinney.
A Family of Parallel Sorting Algorithms.
In *Proc. of the 1988 International Conference on Parallel Processing*, pages 83-85. Aug, 1988.
- [Soucek 88] B. Soucek and M. Soucek.
Neural and Massively Parallel Computers- The Sixth Generation.
John Wiley and Sons Publication, 1988.
- [Stout 83] Quentin F. Stout.
Mesh-Connected Computers with Broadcasting.
IEEE Transactions on Computers c-32(9):pp. 826-830, Sept, 1983.
- [Stout 87] Quentin F. Stout.
Supporting Divide-and-Conquer Algorithms for Image Processing.
Journal of Parallel and Distributed Computing 4(3):pp. 95-120, Feb, 1987.
- [ThoKun 77] C. D. Thompson and H. T. Kung.
Sorting on a Mesh-Connected Parallel Computer.
Comm. of ACM 20(4):pp. 263-271, Apr, 1977.
- [Thompson 83] Clark D. Thompson.
The VLSI Complexity of Sorting.
IEEE Transactions on Computers c-32(12):pp. 1171-1184, Dec, 1983.
- [ThStSa 88] Charles P. Thacker, L. C. Stewart, and E. H. Satterthwaite.
Firefly: A Multiprocessor Workstation.
IEEE Transactions on Computers 37(8):pp. 909-920, Aug, 1988.

- [TuaPet 85] J. Tuazon, J. Peterson, M. Pniel, and D. Liberman.
Caltech/Jpl MARK II Hypercube Concurrent Processor.
In *Proc. of the 1985 International Conference on Parallel Processing*,
pages 666-673. Aug, 1985.
- [Ullman 84] Jeffrey, D. Ullman.
Computational Aspects of VLSI.
Computer Science Press, 1984.
- [Weitek 84] IEEE Floating Point Arithmetic with the WTL 1064/1065.
Weitek Corporation.
1984
- [Whelan 88] M. Whelan, G. Gao and T. Yum.
Optimal decomposition of Matrix Multiplication on Multiprocessor
Architectures.
In *Proc. of the 1988 International Conference on Parallel Processing*,
pages 181-185. Aug, 1988.
- [YaTaYa 82] H. Yasuura, N. Takagi and S. Yajima.
The Parallel Enumeration Sorting Scheme for VLSI.
IEEE Transactions on Computers c-31(12):pp. 1192-1201, Dec, 1982.
- [YounSing 88] H. Y. Youn and A. D. Singh.
A Highly Efficient Design for Reconfiguring the Processor Array in
VLSI.
In *Proc. of the 1988 International Conference on Parallel Processing*,
pages 375-382. Aug, 1988.