HIGH PERFORMANCE COMMUNICATIONS FOR HYPERCUBE MULTIPROCESSORS

by

Gregory Dean Buzzard

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1988

Doctoral Committee:

Associate Professor Trevor N. Mudge, Chairman
Professor Daniel E. Atkins III
Professor Ronald J. Lomax
Professor William R. Martin

RULES REGARDING THE USE OF

MICROFILMED DISSERTATIONS

Microfilmed or bound copies of doctoral dissertations submitted to The University of Michigan and made available through University Microfilms International or The University of Michigan are open for inspection, but they are to be used only with due regard for the rights of the author. Extensive copying of the dissertation or publication of material in excess of standard copyright limits, whether or not the dissertation has been copyrighted, must have been approved by the author as well as by the Dean of the Graduate School. Proper credit must be given to the author if any material from the dissertation is used in subsequent written or published work.

To my family and friends

ACKNOWLEDGEMENTS

I would like to thank the members of my dissertation committee: Dan Atkins, Ron Lomax, Bill Martin and Trevor Mudge for their helpful comments and constructive criticism. Special thanks go to my advisor and friend, Trevor Mudge, for his many useful comments and observations on my thesis dissertation, papers and reports, and on the graduate school experience, in general. I also appreciate the efforts of my fellow students, Don Winsor, Tarek Abdel-Rahman, Kunle Olukotun, Chuck Jerian and Mark Segal who have provided assistance in many stages of this work.

Financial support has been provided by a Kodak Ph.D. fellowship, the Advanced Computer Architecture Laboratory, the Robotics Research Laboratory and the Center for Information Technology Integration. The computer facilities of the latter three groups have been an invaluable resource in the development and production of this dissertation.

Finally, I'd like to thank my moral support group which includes many more people than I could list here. Special thanks, however, go to my Mother, for her continual support; to Patty Dvorak, for her encouragement in the early stages of this effort, and to Cheryl Huntington, for her support, encouragement and efforts to straighten out my many convoluted prepositional phrases. What a long strange trip it's been.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vii

LIST OF TABLES ix

CHAPTER

1 INTRODUCTION 1

1.1 Hypercube Connected Multiprocessors 3
1.1.1 Structural Definition 3
1.1.2 Advantageous Features 4
1.1.3 Limitations 5
1.2 Hypercube Communications 5
1.3 History and Background 8
1.4 Goal and Scope of this Dissertation 10
1.5 Major Contributions 11

2 CHARACTERIZATION OF COMMUNICATIONS 13

2.1 Classification of Communications 13
2.1.1 Types of Communication 14
2.1.2 Modes of Communication 15
2.1.3 Taxonomy of Modes and Types 17
2.2 Classification of Programs 18
2.3 Logical Communication Topologies 21
2.3.1 Efficient Homomorphic Embeddings 23
2.3.2 Inefficient Embeddings 26

3 NEAR NEIGHBOR COMMUNICATIONS 29

3.1 Existing Implementations 29
3.1.1 NCUBE – Vertex 30
3.1.2 Caltech – CrOS 35
3.1.3 Broadcast Algorithms 36

3.2 Points of Inefficiency 37
3.3 Alternatives within the Existing Architecture 39
3.4 Near Neighbor Execution Time Model 41
3.5 Experimental Results 44
3.5.1 Basic Parameters 44
3.5.2 Instrumented Programs 49
3.6 Communications Architecture Support 61
3.6.1 Modelled Results 62
3.7 Chapter Highlights 69

4 RANDOM COMMUNICATIONS 72

4.1 Software Communication Environment Issues 72
4.1.1 Communication Instruction Semantics 74
4.2 Message Transport Strategies 77
4.3 Existing Implementations 80
4.3.1 NCUBE 80
4.3.2 Intel iPSC/2 82
4.3.3 Ametek 2010 83
4.3.4 JPL Hyperswitch 85
4.4 Message Transport Comparisons 86
4.4.1 Investigative Approach 87
4.4.2 Comparitive Observations 90
4.5 Communication Processor Design Issues 102
4.6 Chapter Highlights 109

5 COMMUNICATIONS SYSTEM ARCHITECTURE 111

5.1 Architectural Overview 111
5.1.1 Communication System Queues 116
5.1.2 Message Initiation and Reception 120
5.2 Sequential Bottleneck at Source 124
5.3 Deadlock Avoidance 126
5.4 Implementation of Functional Units 127
5.4.1 Buffer RAM 128
5.4.2 Packet Buffer Units 129
5.4.3 Crossbar Switch 134
5.4.4 Node Output Port 136
5.4.5 Node Input Port 138
5.4.6 Arrival Tracking Unit 140
5.4.7 Control Unit 141
5.4.8 Flow Control Unit 142
5.5 Summary of Functional Unit Implementations 145
5.6 Buffer and Packet Sizes 145
5.7 Simulation Results 145
5.8 Chapter Highlights 153

6 CONCLUSION **154**

6.1 Future Work 156

BIBLIOGRAPHY **158**

LIST OF FIGURES

1.1 Hypercubes of dimension 0, 1, 2, and 3. 4

2.1 Embedding of a one-dimensional grid. 23
2.2 Embedding of a two-dimensional (4 × 4) grid. 24
2.3 Embedding of an odd dimensioned (4 × 3) grid. 25
2.4 Homomorphic embedding of a two-level tree. 26
2.5 Efficient embedding of a two-level tree. 27

3.1 Node-to-Node Communication (Vertex). 33
3.2 Global Send Routine. 36
3.3 Broadcast Spanning Tree. 37
3.4 Basic Ring Message Times. 46
3.5 Basic Ring Message Times. 47
3.6 Basic Ring Message Times. 47
3.7 Send/Receive Overhead Times. 48
3.8 Remote Data Requirements for Node A. 50
3.9 Linpack LU Factorization (Standard Vertex). 57
3.10 Linpack LU Factorization (Extended Vertex). 58
3.11 Coprocessor Interface Protocol. 63
3.12 Linpack LU Factorization (Proposed Scheme). 66
3.13 Linpack LU Factorization (Asynchronous Broadcast, Proposed Scheme Semantics). 71

4.1 Send header block. 74
4.2 Broadcast header block. 75
4.3 Near neighbor broadcast header block. 76
4.4 Receive header block. 78
4.5 Legend Explanation for Simulation Figures. 89
4.6 Message Time: Flooded Traffic, Len = 16, Dest = uni(). 90
4.7 Message Time: Flooded Traffic, Len = exp(512), Dest = uni(). 90
4.8 Message Time: Flooded Traffic, Len = nor(512, 256), Dest = uni(). . . . 91
4.9 Message Time: Flooded Traffic, Len = 8192, Dest = uni(). 91
4.10 Message Time: Flooded Traffic, Len = nor(8192, 2048), Dest = uni(). . . 92
4.11 Message Time: Flooded Traffic, Len = exp(512), Dest = dpf(0.2). 93
4.12 Message Time: Flooded Traffic, Len = exp(512), Dest = sl(2, 0.8). . . . 93
4.13 Message Time: Flooded Traffic, Len = 16, Dest = dpf(0.2). 94

4.14 Message Time: Flooded Traffic, Len = 16, Dest = sl(2, 0.8). 94
4.15 Message Time: Freq = exp(512), Len = exp(512), Dest = uni(). 96
4.16 Message Time: Freq = exp(1536), Len = exp(512), Dest = uni(). 97
4.17 Message Time: Freq = exp(2560), Len = exp(512), Dest = uni(). 97
4.18 Message Time: Freq = exp(8192), Len = exp(8192), Dest = uni(). 98
4.19 Message Time: Freq = exp(24486), Len = exp(8192), Dest = uni(). 98
4.20 Message Time: Freq = exp(40960), Len = exp(8192), Dest = uni(). 99
4.21 Message Time: Freq = exp(16), Len = exp(16), Dest = uni(). 99
4.22 Message Time: Freq = exp(48), Len = exp(16), Dest = uni(). 100
4.23 Message Time: Freq = exp(80), Len = exp(16), Dest = uni(). 100
4.24 Message Time: Freq = 1536, Len = nor(512,256), Dest = uni(). 102
4.25 Message Time: Freq = 2560, Len = nor(512,256), Dest = uni(). 103
4.26 Message Time: Freq = 48, Len = nor(16,8), Dest = uni(). 103
4.27 Message Time: Freq = 80, Len = nor(16,8), Dest = uni(). 104
4.28 Wormhole Routing Blockage. 106

5.1 Major Functional Units of Communication Processor. 112
5.2 Message Send Queues. 117
5.3 Message Packet Format. 127
5.4 Packet Buffer Unit. 131
5.5 Connectivity of Input and Output Ports. 135
5.6 A 1-of-8 Arbiter Constructed from a Tree of 1-of-2 Arbiters. 136
5.7 Node Output Port. 137
5.8 Node Input Port. 139
5.9 Arrival Tracking Unit. 141
5.10 Flow Control Circuit. 143
5.11 Spice Simulation of Flow Control Circuit. 144
5.12 Mean Message Times, Length = exp(512). 148
5.13 Mean Message Times, Length = exp(2048). 150
5.14 Message Times, Buffer/Packet Tradeoffs, Length = exp(512). 151
5.15 Message Times, Buffer/Packet Tradeoffs, Length = exp(2048). 152

LIST OF TABLES

2.1 Communication Mode Taxonomy. 18
2.2 Index of Definitions. 28

3.1 Basic Communication Performance. 45
3.2 NCUBE Send/Receive Call and Polling Overhead. 48
3.3 NCUBE Interrupt Driven Protocol Overhead. 49
3.4 Sobel Execution Profile, Standard Vertex. 53
3.5 Sobel Execution Profile, Extended Vertex. 56
3.6 Linpack Execution Profile, Extended Vertex. 59
3.7 Sobel Execution Profile, Proposed Scheme. 65
3.8 Linpack Execution Profile, (Proposed Scheme). 67
3.9 Linpack Performance for Standard and Extended Vertex and the Proposed Communication Scheme. 68
3.10 Linpack Effective Processor Utilization for Standard and Extended Vertex and the Proposed Communication Scheme. 69

5.1 Transistor Sizes (in microns) for Flow Control Unit. 144

CHAPTER 1

INTRODUCTION

Conventional supercomputers rely upon pipelined operations and ever faster basic components to achieve their high levels of performance. The incremental cost of faster components translates into super price tags, even for machines of modest supercomputer performance. Pipelining has been exploited close to its practical limit, and its benefits are restricted to vector based arithmetic computations. Thus, massively parallel supercomputers built by interconnecting hundreds, or even thousands, of small and inexpensive microprocessors are beginning to emerge as viable alternatives to the more conventional supercomputers mentioned above. The success of massively parallel supercomputers will hinge on the ability of the individual processors to cooperate productively on the execution of large programs. This cooperation requires both software tools to aid in the development of algorithms from which inherent parallelism can easily be extracted and an interconnection architecture that is sufficiently efficient so that it does not negate the benefits of parallel execution. The remainder of this section will describe the major distinguishing characteristics of parallel computer systems.

One of these characteristics is whether or not the individual processors share a common instruction stream. Systems with processors that do share a common instruction stream are classified as SIMD, single instruction stream multiple data stream, computers [Fly66]. Systems in which individual processors fetch and execute independent instruction streams are classified as MIMD, multiple instruction stream multiple data stream.

MIMD machines allow several different applications to be executed simultaneously by disjoint groups of processors. This can be of great practical importance because it allows several users to verifying and debug their individual codes simultaneously. The simultaneous execution of independent sections of code may also be required by certain classes of applications where the actions of individual processors are influenced by external events. Real-time simulations comprise one such class.

Another important distinguishing characteristic is the memory organization. Shared-memory systems, in which a global memory is accessible by all processors, are often considered easier to program than distributed-memory systems. However, they require the use of synchronization protocols to prevent simultaneous access by more than one processor. They are also susceptible to large system-wide bottlenecks as processors repeatedly compete for access to common memory units. Distributed-memory systems provide local private memory for each processor. Information is exchanged between processors by sending and receiving messages. Frequently, this message passing can be controlled in an orderly fashion to minimize the competition for shared resources such as communication links and buffers.

The choice of interconnection topology for distributed-memory multiprocessors involves satisfying many requirements that frequently conflict. As a simple example, at one extreme we can consider a unidirectional ring interconnection in which each processor has only one inbound and one outbound communication link. Routing is simple, but the average distance between processors grows linearly with N, the number of processors. Clearly this is undesirable for large N. At the other extreme is a fully connected network in which each processor has a direct connection to every other processor. While this reduces the average distance between processors to one and retains simple routing, it also requires that the number of (bidirectional) communication links grow as $\frac{N}{2}(N-1)$ and the number of processor connections (fanout) as $N-1$. It is clearly impractical to build a fully connected computer for large N. As we show below, a topological structure known as the binary hypercube provides a reasonable compromise to these conflicting

requirements.

1.1 Hypercube Connected Multiprocessors

In the hypercube interconnection topology, both the average distance between processors and the complexity of the connections between processors grow logarithmically with the number of processors in the system. Hypercubes also possess mathematical properties that further contribute to their usefulness in parallel computer systems. The properties, features, and limitations of hypercubes will be examined in the following sections.

1.1.1 Structural Definition

An n-dimensional hypercube computer, also known as a binary n-cube, is a multiprocessor comprising $N = 2^n$ processing nodes interconnected as an n-dimensional binary cube. Each processing node, P, corresponds to a vertex of the cube, and contains its own CPU and local memory. P also has direct communication links to n other (neighbor) processors. The binary representation of the addresses of each of the 2^n processors will differ from that of their neighbors in exactly one bit position. This is shown for hypercubes of dimension $n \leq 3$ in Fig. 1.1. The following recursive procedure describes the construction of a hypercube starting with a 1-cube (two nodes). Given a 1-cube, label one of the nodes with a 0, the other with a 1. In general, an n-dimensional cube is constructed from two $(n-1)$-dimensional cubes by prefixing the labels in one of the $(n-1)$-cubes (the *zero* cube) with a 0 and the labels in the other (the *one* cube) with a 1; then each of the nodes in the *one* cube is connected to its counterpart in the *zero* cube. Thus, the node $P_{0b_{n-2}\cdots b_0}$ (subscripts indicate addresses) is connected to node $P_{1b_{n-2}\cdots b_0}$, for each address substring $b_{n-2}\cdots b_0$, where each $b \in \{0,1\}$. All hypercubes of higher dimension can be constructed and labelled by repeated application of this procedure. Each dimension of a hypercube has an associated axis that is labelled as follows: Node $P_{b_{n-1}\cdots b_{i+1}0b_{i-1}\cdots b_0}$ is connected to node $P_{b_{n-1}\cdots b_{i+1}1b_{i-1}\cdots b_0}$ by an edge that is in the direction of the ith axis, where $i \in \{0, n-1\}$ is the position of the differing bit in the respective node addresses.

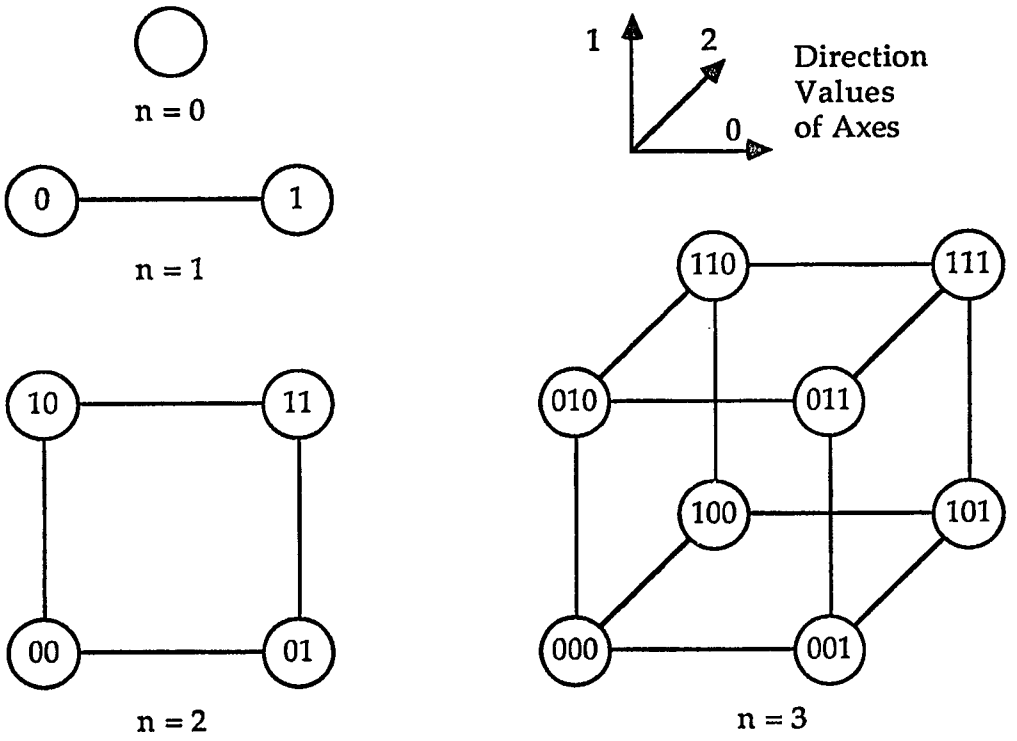


Figure 1.1. Hypercubes of dimension 0, 1, 2, and 3.

1.1.2 Advantageous Features

There are several features of the hypercube interconnection scheme that are useful for supporting parallel computation. The fanout of each processor scales logarithmically (as $\log_2 N$) with the total number of processors. The structure of hypercubes allows a large cube to be easily divided into smaller sub-cubes, and node adjacency is preserved within these sub-cubes. An important consequence is that the view of the system that is seen by an application program is isotropic. That is to say, application programs have the same view of the hypercube environment, regardless of the sub-cube in which the program is executed. As an example, a machine with $N = 64$ may be divided into a 5-cube, composed of processors $P_{0b_4b_3b_2b_1b_0}$; and two 4-cubes composed of processors $P_{10b_3b_2b_1b_0}$ and $P_{11b_3b_2b_1b_0}$. Programs that are assigned to a subcube of dimension m ($m \leq n$) need to use only m bits to address other nodes within the subcube. Thus, a dimension 4 program may be executed in either the $P_{10b_3b_2b_1b_0}$ or $P_{11b_3b_2b_1b_0}$ subcube without modification. Many

other interconnection structures may be embedded into hypercubes [NCU85]. Two important examples are meshes (of any dimension) and binary trees [BI85]. Both of these structures maintain their node adjacency when embedded into hypercubes. The communication structures employed by many parallel sorting and FFT algorithms also map directly [Swa86,Wal87,SZ87,Wag87]. Communications between random nodes travel a maximum distance of $\log_2 N$, and an average distance (as $n \rightarrow \infty$) of $\frac{1}{2}\log_2 N$.

1.1.3 Limitations

One of the more notable limitations of the hypercube interconnection scheme arises from the fact that the fanout of communication links, despite its modest logarithmic growth, is not fixed. This implies that hypercube systems must initially be designed to accommodate growth up to a maximum limit that is fixed by some design parameter such as chip pin limitations or the width of either the backplane or printed circuit board wiring channels. In practical systems, however, the judicious use of modular design techniques and clever engineering may mitigate many of these problems.

Another limitation is the potential for underutilized communication paths. This problem is noticeable when low dimensional problems are embedded into large dimension hypercubes; for example, using an $N = 1024$ processor 10-cube to solve a 32×32 mesh problem. In such a problem, at most $\frac{4}{10}$ths of the maximum channel capacity can be used since the fanout of a two dimensional mesh is 4, irrespective of its size. However, this underutilization constitutes a real limitation only when the maximum useable channel capacity is insufficient to meet processing demands. This latter point is addressed in more detail in Sec. 5.2.

1.2 Hypercube Communications

The performance of communications is crucial to the success of massively parallel distributed-memory computers. Aside from the data sharing necessary for productive cooperation during the computational phase of parallel program execution, the communi-

cation architecture must also provide for the efficient loading of programs and handling of I/0 with external devices. Evaluations of communication performance can be made in several contexts. At an abstract level, one can define the ratio of time required to communicate a simple data item to a neighboring processor versus the time required to calculate it. This simple architectural parameter can then be used to determine, in a general sense, the suitability of various algorithms for a given machine architecture. Definitions of this type are given in [FK86a], as t_{comm} and t_{calc}, and the use of these metrics is described in [FK86b]. The value of t_{calc} is defined as the time to execute a typical arithmetic instruction, and t_{comm} is defined as the transfer time of one word to a neighboring node. In [FK86a] t_{calc} is defined as the time to execute a Fortran double precision floating point statement of the form x = a * b . Though these definitions are not precise, they can be employed to make some useful generalizations about parallel programming environments. In particular, we classify the granularity of communications as follows:

		t_{comm}/t_{calc}	$>$	1000	as coarse grained parallelism
1000	$\geq$	t_{comm}/t_{calc}	$>$	100	as medium grained parallelism
100	$\geq$	t_{comm}/t_{calc}	$>$	10	as fine grained parallelism
10	$\geq$	t_{comm}/t_{calc}			as very fine grained parallelism

The t_{comm}/t_{calc} ratios for the commercially available first generation hypercube multiprocessors lie in the medium grained parallelism range. Even at this rather large level of granularity a large class of algorithms are being efficiently executed on existing machines[Fox85], and as architects are able to lower the t_{comm}/t_{calc} ratio this class will grow.

A basis for more detailed communication performance evaluations is provided by the following measurements. *Latency* is a measure of the elapsed wall clock time between the sending of a message on one node and the reception of its first byte on the destination node. The *sustained bandwidth* indicates the rate of message delivery once the first byte

has arrived. *Sending overhead* is the time consumed by a process making a call to the run-time system to send a message. *Receiving overhead* is the time consumed by a process making a call to the run-time system to receive a message. This neglects any waiting due to the message not having yet been delivered to the receiving node. The relative importance of these different performance measurements will vary, depending on the granularity, frequency and predictability of communications in the program.

When the apparent granularity, that is, the communication overhead that is noticeable to a user process, drops to the fine or very fine range the potential exists for providing the appearance of a shared-memory space to application programs. Programs may then communicate through the shared-memory in addition to, or instead of, communicating via message passing. The memory in a virtual shared-memory space may be either real or virtual. Support for virtual memory (VM) is costly. VM systems require additional resources such as a swapping device and an address translation mechanism. Additionally, the handling of page faults can be very time consuming (typical times are reported in the following paragraph). Virtual memory systems are almost always used in conjunction with multiprocessing to minimize the effect of the time consumed by a page fault. Multiprocessing may also reduce the apparent granularity of communications by scheduling another process, if possible, to execute while the previous process waits for the completion of a remote communication. However, multiprocessing on a distributed-memory multiprocessor can be much more problematic than on a shared-memory machine. For instance, process switches and swapping activity need to be carefully coordinated across all cooperating nodes so that communication thrashing is avoided. The prefetching of remote data may provide an alternative approach to minimizing the effects of page faults and communication delays. It has been shown that prefetching remote data is more effective than caching data on demand (which is a standard VM technique) for numeric codes in shared-memory parallel programs [LYL87,MP85]. As communication latencies drop, similar results will be realized in hypercube systems.

The communication requirements for supporting a single VM space that is shared

across all nodes may be put in perspective by comparing the impact of communication delays that are realized in hypercubes communications with the time required for common virtual memory operations such as servicing page faults. As an example, we have measured the page fault times on a Sun 3/260, a high performance computation/disk server that runs Sun's implementation of the Unix 4.3bsd operating system. These times are expressed in terms of the amount of time required to add a vector element and a scalar and place the result in a scalar memory location. The real time consumed by a page fault is 18,500 VSOPs (vector sum operations). Most of this time (86% or 15,900 VSOPs) is spent waiting for the disk. We will show in Sec. 5.7 that internode communication latencies can be reduced to a point where the time consumed communicating between two arbitrary nodes in the hypercube is insignificant when compared to the latency time of a disk access.

The reduction of granularity also increases the number of problems which massively parallel machines can execute with reasonable efficiency. For most problems there is a limit on the extent to which they can be parallelized. This limit is typically a function of the overhead cost associated with communications that are necessary to support the desired level of parallelism. Decreasing the communication costs frequently leads to an increase in the number of nodes that may be effectively used to solve the problem. This, in turn, lowers the minimum possible overall execution time and opens up the possibility of providing a cost effective and time efficient execution environment for computationally demanding problems.

1.3 History and Background

One of the earliest multicomputer design proposals to employ a hypercube interconnection scheme appeared in a 1963 paper by Squire and Palais from the University of Michigan [SPCC]. They provided a paper design of a 4096 node hypercube connected multiprocessor. Several other proposals appeared in the mid-1970's, two of the more notable were the CHOPP architecture of Sullivan, Bashkow and Klappholz at Columbia

University [SB77] and a machine comprising 256 Intel 8008 processors that was offered commercially by IMS Associates. Unfortunately, IMS ceased to exist before building their first machine. The CHOPP proposal called for a machine that was physically configured as a distributed-memory hypercube, but which provided the virtual appearance of a shared-memory multiprocessor to applications software. In a companion paper [SBK77] they describe a distributed operating system to control their multiprocessor system without relying on a single master processor. However, it was not until high density and low cost VLSI processors, memories, and communications chips became widely available in the early 1980's that production of large scale hypercube multiprocessors became economically feasible.

The first such machine built was the Cosmic Cube [Sei85]. This machine was the product of research that began in 1978 at the California Institute of Technology. In 1983 the first production size Cosmic Cube became operational. It consisted of 64 nodes, with each node containing an Intel 8086/8087 processor pair, 128 K-bytes of memory, and six bit-serial communication links. Over the next few years, Caltech, in conjunction with NASA's Jet Propulsion Laboratory (JPL), built a series of progressively more powerful machines named the Mark I, Mark II, and Mark III [TPPL85,PTLP85].

Three commercial manufacturers: Intel, Ametek and NCUBE, introduced hypercube computer systems in 1985. This first generation of commercial machines consisted of nodes with CPUs of about 2 MIPS of useable performance, 128–1024 K-bytes of memory, and communication links with a bandwidth of about 1 M-byte per second [NCU85,iPS85,Hyp]. In conjunction with the link bandwidth, the operating systems on these machines supports levels of parallelism that ranged from coarse grained to about midway into the medium grained range. Floating Point Systems (FPS) offered a first generation hypercube [BFHP87] that was based on the Inmos Transputer chip. However, for a variety of reasons, not all of which were technical, the machines never became viable enough to make FPS a major hypercube vendor.

Two second generation commercial distributed-memory multiprocessors have been an-

nounced by Intel and Ametek in 1988. The Intel iPSC/2 is presently available [Nug88]; however, it has not been in the field long enough for any meaningful performance benchmark results to be available. The Ametek Series 2010 has not yet been shipped [Ame87a]. In their second generation machine, Ametek has abandoned the hypercube interconnection structure for that of a two-dimensional mesh with byte-wide communication links. The design of the Ametek 2010 was influenced to a large extent by the thesis research of William Dally at Caltech [Dal86b]. Dally's thesis advisor, Charles Seitz, was the principal consultant on the design of the Ametek 2010. It is widely expected that communications performance of both these machines will lie near the fine grained end of the parallelism range.

At present, JPL is upgrading the capabilities of the Mark III. They have recently integrated the Weitek floating point unit into the nodes of the Mark III. Another upgrade that is being worked on presently is the hyperswitch network (HSN) communication chip [CMP87a,CMP*87b,CMP*88]. The hyperswitch allows the user to choose from several different communication modes to route messages through the hypercube interconnection network of the Mark III in a time efficient manner. The HSN chip is discussed in more detail in Sec. 4.3.4.

Another machine that played a part in the development of hypercube technology is the Connection Machine from Thinking Machines Corporation [Hil85]. The Connection Machine differs from the mainstream hypercube machines because it is structured as a massive (65,536 node) collection of simple (bit-wide) processors that execute a single instruction stream. Nonetheless, it has played an important role, especially in the development of massively parallel algorithms [HS86].

1.4 Goal and Scope of this Dissertation

The goal of this thesis is to analyze communications on an existing hypercube system to identify critical design issues, and then use the results of these analyses to specify a *paper design* for a communication architecture to support fine grained parallelism for

MIMD hypercube distributed-memory multiprocessors. This investigation focuses on two main design objectives. The first objective is to reduce the actual granularity of parallelism, that is, minimize the actual cost of communications. The second objective is to search for ways to overlap communications with other useful work as much as possible. The extent of design considerations is bounded at the level of hardware functional unit interconnections on one end, and run-time system issues at the other.

1.5 Major Contributions

The following are the major contributions of this thesis:

- A taxonomy of communications paradigms for parallel programming is developed in Chapter 2. The concept of C-deterministic programs is also developed. In a C-deterministic program certain communication activities and parameters are known slightly in advance of their occurrence. It is later shown that C-deterministic programs may be speeded up considerably on existing first generation machines.

- In Chapter 3 we describe a new node operating system communications module for use by C-deterministic programs. Use of this new communication system leads to significant improvements in communication system performance which is highlighted by a total program execution time improvement of about 1.33 on the Linpack code.

- In Chapter 3 we also profile two broadly representative numeric codes to determine precisely where the execution time is spent. A model is then developed and used to predict the effects of various architectural changes.

- In Chapter 4 a new message routing scheme is introduced. This new routing scheme increases performance in a wide variety of programs in addition to those that are C-deterministic.

- An architecture for our new routing scheme is described in Chapter 5. The results of simulations that predict its performance relative to other existing schemes are also presented. These results indicate improvements in communication latency that range from 25% to 1000%.

CHAPTER 2

CHARACTERIZATION OF COMMUNICATIONS

In this chapter we develop a taxonomy for hypercube communications and provide a classification for two major types of parallel programs. The relations within the communication taxonomy and between that taxonomy and the program classifications are explained. Finally, the embedding of common communication structures onto a hypercube is discussed. Some of the terms and criteria that we have used for these classifications have appeared independently in other works or have been used without precise definition. The goal in developing the taxonomy is to develop a consistent and complete group of definitions to facilitate a clear presentation throughout this thesis. A key concept developed in this chapter is that of a C-deterministic program phase. A large potential exists for increasing the performance of programs with C-deterministic execution phases on existing hypercube systems.

2.1 Classification of Communications

Hypercube computers are typically comprised of three major units, the hypercube processor array, the array I/O system, and the host system. The CPUs and memories that are interconnected by the hypercube network are referred to as array nodes. The I/O system provides connections from the array nodes to various peripheral devices such as sensors, graphic display units or disks. The host system serves as a master controller for the overall operation of the hypercube system. It usually provides the user interface as

well. Communications in hypercube multiprocessors can be classified in three ways:

- By the type of processors involved. That is, hypercube array (or node) processors, I/O processors, or the host processor.
- By the relative positions of the communicating processors.
- By the synchrony of the communicating processes.

The first of this group will be referred to as the type of communication and a combination of the latter two will be referred to as the communication mode.

2.1.1 Types of Communication

Array-processor-to-array-processor communications are primarily used to pass intermediate calculation results among array nodes. They may also pass system administrative messages. These administrative messages may convey such information as preferred message routes or processor utilization statistics. At least one group [PR87] proposes storing pages of program text (for programs that use the same code on all nodes) throughout the array; each page would be assigned a *home* node from which it would not be discarded (paged out). This would enable programs that would not otherwise fit in the available memory on a single node to have their code demand-paged (via intra-array communications) from other array nodes as needed. The interconnection topology for these communications is a hypercube.

Array-processor-to-I/O-processor communications are used to load initial program code and data, and to collect final program results. This type of communication is also used for accessing I/O devices. It may also be used to support virtual memory or paging capabilities. The interconnection topology for these communications is typically a tree rooted at the I/O processor.

I/O-processor-to-I/O-processor communications are not necessary. However, they can add to the flexibility and availability of the I/O subsystem. These communications can be used to route I/O requests away from I/O subsystems that do not have a desired

I/O resource to one that does, thus preserving the array processors isotropic view of the system. They can also be used to route out-of-band messages between array nodes. A debugger message is an example of an out-of-band message. Such messages are not part of the normal stream of messages that an application program would expect to see flowing between array nodes.

I/O-processor-to-host communications are primarily used for initiating program loads and for collecting any results that are to be presented to the external user interface or saved in the mass storage system. This type of communication may also be used by the host to query the status of the I/O or array subsystems. The logical interconnection topology is typically that of a tree rooted at the host.

2.1.2 Modes of Communication

Communication modes can be classified both spatially and temporally. The spatial classification is described first.

Spatial Mode

Spatially, two communications groups are distinguished: near neighbor and random. They are defined as follows.

Definition 1 (Near Neighbor – NN) *NN communications are those where, from the point of view of the application program, the destination node will always lie at a Hamming distance of one from the source in the allocated subcube. That is, the binary representation of the source and destination node differ in only one bit and, as consequently, the two nodes will have a direct connection to each other.*

Definition 2 (Random – RA) *RA communications are those where, from the point of view of the application program, the destination node may lie at a Hamming distance of greater than one from the source. In this case, the two communicating nodes are not likely to have a direct connection to each other and, hence, must rely upon other nodes to properly forward their messages.*

The term *random* is used because the destinations of such communications are frequently unknown prior to program execution and posses no apparent pattern.

Finally, a special case may exist when common data is to be sent from the same source to multiple destinations. The information can be broadcast simultaneously to all destinations, thus incurring a communication startup overhead only once for each sending node. Broadcasts (BC) are defined for both NN and RA communications as follows:

Definition 3 (Near Neighbor Broadcast – NNBC) *NNBC communications are those where, from the point of view of the application program, the set of destination nodes all lie at a Hamming distance of one from the source, in the allocated subcube.*

Definition 4 (Random Broadcast – BC) *BC communications are those where, from the point of view of the application program, the destination nodes are all of the remaining nodes in the allocated subcube.*

Temporal Mode

Communications may also be referred to as being either synchronous or asynchronous. The sending process does not immediately await a response from the recipient in asynchronous communications. Conversely, in synchronous communications, the sending process immediately begins waiting for either return values (in the case of remote procedure call semantics) or confirmation that the message was correctly received (in the general case). To provide a more detailed distinction in multiprocessing systems, the synchronous class may be further divided into process synchronous and processor synchronous communications [Spe81]. The distinction is that in the process synchronous case the processor may perform a context switch and begin to execute another available process. In the processor synchronous case, no such context switch occurs and all processing is suspended until the communication is completed. In the uniprocessing environment of current hypercube systems, the process synchronous case is the same as the processor synchronous case.

Asynchronous communications provide more flexibility than synchronous communications. In fact, synchronous communications can be readily constructed from asynchronous primitives by requiring that all send operations be immediately followed by receive operations that block, awaiting a possibly empty set of return values. Conversely, asynchronous communications may be constructed from synchronous primitives only when some form of multiprocessing is available to immediately handle and acknowledge an incoming message. Even then, this case incurs the additional expense of the return communication and context switching overhead.

The chief advantage of synchronous communication semantics is that programs that use them are more easily understood than those that rely on asynchronous communications. Consequently, most current parallel procedural programming languages provide synchronous communication primitives. However, these languages are also typically designed with the multiple code multiple data (MCMD) programming model in mind. Unfortunately, as discussed in Sec. 2.2, this is not the predominant programming paradigm for massively parallel computers. Thus, there is still an existing need for improved communication support in languages that are intended for massively parallel machines.

The individual send and receive operations in an asynchronous communication environment may be classified as either *blocking* or *non-blocking*. A blocking send operation will not release until the last byte of data has proceeded onto the communication link. Non-blocking sends release immediately. Blocking receive operations will not release until the awaited message has been delivered to the specified address. Non-blocking receive operations simply check if the designated message has arrived. The semantics and implications of these operations are discussed in Chapter 3.

2.1.3 Taxonomy of Modes and Types

The allowable spatial communication modes for each communication type, as viewed by the application program, are given in Table 2.1. Only the array-processor-to-array-processor and I/O-processor-to-I/O-processor communication types allow all of the possi-

Source Node Type	Destination Node Type: array	I/O	host
array	RA, BC, NN, NNBC	NN	Do Not Directly Communicate
I/O	NN, NNBC	RA, BC, NN, NNBC	NN
host	Do Not Directly Communicate	NN, NNBC	Do Not Directly Communicate

Table 2.1. Communication Mode Taxonomy.

ble communication modes. Array-processor-to-I/O-processor communication topologies can be viewed as a single level tree rooted at the I/O processor. All of these cases are inherently NN since the tree has only a single level; also, broadcast mode is available outbound from the root (i.e., from the I/O processor). The I/O-processor-to-host communication topology is also a single level tree. In this case, the tree is rooted at the host. The communication modes are similar to those for array-processor-to-I/O-processor communications. The other possible source/destination pairs do not directly communicate with each other. This taxonomy assumes a system configuration like that of the NCUBE, variations may exist for systems that are configured differently. However, many systems are moving toward such a configuration so that they may provide greater I/O support to the array processors.

2.2 Classification of Programs

Parallel programs can be broadly grouped into two categories based on the similarity of the code being executed on processors throughout the computer. These groups are defined below,

Definition 5 (Single Code Multiple Data – SCMD) *In SCMD programs all processors in the same allocated subcube execute the same code. The data that is processed by each node is different. Differing paths of execution may be taken through the code due to data*

dependent branches.

We will assume that all nodes executing SCMD programs are executing the same section of code at about the same point in time. Throughout this dissertation any time that this assumption does not hold it will be explicitly pointed out, and such SCMD programs will be referred to as *loosely coupled.*

Definition 6 (Multiple Code Multiple Data – MCMD) *In MCMD programs different codes are executed on each processor. The data in each node may also be different. The number of different copies of code will be referred to as the code multiplicity.*

The SCMD programming model is by far the most prevalent for general problem solving on massively parallel machines. This is to be expected since the primary reason for building large scale hypercube multiprocessors is to dedicate a large number of processors to work together to solve a single large problem. Intended usage aside, the independent programming of several hundreds or thousands of individual processors is, presently, an intractable proposition and is likely to remain so for the foreseeable future. In those few cases where there are different copies of code loaded on the processors within an allocated subcube (i.e., MCMD), the code multiplicity is small and the SCMD model still plays a dominant role. The overall structure usually comprises a small number of distinct SCMD groups each performing a specific set of tasks. These SCMD groups are then interconnected to accomplish the overall programming goal, An example of such a system is given in [Bra86].

To date, the overwhelming majority of programs written for hypercube machines consist of a sequence of execution phases during which only one mode of inter-array processor communication is active [Hea87]. These programs are typically decomposed into program phases where the nodes communicate among themselves (either globally or within a group of their immediate neighbors) and perform calculations on local data and/or data received from other nodes. These steps are then repeated as necessary until the final results are assimilated and the program is terminated. Each successive communication

phase may employ a different communication mode, but the different communication phases will not overlap as long as the program execution on the different processors remain approximately synchronized. This approximate synchronization is inherent in any algorithm in with program phases that employ NN, NNBC, or BC communication modes that involve all of the nodes. Additionally, the run-time system can provide an explicit synchronization service to prevent communication phases on communicating processors from overlapping for any SCMD programs in which approximate synchronization is not maintained (i.e., loosely coupled SCMD programs). The points that were enumerated above lead us to make the following thesis statement:

Thesis Statement 1 *A substantial performance increase is possible for a large and important class of hypercube programs by exploiting the knowledge of the communication structure that is inherent in the algorithm.*

We substantiate this statement by defining the class of programs for which substantial performance increases are possible. Communication primitives that lead to this performance increase are then described in detail, along with their use, operation and performance in Chapter 3.

Definition 7 (C-deterministic) *A program phase is C-deterministic when each processor knows the size of the messages that it is about to receive on each of its channels. It is not required to know the order in which the messages arrive.*

For example, consider the processors P_i and P_j, where P_i and P_j are two adjacent processors in the allocated subcube. Define $M_{i,j} = \langle m_{i,j,k} \mid 1 \leq k \leq L_{i,j} \rangle$ to be the ordered set of messages (m) sent from node P_i to P_j (where $L_{i,j}$ is the total number of messages sent from P_i to P_j). The program phase is C-deterministic if, after receiving $m_{i,j,k}$, every P_j knows the size of at least $m_{i,j,k+1}$ for each $M_{i,j}$. In other words, the arrival of an incoming message must always be expected by the receiver—the source node may provide indication of a subsequent message delivery to the receiver by embedding

such information in an earlier message. The order of arrival of messages from different sources need not be known in advance by the receiver.

A large number of NN, NNBC and BC program phases fall into this category, including most numeric, search, and sort phases. Most communication phases that employ RA communications will not be C-deterministic. This is because RA communications often pass through intermediate nodes *en route* to their destination. These intermediate nodes will not, in general, be able to anticipate the order of messages for which they are not the final destination.

In Chapter 3 it will be shown that the communications performance of NN, NNBC, and BC program phases that are also C-deterministic can be significantly improved by exploiting this knowledge.

Thesis Statement 2 *The performance of RA or non-C-deterministic programs may be improved significantly over the performance that is possible on existing systems with suitable modifications to the communication architecture.*

This point is examined further in Chapter 4. We propose a new communication architecture that should lead to substantially performance increases in Chapters 4 and 5.

2.3 Logical Communication Topologies

One of the most useful features of the hypercube interconnection topology is that several other common topologies map into it in an efficient manner. It is desirable for these mappings to be *homomorphic* because homomorphic mappings preserve node adjacency. The following definition of a homomorphic mapping is taken from [Har69].

Definition 8 (homomorphic mapping) *A homomorphic mapping, h, of graph G into G' can be considered as a function $h : V(G) \rightarrow V(G')$ (where $V(G)$ is the node set of graph G) such that if nodes u and v are adjacent in G, then $h(u)$ and $h(v)$ are adjacent in G'.*

However, we will show in Sec. 2.3.2 that not all common topologies have both homomorphic and efficient mappings into hypercubes. As a practical matter, many hypercube

mappings use some version of the Gray code.

Definition 9 (Gray code) *A Gray code, $G(x)$, is a one-to-one mapping between integers such that for two consecutive integers in the domain, the binary representation of their corresponding values in the range will differ in exactly one bit position. The domain of integers is assumed to be finite and the largest and smallest integers in the domain are consecutive. The inverse Gray code function will be denoted by $F(x)$, thus $F(G(x)) = x$.*

Intuitively, one can see that the Gray code is important to hypercube mapping by noting that processors whose binary representations differ in exactly one bit position are physically connected to each other. Many unique Gray code mappings are possible. The specific Gray code used in the examples in the following sections is given by the following: let y be the logical right shift of x by one bit position; $G(x)$ is then the exclusive or of x and y.

Two figures of merit for the embedding of a graph G into another graph G' via a function $h : V(G) \rightarrow V(G')$ are defined below.

Definition 10 (Dilation) *The dilation, D, of a graph embedding function is the maximum distance separating two nodes that were adjacent in the original graph: $D = \max(d(h(u), h(v)))$ for all (u, v), where (u, v) are edges in G and $d(x, y)$ is the distance between nodes x and y in G'.*

Definition 11 (Expansion) *The expansion, E, of a mapping is a measure of the efficiency in terms of the number nodes (graph vertices) required to achieve the desired mapping:*

$$E = \frac{\#\ of\ vertices\ in\ G'}{\#\ of\ vertices\ in\ G}$$

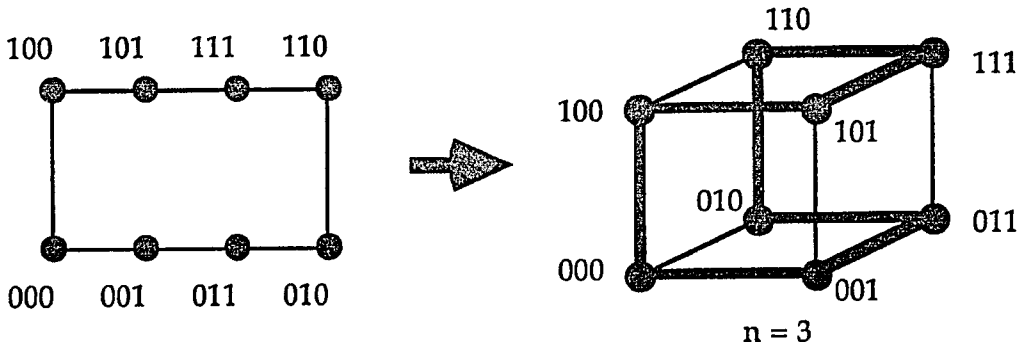


Figure 2.1. Embedding of a one-dimensional grid.

2.3.1 Efficient Homomorphic Embeddings

Efficient homomorphic embeddings exist for both perfect shuffles and toroidal and linear grids of any dimension as long as the number of processors along any dimension is a power of two [NCU85]. A one dimensional toroidal grid, shown in Fig. 2.1 (dark lines indicate active communications paths), is simply a ring. The predecessor and successor are given by the following:

$$\textit{predecessor:} \quad G(F(x) - 1)$$

$$\textit{successor:} \quad G(F(x) + 1)$$

As a further example, consider the embedding of the two dimensional toroidal grid shown in Fig. 2.2. All of the available communications paths are used in this case. The mapping of processor (x, y) in a two dimensional grid of 2^M processors in the x-direction and 2^N processor in the y-direction is given by:

$$\textit{processor id} = 2^M G(y) + G(x)$$

Further, if a processor id number is $k = 2^M y + x$ then its neighbors are:

$$2^M y + G(F(x) - 1)$$

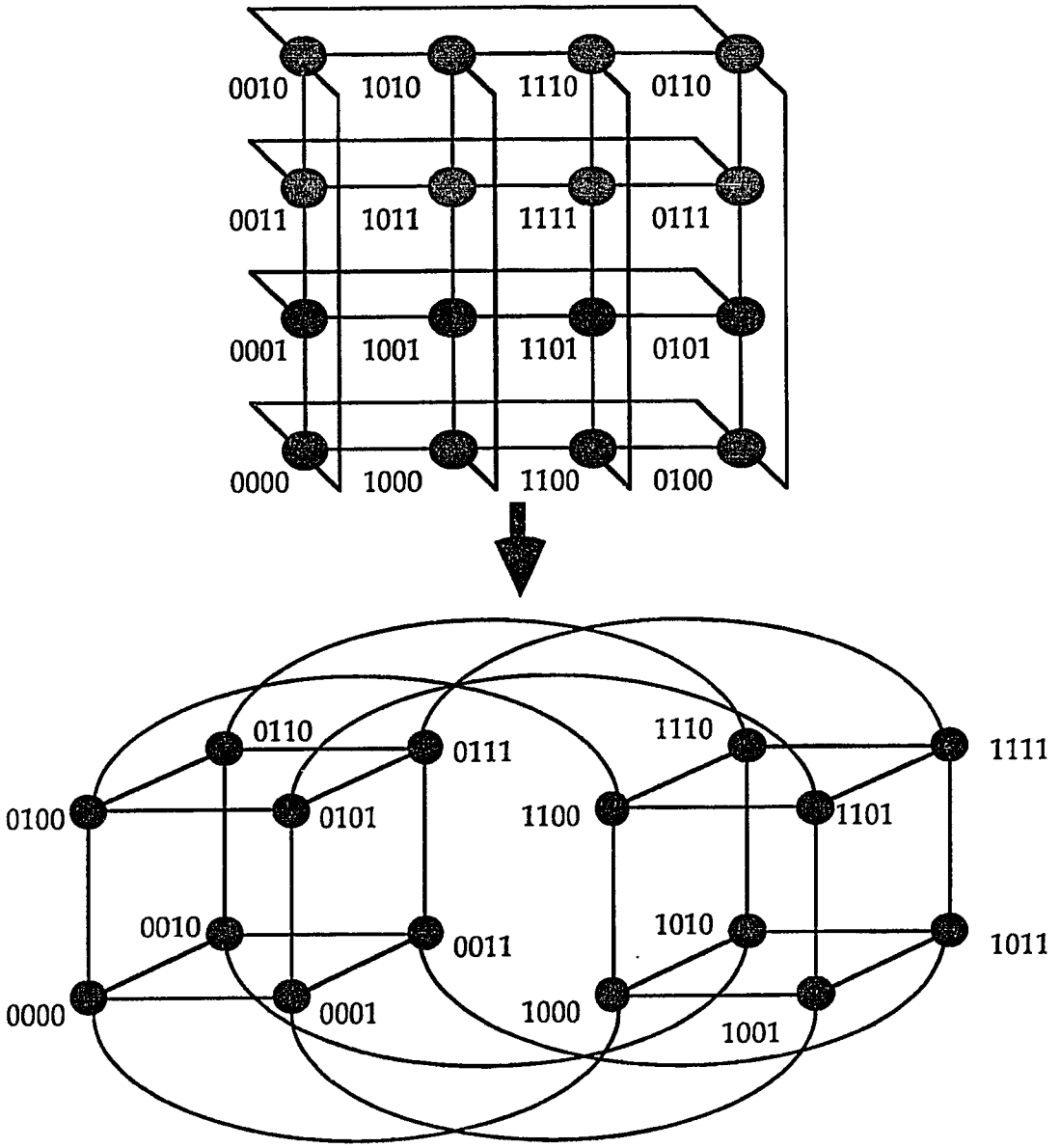


Figure 2.2. Embedding of a two-dimensional (4 × 4) grid.

$$2^M y + G(F(x) + 1)$$

$$2^M G(F(y) - 1) + x$$

$$2^M G(F(y) + 1) + x$$

This scheme can be generalized to the n-dimensional case by dividing the processor id into n fields, and computing the successor and predecessor along each of the n axes.

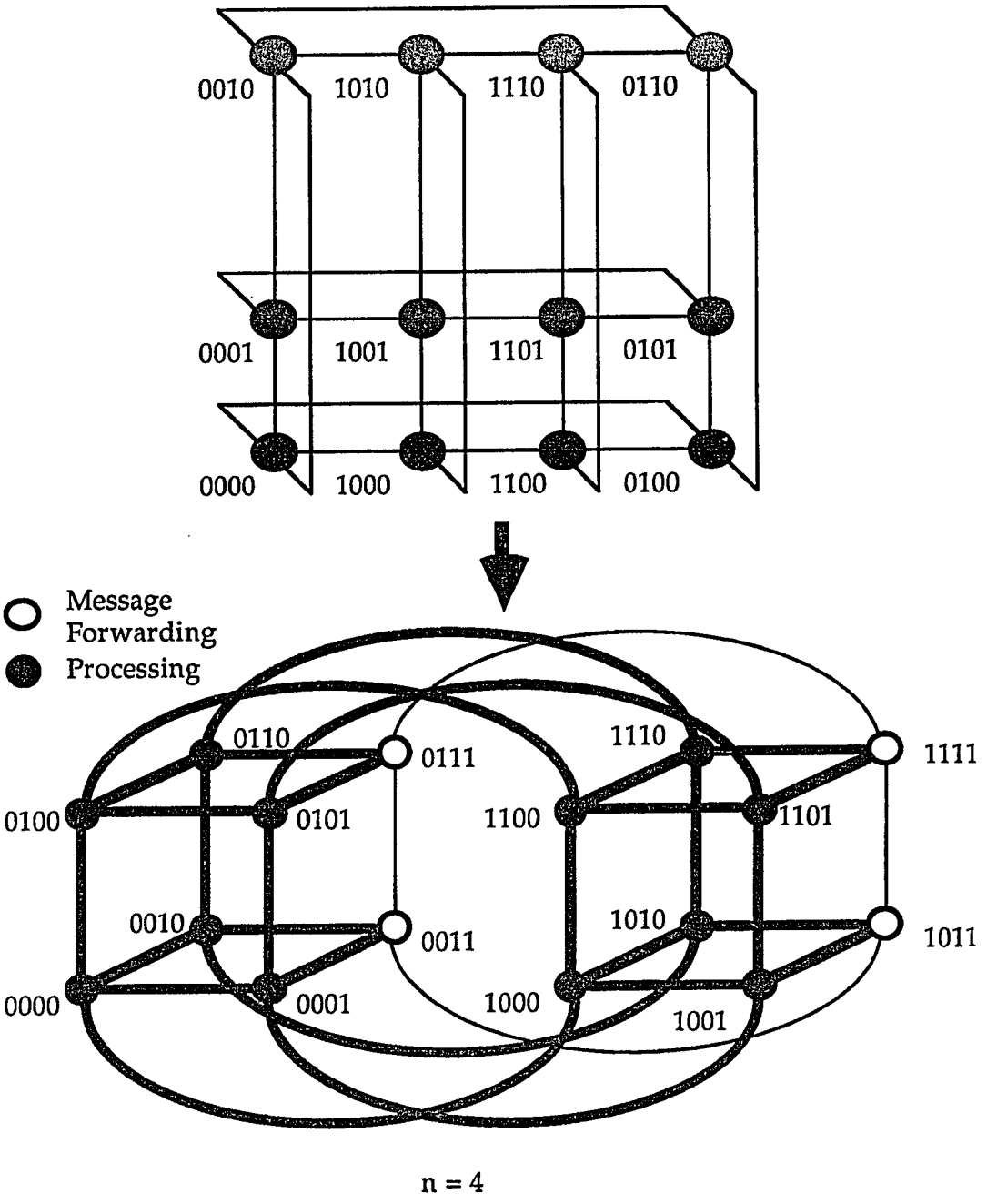


Figure 2.3. Embedding of an odd dimensioned (4 × 3) grid.

2.3.2 Inefficient Embeddings

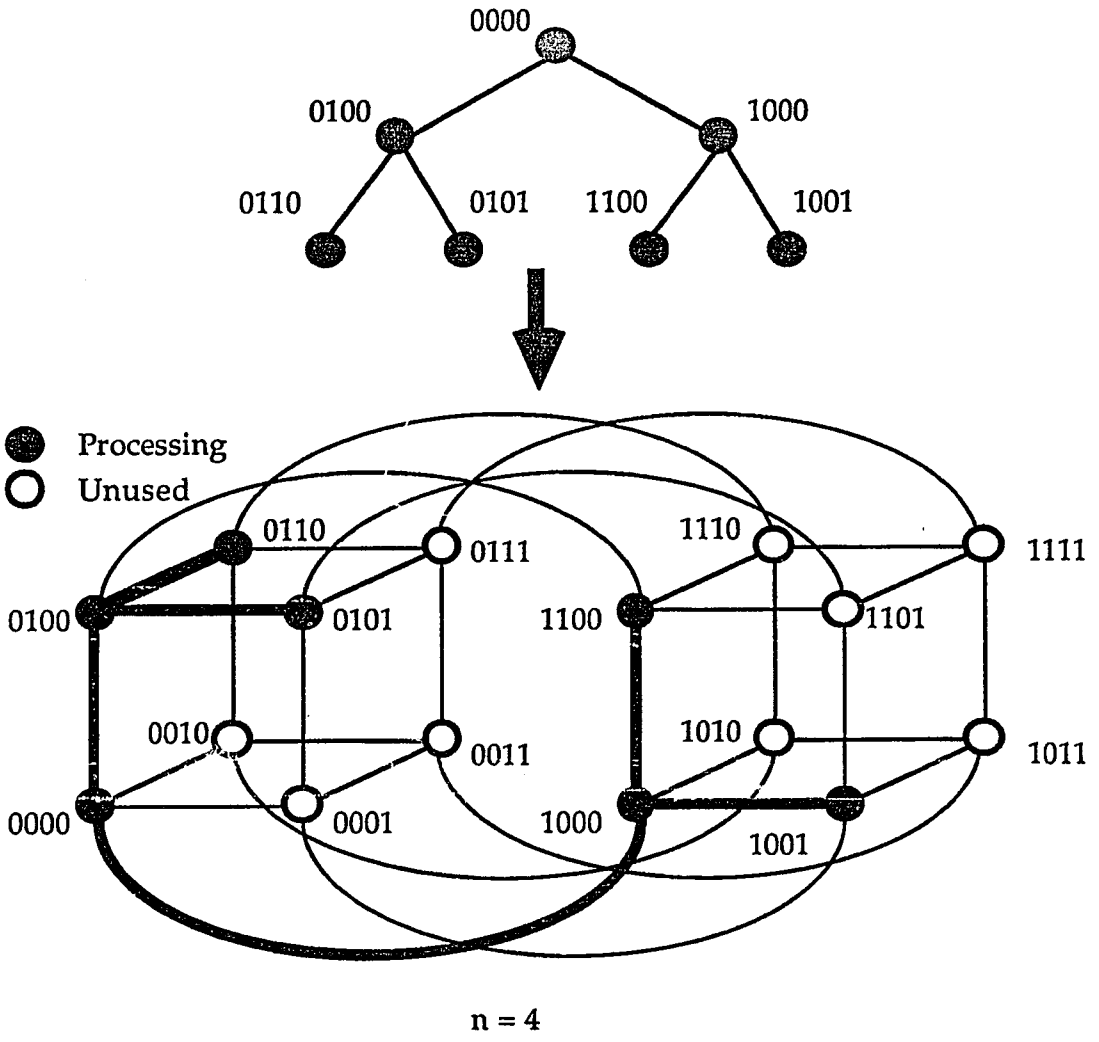


Figure 2.4. Homomorphic embedding of a two-level tree.

Grids that are not an exact power of two in each dimension will not homomorphically embed into a hypercube. This situation may be ameliorated, however, by adding processors to the dimensions that are not powers of two, see Fig. 2.3. These added processors serve only as connecting nodes to forward communications along the given dimension. They are labelled *message forwarding* in Fig. 2.3; again, the active communication paths are indicated by the darker lines. The inclusion of the connecting nodes will add to

the communications cost incurred by all messages that must pass through them. Such messages will be referred to as *pass through* messages.

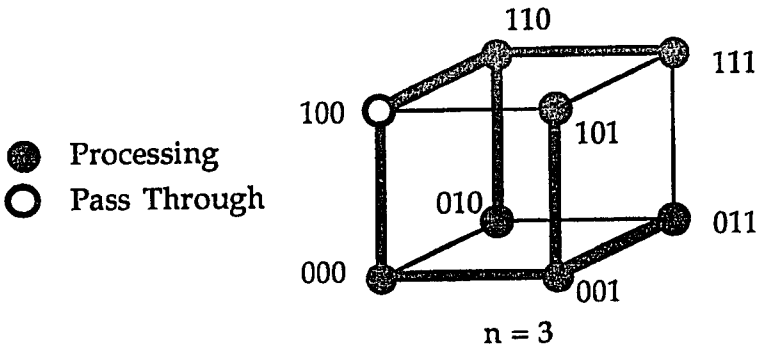


Figure 2.5. Efficient embedding of a two-level tree.

Binary trees may be embedded homomorphically, albeit inefficiently. The mapping of a two level binary tree rooted at node 0 is shown in Fig. 2.4. This tree could not be homomorphically embedded into a 3-cube because each of the level 1 nodes has two children and there are only three nodes that lie at a distance of two from the root. In general, an $(n+2)$-dimension cube is required to embed an n-level tree with unit dilation, even though only $2^{n+1} - 1$ processors are actively used. In other words, an expansion factor of slightly greater than two is required to embed a binary tree with a dilation of one. However, the spatial requirements drop to 2^{n+1} processors if communications between certain pairs of processors are allowed to pass through otherwise unused channels of other nodes in the tree. It has been shown in [BI85] that a complete binary tree of height n with $2^n - 1$ processors can be embedded within an n-dimensional cube with one edge of dilation two (that is, requiring one pass through node) and every other edge of dilation one, for every n. They further show that the passed through node is the unused one. Such an embedding is shown in Fig. 2.5. In this case node 4 (100) acts as the message forwarding node for nodes 0 and 6. However, many applications that employ tree communication structures do so with trees that grow and shrink dynamically [AM88,PLG87]. It is also frequently the case that only the leaf nodes that are active at any given point in time. Thus, the full usefulness of embedding trees will not be realized until they can be

Term Defined	Page
NN – Near Neighbor Communications	15
RA – Random Communications	15
NNBC – Near Neighbor Broadcast Communications	16
BC – (Random) Broadcast Communications	16
SCMD – Single Code Multiple Data Programs	18
MCMD – Multiple Code Multiple Data Programs	19
C-deterministic Communications	20
Homomorphic Mapping	21
Gray Code	22
Dilation	22
Expansion	22

Table 2.2. Index of Definitions.

embedded dynamically. The dynamic embedding of trees will become more feasible as multiprocessing support becomes more prevalent on hypercube array nodes.

The definitions introduced in this chapter and the page on which they appear are listed in Table 2.2.

CHAPTER 3

NEAR NEIGHBOR COMMUNICATIONS

In this chapter we concentrate on the issues related to providing support for near neighbor (NN) communications. Some of the issues we discuss will also be applicable to the more general case of random communications which is the topic of the next chapter. We begin by discussing the implementation of NN communications in existing systems. The inefficiencies are enumerated and alternatives are suggested. In particular, we describe our new communications module which leads to significant performance increases in C-deterministic phases of program execution. A model for the execution time of SCMD near neighbor algorithms is then developed. We present results from both measurements on actual programs and simulations of program execution. The experimental results are provided to both verify the model that we have developed for use in our simulations and to quantify the improvements that are realized by our new communications module. Finally, we use our simulation models to quantify the further improvements in efficiency that are possible with architectural changes.

3.1 Existing Implementations

The largest class of algorithms that have been implemented on large distributed memory machines employ NN, NNBC and BC communications. Efficient support for these types of communications is absent from commercial operating systems. The only available system that attempts to provide efficient support for NN, NNBC and BC com-

munications is the Crystalline operating system (CrOS) which was developed at the Caltech[FK86a].

Of the existing available operating systems NCUBE's Vertex and Caltech's CrOS exhibit the best performance. Vertex is faster than CrOS for all but the smallest of messages [MBA87]. These two operating systems provide communication primitives that are close to the opposite extremes of functionality. Vertex provides a general mechanism that will route arbitrary length messages from any source to any destination in the cube. Whereas CrOS will send only fixed length messages to immediate neighbors. These two operating systems are described in the following sections. The Vertex operating system is explained in considerable detail because it serves as the basis for the communications module that we have developed.

3.1.1 NCUBE – Vertex

Vertex is a run-time executive program that executes on the NCUBE array processors. The services provided by Vertex are program loading, low-level error handling, low-level debugger support, node identification and local time routines and communications support that includes message routing and buffer handling. Vertex is compact, requiring only about 6 K-bytes of memory for both code and data (excluding the communication buffers). Entry to Vertex from high level language libraries and user written assembly language routines is via an operating system trap call that saves the current program status word and program counter and branches into Vertex code via an interrupt jump table. A similar mechanism may also be invoked by the hardware in response to an execution exception, or an external interrupt request.

User programs may query Vertex to learn the logical (with respect to the currently allocated sub-cube) node address on which they are executing, the host interface processor address and the dimension of the currently allocated sub-cube. They may also request the time since node initialization, which is kept in multiples of 1024 clock cycles. These two call handlers, and those of the communication system that will be discussed later,

comprise the primary operating services. These services are invoked via software traps.

The low-level error handlers are invoked by hardware recognized program exceptions. They save processor state information and suspend execution of the user process. This allows the debugger to examine the state of the node as it was when the exception was detected.

Vertex communications are driven by two events: requests from the user program via operating system traps; and responses to communication channel interrupts. A brief description of the three calls that comprise the interface to the user program is given here, a more detailed discussion appears later. The node write call is named *nwrite*. It sends a message of a specified type to a designated destination (users may associate a type value, an integer between 0 and 32783, with any message). The *nread* call examines the received-but-unclaimed message queue for a message from the specified source of the desired type. When the desired message is found, it is returned to the caller. The *ntest* call performs a function similar to nread, except that it returns immediately after checking the existing messages and only reports on the success of the search — located messages are not returned.

A pool of communication buffers is maintained by Vertex. These buffers are used on both the sending and receiving nodes. They allow callers of nwrite to be released after a communication buffer is allocated and the user data is copied into it. Therefore, callers may proceed while the system is waiting for access to a communication channel. Furthermore, data to be sent that is allocated from dynamic storage may be released any time after returning from the nwrite call. Callers of nread are not required to make their calls before messages arrive since all arriving messages are first stored in communication buffers.

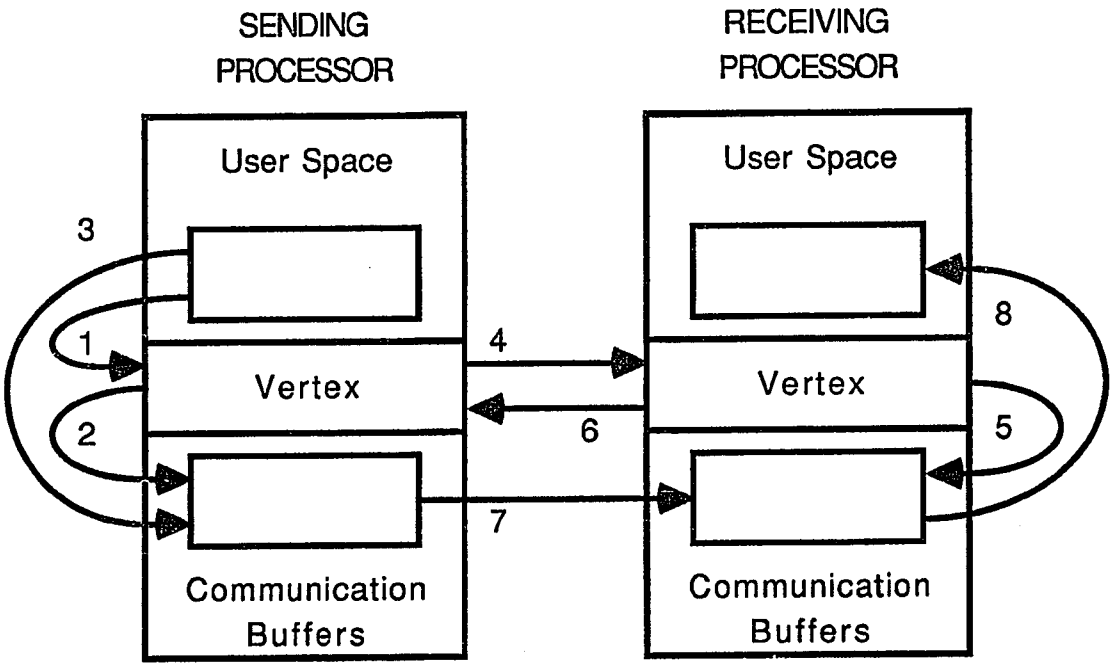
The communication buffer pool begins as a single free buffer. Requests for buffers are satisfied by searching linearly through the doubly linked buffer pool for free buffers of a sufficiently large size. When one is located it is further checked to see if it exceeds the requested size by 32 or more bytes. If so, the free buffer is split so that a buffer

of the requested size can be allocated from the end furthest from the front of the list. This policy tends to concentrate free buffers near the beginning of the list. If no buffers of sufficient size are located the request may be queued to search again when another buffer is returned. Adjacent free buffers may be collapsed into a single buffer by the buffer deallocation routine whenever a buffer is returned. Buffers that are in use may be queued on other system queues via a second pair of link fields. Thus, Vertex provides buffer allocate, deallocate, enqueue, and dequeue procedures for its own internal use.

The specific action of nwrite is to check for valid message length and destination, allocate a communication buffer and copy the user data to it, calculate the outbound channel to use, and, if the channel is inactive, initiate the message transfer protocol. In the case that the channel is busy, the message is queued to be sent as soon as the channel becomes available. In either case, the caller is released to continue execution. An interrupt is signaled when the final step of the DMA controlled message transfer has been completed. The service of this interrupt includes releasing the communication buffer. Incoming messages are placed in communication buffers and then queued in an unclaimed message list. Nread checks this queue for a message with a specific source and type, if it is not found it will wait until an appropriate message arrives. When the requested message is located, it will be copied to the user data space, dequeued from the unclaimed message list and the communication buffer will be released. Callers to nread and ntest may specify that messages of any incoming source or type are desired.

Message transmission is controlled via a three way handshake protocol. Simply stated, the sender sends a two byte message to the receiver indicating the number of bytes it wishes to transmit (all inactive channels are set to receive two byte transmissions). The receiver allocates a communication buffer, sets the appropriate DMA channel to receive the indicated number of bytes and then transmits a two byte acknowledgement back to the sender. Upon receiving this acknowledgement, the sender transmits the entire message. This protocol is initiated by a call to nwrite. However, the remaining steps are handled by a sequence of interrupt events. Specifically, the reception of the two byte length message

by the receiver and the reception of the two byte acknowledgement by the sender both signal interrupts. An interrupt is also generated on both the sending and receiving nodes when the DMA transfer of the message is completed.



Actions:

1. Sender: nwrite call
2. Sender: allocate communication buffer
3. Sender: copy buffer, release nwrite caller
4. Sender: transmission request
5. Receiver: allocate communication buffer
6. Receiver: transmission request accepted
7. Sender: transmit data
8. Receiver: if nread pending, copy buffer, release nread caller

Interrupt:
after 4 on receiver
after 6 on sender
after 7 on sender and receiver

Potential Waits:
at 2 for local buffer
at 4 for use of channel
at 5 for remote buffer
at 6 for use of channel

Figure 3.1. Node-to-Node Communication (Vertex).

A more detailed description of a message send/receive transaction between two adjacent nodes is given below. Figure 3.1 expresses this transaction pictorially. In step 1, the sender issues an nwrite call which generates an operating system trap event. In step 2, the Vertex nwrite trap handler allocates a communication buffer, waiting in a buffer request queue if none are presently available. Once allocated (step 3), the message data is copied from the user process to the communication buffer and the buffer is queued in the send list. The nwrite caller is released at this point. In step 4, the two byte transmission request that indicates the message length is sent. If the channel to the requested destination is not busy this action occurs as the final action of step 3. Otherwise, this transmission request is queued awaiting the availability of the requested channel. Reception of the two byte transmission request generates an interrupt on the receiving node. The handler for this interrupt (step 5) attempts to allocate a communication buffer of the requested length. If it is successful, an acknowledgement message is sent back to the sender (step 6). If a buffer is unavailable, a buffer request is queued and the acknowledgement message is postponed until after this request is satisfied. It may also be necessary to wait for the channel from the receiver to the sender to become free at step 6 (if a previous message send is still in progress). The reception of the two byte acknowledgement generates an interrupt for the sender. In this interrupt handler (step 7), the DMA transfer of the message is begun. Upon completion of this DMA transfer, an interrupt is generated for both the sender and the receiver. On the sender, the communication buffer is dequeued from the send queue and released. If another message is ready to be sent on the same channel, this procedure is repeated from step 4, otherwise the channel is reset to the inactive state. On the receiver, the message is checked to see if this node is the final destination and if this message is a user message. If so, the communication buffer is queued on the incoming unclaimed message list and the channel is reset to its inactive state. If an nread call has been issued for the incoming message, it is dequeued from the incoming unclaimed message list, copied to the user data space, and the communication buffer is released. This corresponds to step 8.

Multi-hop messages are handled in a manner very similar to the above. The following actions, beginning with step 8, occur on each of the intermediate nodes. The receiver interrupt handler inspects the destination field of the message (this field is stored in a header that is prepended to the message data); calculates the next channel to send it out on in order to get it another step closer to its final destination; queues the communication buffer on the send list and, if the channel is not busy, sends the two byte transmission request message thus assuming the role of the sender at step 4. This procedure is repeated on each of the intermediate nodes.

System messages (primarily for interactions with the low-level debugger) are also handled in a very similar manner. On the destination receiving node at step 8 the message is checked to see if it is a system message. If so, the appropriate system message handler is invoked.

3.1.2 Caltech – CrOS

CrOS provides a very simple interface to the communications architecture [FK86a]. It has been implemented on variety of systems: the Intel iPSC/1, the JPL Mark II and, recently, the JPL Mark III [FJL*88] and the NCUBE [BF87a]. Though many versions of CrOS exist, they all share two basic routines to read and write the physical communication channels. The message data is sent as a sequence of fixed size packets. The size of these packets is dictated by either the hardware (16 bytes on the iPSC/1) or the size of the operating system communications buffer that is statically assigned to each communication channel. The CrOS protocol is synchronous. A sending node will not return from a call to the write operation until receiving a signal confirming that the receiving node has read the packet from its communication buffer. Before a node writes to a channel, it must first ensure that there is not an unread packet in the communication buffer of the corresponding input channel. Failure to read a packet that is present in the corresponding input buffer before writing to the output channel will result in deadlock. The arrival of messages does not cause an interrupt. CrOS will repeatedly poll the input channel

```
if (not root)
{
    from_node = father(root_node, current_node)
    receive message (from: from_node)
}
son_nodes = sons(root_node, current_node)
for (each son in son_nodes)  send message (to: son)
```

Figure 3.2. Global Send Routine.

until the incoming message is received. The limitations of the CrOS communications paradigm are described in Sec. 3.2

3.1.3 Broadcast Algorithms

Broadcast messages are not directly supported by either Vertex or the early versions of CrOS. A recent version of CrOS that has been designed for the NCUBE includes a broadcast capability that is similar to the one that we use in our communications module which is described in Sec. 3.3. This capability was first described in [MBA87]. Several algorithms for implementing efficient broadcasts have been proposed [SW87,HJ86]. A library of broadcast based operations is available from Intel for the iPSC/1 and iPSC/2 [Cub]. This set of operations has gained wide use due to its distribution through the Intel iPSC users group. These routines have been ported to many other hypercube machines, including the NCUBE. The basic broadcast routine is described below. A pseudo-C code listing is given in Fig. 3.2. The subroutine *father* returns the node identifier (possibly null) for the node that lies one hop closer to the root than the current node. The subroutine *sons* returns the list (possibly null) of nodes that are one hop further from the root than the current node in the spanning tree.

This routine implements a spanning tree rooted at the node originating the broadcast [BS86]. A parent child ordering of the spanning tree is used by this algorithm as shown in Fig. 3.3. The diagram labelled *local send loop* in Fig. 3.3 shows the progress

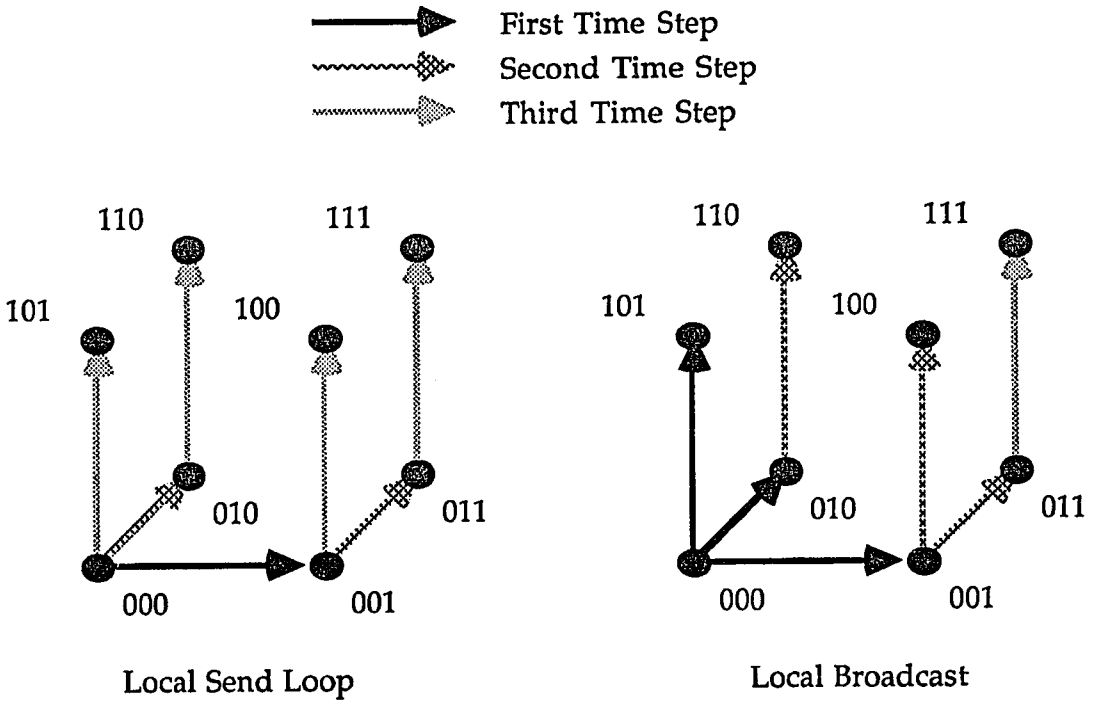


Figure 3.3. Broadcast Spanning Tree.

of broadcast messages in a typical implementation. This corresponds directly to the code given in Fig. 3.2. If the communication architecture provides a broadcast instruction, an algorithm that progresses as depicted by the diagram labelled *local broadcast* may be used. In this case, the last line of code in Fig. 3.2 is replaced by a single call that simultaneously broadcasts the message to all sons.

3.2 Points of Inefficiency

Two major points of inefficiency can be identified within the communication scheme implemented in Vertex with respect to NN communications. One is the use of buffer copies to move messages between the system communication buffers and user space. The second is the overhead incurred between communicating nodes by the three-way handshake protocol. In many cases, algorithms that fall into the NN classification are C-deterministic. Presently, the knowledge that is inherent in the C-determinism is unused.

With simple extensions to Vertex, the information about impending communications can be exploited to minimize both of the above inefficiencies.

The costs associated with the buffer copies are easy to quantify. Both the best case time to dynamically allocate a buffer and the amount of CPU time consumed by the copying of data from one buffer to another can be directly measured. The amount of memory bandwidth utilized in a copy operation can also be calculated. The cost of using a communication protocol that is more complex than necessary is more difficult to quantify. These costs can be traced to two major sources. One is the overhead incurred by servicing interrupts generated by unnecessary protocol messages. This cost can be directly measured. The second cost results from the loss of the use of the channel from the receiving node to the sending node for the duration of time between the message length transmission arriving at the receiver (step 4 in Fig. 3.1) and the acknowledgement arriving back at the sender (step 6 in Fig. 3.1). This duration may be arbitrarily long because the receiving node may not have sufficient communication buffer space immediately available. These effects are reflected in the experimental results that are presented in Sec. 3.6.1.

The communications paradigm employed by CrOS has four limitations. First, the communications between adjacent nodes can occur in only one direction at a time. For example, during the time that a message is traveling from node 0 to node 1 the protocol precludes a message from node 1 to travel to node 0. Secondly, it is not possible for a node to proceed with normal program execution while it is either sending or receiving a message. The suspension of normal program execution on the sending node will last from the time that the message send is initiated until the message has been completely received on the receiving node. On the receiving node the suspension of normal programs execution will last from the time that the call indicating the intention to receive the message is made until the message has completely arrived. Send and receive activities on different communication channels are also precluded from occurring simultaneously. The third limitation is that the order of all message sends and receives must be known

on each node. This requirement is more restrictive than C-determinism. Finally, the cost of supporting message packetization in CrOS leads to lower overall throughput when moving large amounts of data.

Broadcast communications are an additional source of inefficiency. They are not directly supported in any of the widely used systems. Broadcasts are frequently implemented by a sequence of NN communications arranged in a spanning tree order as shown in the local send diagram of 3.1.3. Each node (except the root) receives a message from its parent node, then serially relays the message to each of its child nodes. The root node begins the operation by serially sending the message to each of its child nodes. Thus, in addition to the inefficiencies already noted for near neighbor communications, broadcasts also incur a store and forward overhead (with two buffer copies), a serialization cost associated with sending to only one spanning tree child node at a time, and the overhead of making repeated calls to the operating system. These costs are accrued at each non-terminal node in the spanning tree. On the NCUBE the costs associated with serialization and repeated operating system calls are unnecessary because a broadcast instruction is implemented in the instruction set of the node processors; however, NCUBE does not take advantage of this instruction in their implementation of Vertex. This broadcast instruction allows a message to be sent on any subset of the output channels of a node in a single DMA action.

3.3 Alternatives within the Existing Architecture

We have developed a new set of operating systems communication primitives to address the major inefficiencies described in the preceding section. These operating system routines and interrupt handlers may be installed on a channel-by-channel basis. This allows Vertex to be used for communications along certain dimensions while simultaneously employing the new system for communications along other dimensions. The choice of communication system may be changed during program execution as long as care is taken to ensure that no communications are pending or in progress during the change.

This new system will be henceforth be referred to as *Extended Vertex* to distinguish it from the *standard Vertex* system. The user interface to this system is described in the following paragraphs.

The new send call, named *send*, and new receive call, named *rcvreq*, do not use communication buffers allocated from system space. This eliminates the need for buffer copying. These calls are also asynchronous. Consequently, the return from a send or rcvreq call does not indicate completion of the communication activity. Instead, the completion of communication activities is indicated in a flag variable that is specified by the caller of the communication routine. The setting of the completion flag for both the sender and receiver is triggered by the end-of-DMA interrupt. The caller of send is required to ensure that the message data is not corrupted before the completion flag is set. The receiving process is required to make a call to rcvreq to allocate a buffer before the anticipated message begins to arrive. The primary benefit of bufferless communication is that the incremental (i.e., per byte) cost is substantially reduced.

As a slight digression, we note that the requirement of having to call rcvreq before the anticipated message begins to arrive could be relaxed if minor modifications were made to the existing NCUBE architecture. The primary change would be to accommodate the reception of messages whose length is embedded into the first few bytes of the message. This change would allow messages that arrive before they are expected to be stored in a predetermined system buffer for later retrieval by the receiving program. Such a scheme would require protection against memory buffer overruns. This protection could be provided by accepting only those incoming messages with lengths that are less than the lengths of the available system buffers.

The send and rcvreq calls further exploit the *a priori* knowledge that is inherent in C-deterministic programs by eliminating the three-way handshake protocol. The send call makes the assumption that the receiver is ready, thus no transmission request message is used. In addition, the caller of rcvreq is required to specify the exact number of bytes to be received and, finally, no acknowledgement of message receipt is offered. The chief benefit

of eliminating the handshake protocol is a substantial reduction in the fixed overhead of NN communications. A variation of the rcvreq call is also available. It allows a user specified routine to be executed on the end-of-DMA interrupt. This variation provides a great deal of flexibility for allowing further asynchrony of communication operations. An example of this that leads to significant performance increases appears in the Linpack part of Sec. 3.5.2.

There are three other calls that are available in Extended Vertex. Two routines switch the currently active communication scheme. The final extension is a broadcast call that allows a node to send the same message to any set of nearest neighbors simultaneously. The broadcast routine is analogous to the send routine in all other aspects of its operation.

3.4 Near Neighbor Execution Time Model

We develop an execution time model for programs with NN communications in this section. The chief attribute of this model is that it accounts for the overlap of communications with calculations. Its is verified in Sec. 3.5.2. The model is then used to predict the effects of various architectural changes on program execution times in Sec. 3.6.1.

Interacting parallel SCMD programs can be described from the viewpoint of the executing process as consisting of a sequence of program steps whose execution time is given by: t_{xi}, where i is a counter that is incremented for each successive step, and where x is any of c (calculations), s (setup for communications), or w (waiting on communications). During any step i, the program time is accounted for by exactly one of t_{ci}, t_{si}, or t_{wi} with the remaining $t_{xi} = 0$. This notation must be extended to accommodate asynchronous events such as the interrupt service routines that are part of the communication protocol. These asynchronous events occur only as part of the time that is classified as communication setup time (t_{si}). Thus, for a communication instruction that starts at time step i and whose (interrupt driven) communication protocol must complete before step $t_{x(j+1)}$ starts, the notation is extended to: $t_{si,j:k,l}$. The subscripts

k and l are used to handle the possibility of multiple independent asynchronous events (i.e., multiple simultaneous communications) occurring in the interval i, j. The k denotes independent sequences of events (i.e., independent communications) and l indicates the order of occurrence of the event within sequence k. This allows our notation to specify that receive interrupts not occur before their corresponding send interrupts. No ordering is implied for events in different sequences (k). As an example, consider the sequence of events described by:

$$t_{s5};\ t_{s5,7:1,1};\ t_{c6};\ t_{s5,7:1,2};\ t^{5}_{w7};\ t_{c8};\ \cdots \tag{3.1}$$

The major events in this sequence are:

time 5: send operation is started

time 6: calculations (that do not change send data) are performed

time 7: wait to receive data that was sent at time 5

time 8: calculations on the received message data are performed.

Between time steps 5 and 7, two sets of asynchronous interrupts occur. One set is for the send operation, the other is for the receive operation. The times for these events are accounted for, respectively, by $t_{s5,7:1,1}$ and $t_{s5,7:1,2}$. Since the send and receive are part of the same communication operation, we assign them the same k-value (k = 1) (in reality, the related send and receive operations occur on different nodes, but since we are modelling SCMD programs we can collapse the timing information into a single node model). Since receive interrupts will occur no earlier than their corresponding send interrupts, their l-value is higher (l = 2) than the l-value for the send interrupt operations (l = 1). Total program execution time is given by:

$$T = \sum_i t_{ci} + \sum_i t_{si} + \sum_i t_{wi} \tag{3.2}$$

Where the t_{si} include both synchronous and asynchronous execution steps.

The values for t_{ci}, t_{si}, and t_{wi} can be measured in existing programs. For hypothetical programs the designer should be able to calculate these values, either precisely or with knowledgeable estimates. The value $T_s = \sum_i t_{si}$ reflects the total time spent setting up communication operations. This includes the copying of non-contiguous message data to contiguous memory locations, the time spent in the operating system to initiate sends or receives, single polling operations for expected data, and the time spent handling communication protocols including that of the interrupt service routines. Repeated polling operations constitute time spent waiting and, as such, are counted as part of the t_{wi}.

The value $T_c = \sum_i t_{ci}$ accounts for the time spent calculating the desired results of the program. If we had "idealized" communication operations (that is, the sums of t_{si} and t_{wi} both equal zero) T_c would be the total program execution time. The individual t_{ci} may be data dependent. If this is the case, either an average value can be used, or the analysis can be repeated for the anticipated best and worst cases.

Finally, the value $T_w = \sum_i t_{wi}$ reflects the total time spent in a blocked state waiting on communications. In general, t_{wi} is a function of message size (m), message transmission rate (r), communication resource (i.e., link or buffer) contention, and the time elapsed since the awaited communication was initiated, see (3.3). To account for this elapsed time, the parameter j is added to t^j_{wi} to indicate the step when the communication that we are awaiting was initiated (see 3.1). The time elapsed since communication initiation can be determined given knowledge of the t_{ci} and t_{si}. The effects of resource contention are potentially much more difficult to model. However, for SCMD programs that are NN or BC, such conflicts are usually avoidable by careful ordering of communication operations. Those times when such conflicts are not avoidable can be indicated by a non-zero value for t_{di} in (3.3). The time at step i that is spent blocked waiting on a message that was initiated at step j is thus given by:

$$t^j_{wi} = \max(0, t_{di} + mr - \sum_{k=j}^{i} t_{xk}) \tag{3.3}$$

where it is understood that $\sum_{k=j}^{i} t_{xk}$ does not include the time spent on any protocol interrupt overhead for the communication on which we are waiting. Thus, for (3.1) $t_{w7}^{5} = mr - t_{c6}$, because t_{c6} represents the only activity between steps 5 and 7 that could be overlapped with the message transmission (mr).

3.5 Experimental Results

The experimental results reported here, unless otherwise noted, have been measured on an NCUBE/ten system with a 7 MHz clock. Basic communication performance parameters are presented first. They are followed by performance measures extracted from actual programs. The execution time model is verified and then used, along with further program measurements, to demonstrate the program execution time improvements that may be realized by incorporating our new operating system calls into programs executing on the existing NCUBE architecture.

3.5.1 Basic Parameters

The sustained bandwidth and latency are measured by a program that configures 64 node processors into a ring and sends messages of varying lengths around the ring 1000 times, one message at a time. The total time is recorded for each message. A least squares method is then used to determine sustained bandwidth. The latency is the time required for a zero length message to traverse one hop. The latency and inverse sustained bandwidth (the inverse is given so that the measurement units are similar to those for latency) values for both standard Vertex and our extensions are presented in Table 3.1. Total message times for messages of length x can be expressed from the values in Table 3.1 as $t = ax + b$, where a is the inverse bandwidth and b is the latency. This table also lists the values reported in [RG87] for the Intel iPSC/1 (8 MHz) and Ametek System 14/n (8 MHz) systems with their standard operating systems, and values extrapolated from[FK86a] for the CrOS operating system running on an Intel iPSC/1 (8 MHz) and Caltech Mark II (8 MHz). Due to the packetizing nature of CrOS, small

message times cannot be accurately approximated with a single linear equation. Instead, the curves are piece-wise linear with discrete discontinuities at multiples of the packet size. Due to this piece-wise linearity of the message length/time curve for small messages, the linear approximation for large messages (i.e., greater than 32 bytes) under CrOS is given more accurately by: $t = 19.7x + 12$ for the iPSC/1 and $t = 10.15x + 65$ for the Mark II. The results for all processor and operating system combinations are shown pictorially in Figs. 3.4 through 3.6. As can be seen from these figures our extended version of Vertex offers the best basic communication performance. Furthermore, these basic performance measurements do not reflect the full potential for increased performance that is available in extended Vertex. In particular, the benefits of simultaneous bidirectional communications between adjacent processors and the possibility of greater overlap of communications and normal program execution are not revealed. Throughout the remainder of this section we will examine the performance of the Extended Vertex communication system with respect to the performance of the standard Vertex system.

	Latency	Inverse Bandwidth
NCUBE Ext. Vtx.	83 (μsec)	1.29 (μsec/byte)
NCUBE Std. Vtx.	466	3.14
Ametek 14/n	550	9.53
iPSC/1 IHOS	1700	2.83
iPSC/1 CrOS	304	19.70 (for large msg)
Mark II CrOS	90	10.15

Table 3.1. Basic Communication Performance.

The elimination of buffer copying has a large effect on the bandwidth of NN communications. Another way to effectively increase communication bandwidth via software methods is to unburden the reverse channel from carrying protocol messages. This effect is not seen in these simple measures because they only communicate in one direction. However, simultaneous bidirectional communications occur often in actual program runs. Even without the effect of the extra channel capacity that results from unburdening the reverse channel, bandwidth is still increased by 2.43 over the standard Vertex imple-

mentation. The elimination of buffer copies also reduces latency, as buffer allocation is no longer necessary. Buffer allocation takes about 40 μs when buffers are immediately available at the head of the queue. A much larger portion of the improvement in latency times, however, is due to the elimination of the handshake protocol. The overall effect is a speedup of 5.60 over the latency encountered in standard Vertex. The implications that these changes hold for algorithm design will be explored in the following sections.

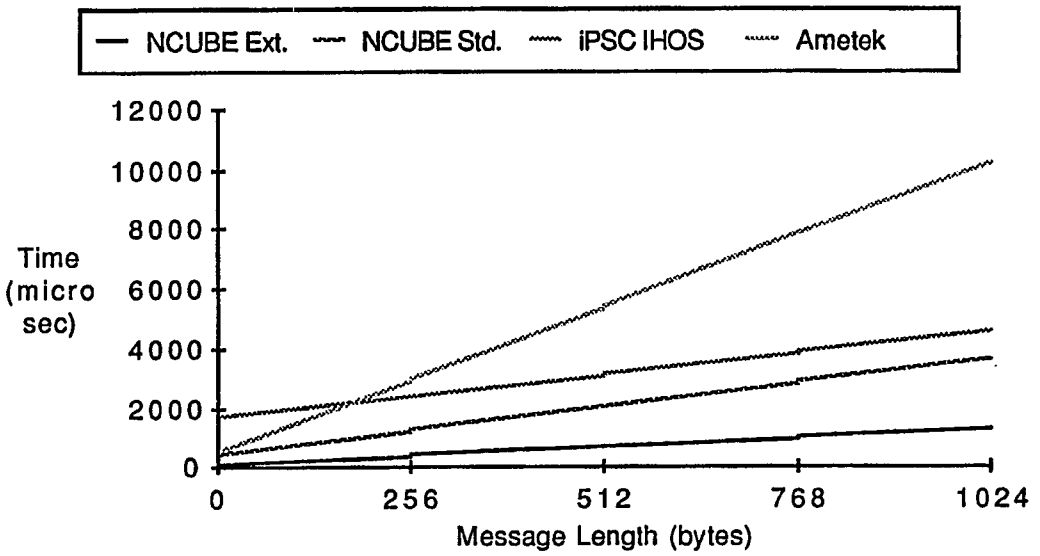


Figure 3.4. Basic Ring Message Times.

The send and receive call overheads have also been measured. All reported times include the call overhead as experienced from a C program using the standard NCUBE compiler calling conventions. Also, the execution path taken through the calls is the one in which no resource contention is encountered. These values, along with the cost of polling for message reception, are given in Table 3.2 for both standard and Extended Vertex. The send and receive call overheads are also shown pictorially in Fig. 3.7. Recall that the Vertex extensions do not require buffer copies to the communication buffer memory area, thus they incur no incremental overhead. These effects all contribute to the ability of Extended Vertex to efficiently support algorithms with finer granularity.

In addition to the cost incurred by initiating a send or receive operation via a library

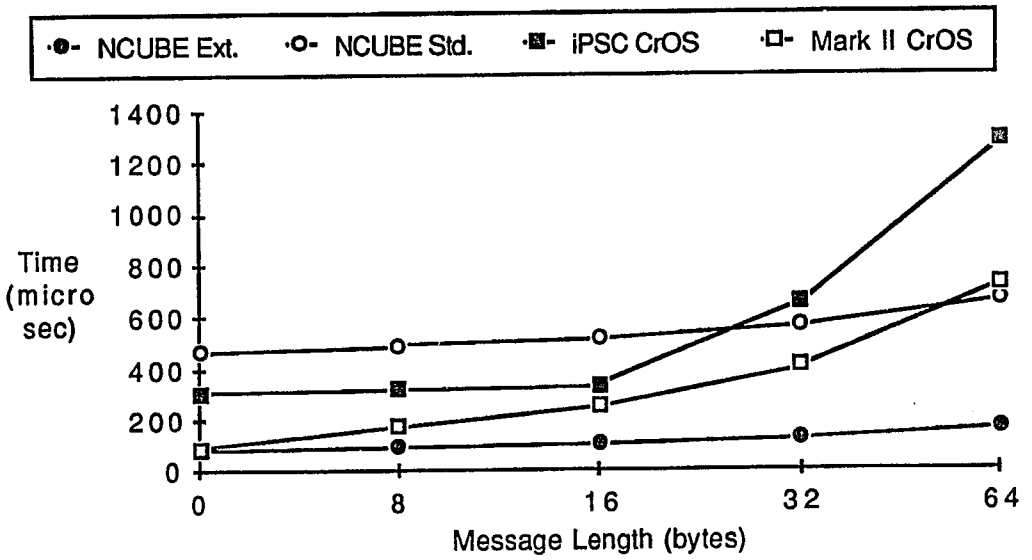


Figure 3.5. Basic Ring Message Times.

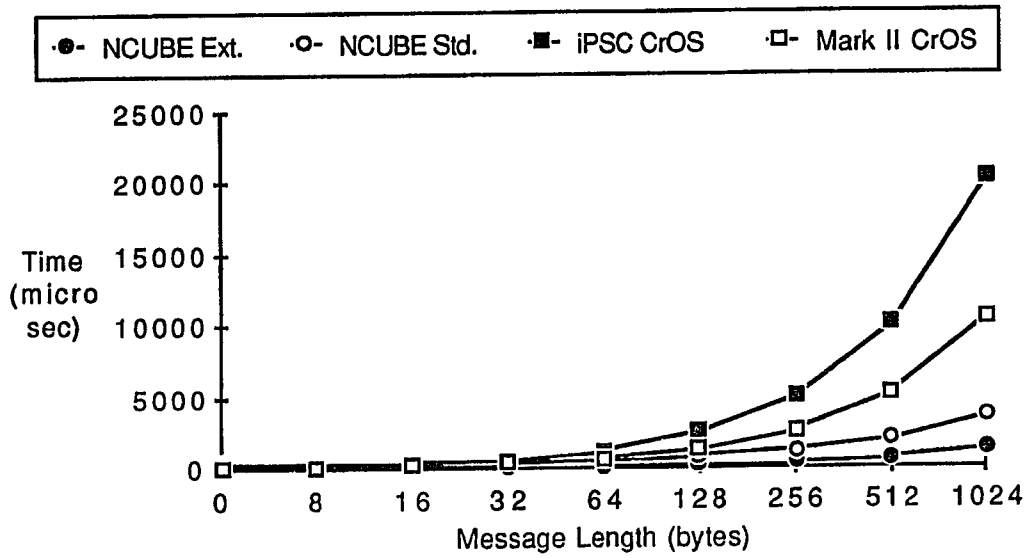


Figure 3.6. Basic Ring Message Times.

call from a C or Fortran program, there is the interrupt service cost associated with the parts of the communication protocol that are interrupt driven. These values are reported in Table 3.3 for both standard and Extended Vertex.

One of the major differences noted when programming with the Vertex extensions is that algorithms may be structured to flow more closely with the availability of data because of the substantially reduced message reception polling times. When polling for

	Fixed Overhead	Incremental Overhead
Ext. Vertex Recvreq	64 (μsec)	
Ext. Vertex Send	64	
Ext. Vertex Poll	5 (all cases)	
Std. Vertex Nread	162	1.00 (μsec/byte)
Std. Vertex Nwrite	172	1.00
Std. Vertex Ntest (poll)	61 (best case)	

Table 3.2. NCUBE Send/Receive Call and Polling Overhead.

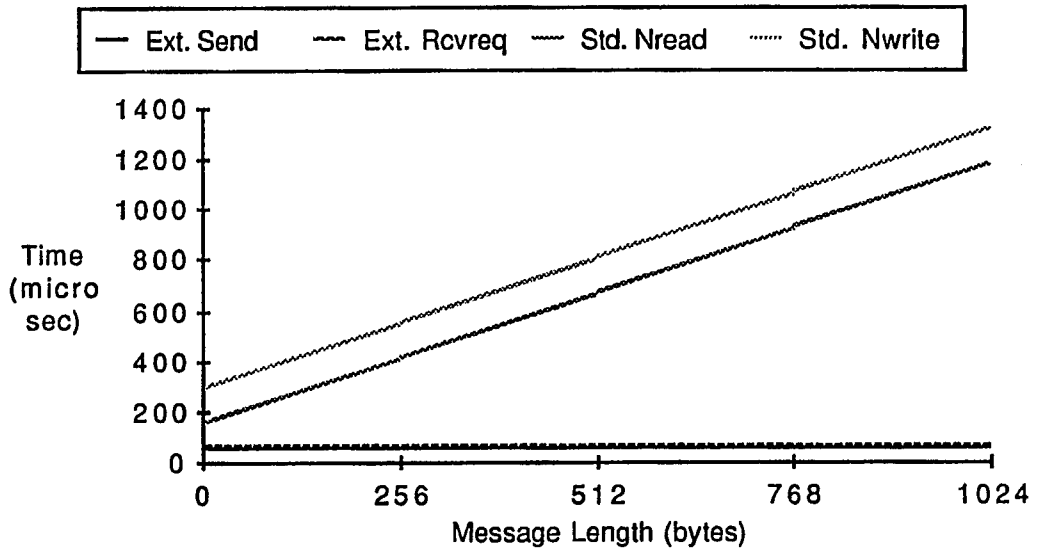


Figure 3.7. Send/Receive Overhead Times.

messages with standard Vertex, the system searches a central list of received but not yet claimed messages for all channels. Thus, the polling time is a function of the length of the unclaimed message list unless the search is ended early with a success. The incremental time increase experienced in standard Vertex is approximately equal to that of the entire search time in extended Vertex. Polling for messages in standard Vertex requires about 30 machine instructions when the message is found immediately at the front of the queue. Roughly 10–20 simple high level language (HLL) statements could be executed in comparable time. For each additional message that is checked while looking for the requested message, about $2\frac{1}{2}$ more machine instructions (or another HLL statement) may be executed. Further measurements that we have made indicate that the execution of

	Overhead (μs)
Standard Vertex Send	117
Standard Vertex Receive	186
Extended Vertex Send	25
Extended Vertex Receive	25

Table 3.3. NCUBE Interrupt Driven Protocol Overhead.

assembly coded BLAs (basic linear algebra operations) is about 2.4 times faster than the equivalent Fortran77 code. Thus, under optimized conditions the equivalent of at least 20–40 simple HLL statements could be executed in each standard Vertex polling time. With Extended Vertex, the cost of polling drops to the equivalent of about $2\frac{1}{2}$ simple HLL statements. The large reduction in the polling cost for Extended Vertex will often affect program design decisions. These design tradeoffs will be addressed in subsequent sections.

Another difference between the standard and extended version of Vertex is that with Extended Vertex the communication operations may be asynchronously decoupled from the calculation sections of the program even further. For example, in some algorithms message reception is polled for at several locations— not because the message is needed at these locations (if it were, a blocking receive would have been used), but rather, because the message needs to be modified and forwarded to another node as expeditiously as possible. An alternative to periodic polling is provided by Extended Vertex. This alternative allows user programs to specify a routine to be executed immediately upon message reception. In this manner, incoming data may be immediately processed and forwarded to other nodes. The user supplied data forwarding routine may also set the message-reception-completed flag.

3.5.2 Instrumented Programs

Sobel Edge Detector. We have coded and executed several versions of the Sobel edge detection algorithm. The Sobel edge detection algorithm is a simple low-level vision

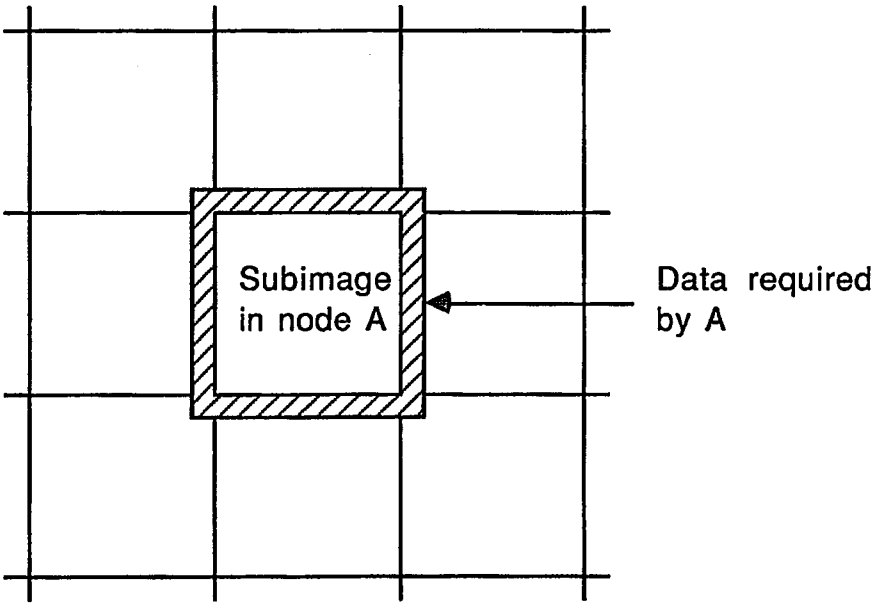


Figure 3.8. Remote Data Requirements for Node A.

algorithm that attempts to extract edge information in a grey-scale image [MA87]. It extracts this information by convolving a mask function (the Sobel operator) with a 3×3 pixel section (or kernel) of the image. The resultant value is used to replace the pixel value at the center of the kernel. This convolution operation is repeated for each pixel in the image. To execute this algorithm on a parallel processor, the image is divided into a two dimensional grid and a subimage is given to each node. The nodes then exchange edge and corner pixel values with the nodes that contain neighboring subimages, see Fig. 3.8. The Sobel algorithm makes a useful example because an algorithm with the same structure can be used for any problem requiring a convolution with a small kernel (for convolutions with large kernels, fast Fourier transform techniques become more efficient). The communication structure of the Sobel algorithm is also representative of many other algorithms [OM87,KJ87].

We have executed the Sobel algorithm with a variety of communication types and data set sizes. We have also varied the order of the communications operations. Execution times of these various versions were extensively profiled. For the standard Vertex cases the algorithm consists of the following steps:

1. gather the right and left edge data from the square subimage array into two buffers
2. send each of the four edge buffers and then the four corner pixels to their respective neighbors
3. calculate interior subimage points
4. receive the first expected edge
5. calculate the corresponding edge points; repeat step 4, in order, for each edge
6. receive the first expected corner
7. calculate the corresponding corner point; repeat step 6, in order, for each edge.

The right and left edge data must be gathered into contiguous memory locations because the DMA capabilities of present communication systems will only transfer contiguous blocks of data.

Initial measurements quickly reveal that the communication phases of the Sobel algorithm can be decoupled from their dependent execution phases sufficiently easily that the specific ordering of communication operations is not critical. That is to say that the nodes may send data to each of their eight neighbors in any order, calculate their interior points, and then receive data from each of their eight neighbors in any order. In all cases the remote data will be received before it is needed. In terms of the model developed in Sec. 3.4, $t_{wi} = 0, \forall i$. Since there is no time spent waiting on blocked receive calls, the analysis of polling costs need not be performed. For this class of algorithms, optimal scheduling of communications is easily achieved.

The times measured for the parameters of the execution time model using the standard Vertex calls are given in Table 3.4 and are discussed below. The time to initiate send operations is given in two parts, *call to send* and *start transmit*, which total to the time for an nwrite of 8 bytes. This value is expressed in two parts to more accurately characterize the time at which the message begins to travel *on-the-wire*. The results are reported for

a 64 × 64 pixel image distributed across 64 nodes. Thus the subimage on each node is 8×8 pixels. Total measured program execution time is $T = 25017\mu$sec. The values of T_c and T_s are 17715 μsec and 5168 μsec, respectively. The approximately 2 milliseconds not accounted for by summing the component times in Table 3.4 is due to the fact that minimum component time values are used and that there are a few source code statements not included in any of the timing loops. The Nread routine is particularly susceptible to underestimation of execution time since the number of arrived, but not yet read, messages that are searched in the process of retrieving a specific message is not accounted for by the model.

There is no contention for communication resources (links and buffers) in this algorithm. Thus, the t_{wi} components of the total execution time are greater than zero only if the mr term of Equation 3.3 is larger than the difference of the times between the initiation of the message send and the message receive (neglecting, as stated earlier, the interrupt overhead times for protocol support for the communication on which t_{wi} is waiting). With $m = 8$ and $r = 1.29$ this clearly cannot be the case for any of the t_{wi}. The t_{wi} terms will be zero even for the limiting case of one pixel per node because the transmission time is insignificant when compared to that of the overheads involved in initiating a communication and handling the resultant protocol interrupts.

The algorithm employed for the Extended Vertex Sobel code consists of the following steps:

1. queue receive requests for each of the edges and corners to be received
2. gather the left and right edge data into contiguous buffers
3. send each of the four edge buffers to its respective neighbor
4. receive the first expected edge
5. polling once for reception of any edge data (if successful, forwarding the corner on to its final destination)

Parameter	time (μs)	Comment
t_{s1}	275	data gather
t_{s2}	160	call to send 1st edge
t_{s3}	20	start transmit
$t_{s3,19:1,1}$	117	send protocol overhead
t_{s4}	160	call to send 2nd edge
t_{s5}	20	start transmit
$t_{s5,20:1,1}$	117	send protocol overhead
t_{s6}	160	call to send 3rd edge
t_{s7}	20	start transmit
$t_{s7,21:1,1}$	117	send protocol overhead
t_{s8}	160	call to send 4th edge
t_{s9}	20	start transmit
$t_{s9,22:1,1}$	117	send protocol overhead
t_{s10}	153	call to send 1st corner
t_{s11}	20	start transmit
$t_{s11,24:1,1}$	117	send protocol overhead
t_{s12}	153	call to send 2nd corner
t_{s13}	20	start transmit
$t_{s13,25:1,1}$	117	send protocol overhead
t_{s14}	153	call to send 3rd corner
t_{s15}	20	start transmit
$t_{s15,26:1,1}$	117	send protocol overhead
t_{s16}	153	call to send 4th corner
t_{s17}	20	start transmit
$t_{s17,27:1,1}$	117	send protocol overhead
t_{c18}	10036	process interior
t_{s19}	170	call to receive 1st edge
$t_{s3,19:1,2}$	186	receive protocol overhead
t_{s20}	170	call to receive 2nd edge
$t_{s5,20:1,2}$	186	receive protocol overhead
t_{s21}	170	call to receive 3rd edge
$t_{s7,21:1,2}$	186	receive protocol overhead
t_{s22}	170	call to receive 4th edge
$t_{s9,22:1,2}$	186	receive protocol overhead
t_{c23}	6681	process edges
t_{s24}	163	call to receive 1st corner
$t_{s11,24:1,2}$	186	receive protocol overhead
t_{s25}	163	call to receive 2nd corner
$t_{s13,25:1,2}$	186	receive protocol overhead
t_{s26}	163	call to receive 3rd corner
$t_{s15,26:1,2}$	186	receive protocol overhead
t_{s27}	163	call to receive 4th corner
$t_{s17,27:1,2}$	186	receive protocol overhead
t_{c28}	998	process corners

Table 3.4. Sobel Execution Profile, Standard Vertex.

6. calculate interior subimage points

7. poll for reception of edges in order (if successful, forwarding the corner and calculating the corresponding edge points), repeat until all edges are calculated

8. poll for reception of the corners in order (if successful, calculate the corresponding corner point), repeat until all corners are calculated.

This main difference between this algorithm and the standard Vertex algorithm is in the handling of the corner pixels. The Vertex extensions forbid non-NN communications. None of the diagonally adjacent subimages are near-neighbors. Therefore, once an edge is received the program attempts to forward the end (corner) pixel to its final destination as soon as possible. Since polling is relatively inexpensive in extended Vertex, a check is made for the reception of edge data before it is needed so that we may forward on corner data if possible.

A more general solution to the problem of receiving and forwarding data in an expeditious manner so as not to delay downstream nodes is provided by one of the options in Extended Vertex. Instead of specifying a flag to be set when an expected message arrives, one can specify a parameterless subroutine to be executed. This subroutine can perform any processing that is necessary, forward the appropriate data, and then set a completion flag for the benefit of the local node before returning. Thus the forwarding of the data occurs as soon as the data is received, rather than at the next poll. Both the first extended Vertex algorithm described, and one employing parameterless forwarding procedures were benchmarked. For the Sobel program both algorithms executed in about the same time, so further results are specified only for the initial algorithm.

The times measured for the parameters of the execution time model using the Extended Vertex calls are given in Table 3.5. These results are also for a 64×64 pixel image distributed across 64 nodes. This time we are less accurate in our modeling of communication transfer start times. Instead of splitting the communication initiation time in two, we leave it as a single time and assume that the communications do not start

on-the-wire until the end of the single step. Since we are, again, well ahead of the point at which we would have to wait on communications, this slightly more coarse modelling of communication start times will have no practical impact other than simplifying table 3.5. Total measured program execution time is $T = 19783\mu$sec. The values of T_c and T_s are 17948 μsec and 1779 μsec, respectively. In this case our model is within 100 μsec of the measured execution time.

The Extended Vertex version of the Sobel algorithm is about 1.26 times faster than the standard Vertex version in total program execution time and 3.06 times faster in communications time. Communication overhead can be lowered even further. Several improvements that require architectural changes will be discussed in Sec. 3.6. However, a significant gain can be easily realized by reducing the number of successive calls to the operating system. The cost of a parameterless operating system call incurred from a C program is 26 μsec. Thus, a 17% improvement in communication performance of the Extended Vertex Sobel code can be realized by coalescing the two groups of repeated calls to the operating system (steps t_{s1}–t_{s8} and t_{s10}–t_{s13}) into just two calls. This improves the speedup to 1.29 times the standard Vertex code.

Linpack LU Factorization. We have also instrumented several versions of the Linpack lower/upper (LU) matrix factorization algorithm. These versions vary in the type of communications, size of matrix and number of processors used. Execution times of these various versions have been profiled. The standard Vertex version of the Linpack LU factorization algorithm is shown in Fig. 3.9 in pseudo-C code, the actual programs were written in Fortran77 by Cleve Moler. The matrix data is distributed, by columns, across the nodes. Node x will contain columns x, $x + N$, $x + 2N$, etc.

The global send operation executes the spanning tree broadcast algorithm described in Sec. 3.1.3. The measured execution of an LU factorization of a 477 × 477 matrix with 64 processors (the largest problem that will currently fit on our NCUBE) using the standard Vertex calls yields $T = 22.475$ seconds and $T_c = 11.524$ seconds. Thus, 10.951

Parameter	time (μs)	Comment
t_{s1}	64	queue 1st edge receive request
t_{s2}	64	queue 2nd edge receive request
t_{s3}	64	queue 3rd edge receive request
t_{s4}	64	queue 4th edge receive request
t_{s5}	64	queue 1st corner receive request
t_{s6}	64	queue 2nd corner receive request
t_{s7}	64	queue 3rd corner receive request
t_{s8}	64	queue 4th corner receive request
t_{s9}	275	data gather
t_{s10}	64	start 1st edge send
$t_{s10,23}$	25	send interrupt overhead
t_{s11}	64	start 2nd edge send
$t_{s11,25}$	25	send interrupt overhead
t_{s12}	64	start 3rd edge send
$t_{s12,27}$	25	send interrupt overhead
t_{s13}	64	start 4th edge send
$t_{s13,29}$	25	send interrupt overhead
t_{s14}	5	poll for edge
t_{s15}	64	send corner
$t_{s15,31}$	25	send interrupt overhead
t_{s16}	5	poll for edge
t_{s17}	64	send corner
$t_{s17,33}$	25	send interrupt overhead
t_{s18}	5	poll for edge
t_{s19}	64	send corner
$t_{s19,35}$	25	send interrupt overhead
t_{s20}	5	poll for edge
t_{s21}	64	send corner
$t_{s21,37}$	25	send interrupt overhead
t_{c22}	10036	process interior
t_{s23}	10	poll for 1st edge (twice)
$t_{s10,23}$	25	receive interrupt overhead
t_{c24}	1743	process edge
t_{s25}	10	poll for 2nd edge (twice)
$t_{s11,25}$	25	receive interrupt overhead
t_{c26}	1743	process edge
t_{s27}	10	poll for 3rd edge (twice)
$t_{s12,27}$	25	receive interrupt overhead
t_{c28}	1743	process edge

Table 3.5. Sobel Execution Profile, Extended Vertex (part 1-of-2).

Parameter	time (μs)	Comment
t_{s29}	10	poll for 4th edge (twice)
$t_{s13,29}$	25	receive interrupt overhead
t_{c30}	1743	process edge
t_{s31}	5	poll for 1st corner
$t_{s15,31}$	25	receive interrupt overhead
t_{c32}	235	process corner
t_{s33}	5	poll for 2nd corner
$t_{s17,33}$	25	receive interrupt overhead
t_{c34}	235	process corner
t_{s35}	5	poll for 3rd corner
$t_{s19,35}$	25	receive interrupt overhead
t_{c36}	235	process corner
t_{s37}	5	poll for 4th corner
$t_{s21,37}$	25	receive interrupt overhead
t_{c38}	235	process corner

Table 3.5. Sobel Execution Profile, Extended Vertex (part 2-of-2).

```
for(i = 1; i <= matrixorder; i++)
{
    if (this node has ith column)
    {
        find pivot index
        calculate multiplier column
    }
    global send
    do row elimination
}
```

Figure 3.9. Linpack LU Factorization (Standard Vertex).

seconds, or 49% of total program execution time is spent either setting up or waiting on communications.

The pseudo-C code description of the Extended Vertex version of the algorithm is given in Fig. 3.10. The major difference in the algorithm is that several steps that were handled by the global send library call in the standard Vertex case have been performed explicitly in this case. This allows the structure of the program to be seen more clearly,

```
do initial receive buffer setups (first two)
for(i = 1; i <= matrixorder; i++)
{
    determine sons in spanning tree and active buffer
    if (this node has ith column)
    {
        find pivot index
        calculate multiplier column
        broadcast to neighbors
    }
    else
    {
        determine father in spanning tree
        await reception of broadcast
        broadcast to sons (if any)
        setup receive (for the one after next)
    }
    do row elimination
}
```

Figure 3.10. Linpack LU Factorization (Extended Vertex).

it also facilitates further experimentation with the design of the algorithm. The standard Vertex program structure could have been left intact, with changes made only to the global send library routine. The difference in performance between these two Extended Vertex versions of the program is negligible (less than one percent). Execution measurements of the 477×477 LU matrix factorization yield $T = 16.786$ seconds. T_c remains the same (11.524 seconds), thus the time spent either setting up or waiting on communications is reduced to 5.262 seconds, or 31% of total program execution time. Total program execution speedup is 1.34, communication speedup is in excess of 2.

The execution time model parameters are given in Table 3.6, where, for the current iteration, x is the number of sons in the broadcast spanning tree, y is the number of active matrix columns and z is the active length of the columns (the active column length decreases by one each time a row is eliminated); i is the current execution step number and d is the dimension of cube on which the program is executed. Table 3.6

Parameter	time (μs)	Comment
t_{s1}	$2(12d+11+63.5)+529$	setup initial receive requests
t_{s2}	$17.4d+19+92.8$	determine sons and active buffer
t_{c3}	$21.26z+280$	determine pivot, calculate multiplier
t_{w4}	0	no wait when we start broadcast
t_{s5}	$14.5x+66$	broadcast
t_{s6}	0	no setup of receive this time
t_{c7}	$y(16.5z+92)+3$	row elimination
t_{s8}	$17.4d+19+92.8$	same as t_{s2}
t_{c9}	0	skip, don't have pivot column
t_{w10}	$t_{ci-7}+\frac{d+1}{2}(t_{si-5}+25+5.2z)$	wait to receive broadcast
t_{s11}	$14.5x+81$ or 15	forward broadcast to sons (if any)
t_{s12}	$12d+278.5$	setup receive (one after next)
t_{c13}	$y(16.5z+92)+3$	same as t_{c7}

Table 3.6. Linpack Execution Profile, Extended Vertex.

shows the execution steps through the first two iterations. The first iteration is one in which the modelled node has the pivot column, the second iteration is one in which it does not. Steps similar to 8–13 are iterated $2^d - 1$ times, followed by an iteration of steps similar to 2–7, and so on until a number of iterations equal to the matrix order of the problem to be solved have executed.

The parameters in Table 3.6 are described below.

t_{s1} Setup the receive requests for the first two iterations for which the node does not contain the pivot column. The parenthesized values are the times to make calls to *father* and *rcvreq*. The father routine determines the father node in the broadcast spanning tree, its execution time is a function of the dimension of the cube plus a small constant for the call/return overhead. The result of father is used as a parameter to the rcvreq call. The remaining time is spent selecting and setting up the buffers for the expected data. Three buffers are preallocated and repeatedly used in a cyclical order.

t_{s2}, t_{s8} The sons in the broadcast spanning tree and the active buffer for the current iteration are determined. The execution time of the *sons* routine is a function of

the dimension of the cube plus a small constant for the call/return overhead. The sons of the current node are used later in t_{s5} and t_{s11}. This time also includes conditional test to determine to determine if the node contains the ith column.

t_{c3} Determine the pivot and calculate the column multipliers. This step is performed only on the node that contains the pivot column. The execution time is a function of the length of the column, which is a function of the iteration step.

t_{s5} Broadcast the column multipliers to each of the sons in the spanning tree. This step is performed only on the node that contains the pivot column. The execution time is a function of the number of sons.

t_{c7}, t_{c13} Eliminate a row of the matrix. This step is performed by all nodes. The execution time is a function of both the length of the columns and the number of active columns located on the node.

t_{w10} This is the time spent waiting to receive the broadcast. This step is performed only on those nodes that do not contain the pivot column. The wait consists of the time for the corresponding node that contains the pivot column to determine the pivot and calculate the column multipliers, plus the product of the average number of hops that the column multiplier broadcast must travel and the time for each hop. The average distance is one-half of the diameter of the cube. The time at each step is comprised of the time required to initiate the broadcast, plus the interrupt overhead, plus the per byte transfer cost.

t_{s11} Relays the broadcast information on to the sons of the current node. If sons exist, the execution time is a function of the number of sons. If not, the execution time is a small constant. This step is performed only on those nodes that do not contain the pivot column.

t_{s12} Setup the receive request for another iteration. This step is performed only on those nodes that do not contain the pivot column for all but the last two iterations. About

half of the work of t_{s1} is repeated in this step.

The model as applied to both the extended and standard Vertex Linpack execution times is accurate to within 5% for all cases between 100×100 and 477×477. In all cases over 50% of the discrepancy is attributable to the T_w term. Even greater accuracy was achieved for all cases of the Extended Vertex Sobel model. Modelled results for a wide variety of problem sizes are given in Sec. 3.6.1.

In the preceding sections the costs of individual communication primitives have been determined, a model that accounts for total program execution time in terms of the individual times of communication operations and calculations has been verified and the potential for significant speedup based upon new operating system calls that can be easily incorporated into existing programs has been demonstrated. The following section will discuss architectural design issues that lead to further performance increases.

3.6 Communications Architecture Support

We have identified five items that should lead to significant performance improvements based upon our examination of the communication performance. These items may be divided into the following two groups: 1) performance improvements that are made directly to the communication system and 2) efforts that are directed towards reducing interference from the communication system on the processor that executes the program computations. Improvements aimed directly at the communication system include:

- minimizing the overhead incurred by invoking communication operations
- increasing the effective system bandwidth by allowing bidirectional communications to proceed simultaneously and
- modifying the hardware design to improve the basic communication system performance.

Changes that are intended to improve the computational performance include:

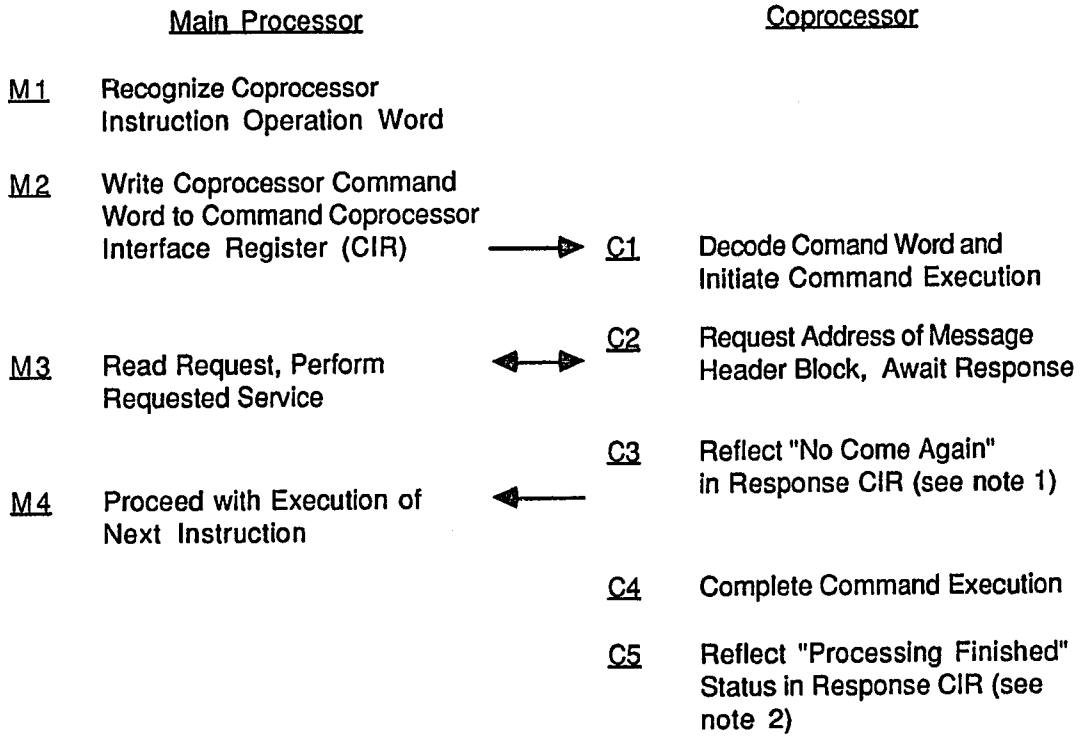
- reducing the cost of communication protocol interrupt handling that must be borne by the compute processor and
- allowing the processing of message data to begin before the entire message has arrived.

The results presented in Sec. 3.5.2 have demonstrated the benefits that are available within the existing communication system architecture by reducing the overhead of communication system calls and returns, reducing the time spent handling the communication protocol interrupts and improving the basic communication system performance. In the following section we will use our program execution time model to predict the performance improvements that would result from a new communication system architecture that addresses all of the items listed above. The details of this new communication system architecture are developed in the next two chapters.

3.6.1 Modelled Results

The new communication system that we are modelling in this section is described in Chapter 5. This section is included in the current chapter so that all of the performance information for the Sobel and Linpack algorithms will be together. It may be desirable to skim this section on the first reading.

Communication instructions in this system are executed by a communication coprocessor. Therefore, the node CPU may be released to continue execution of its instruction stream, after an initial short synchronization with the coprocessor. We assume that the coprocessor interface is like that described in [MC685]. The actions of the main CPU and coprocessor during this short synchronization period for the send, receive and broadcast instructions are given in Fig. 3.11. The message header block is described in Sec. 4.1.1. The synchronization period should last no longer than the time to execute two arithmetic instructions with memory based operands. After this short synchronization period the main CPU is released to continue executing instructions from its instruction stream. Meanwhile, the communication coprocessor can proceed with its actions as described



NOTES:
1. "Come Again" indicates that further service is being requested of the main processor.
2. The main processor may not proceed with M2 of a subsequent instruction until C5 has been completed by the current instruction.

Figure 3.11. Coprocessor Interface Protocol.

in Chapter 5. Typical communication processor instructions (i.e., those that do not have to search through long queues) should complete their initial actions in about the same amount of time as four or five node CPU instructions.

Sobel Edge Detector. The algorithm for the Sobel edge detector using the communication semantics described in Sec. 4.1.1 is similar to the Extended Vertex algorithm. The chief difference is that we can now directly send the corner pixels to their non-NN destinations. This eliminates the *corner forwarding* steps from the Extended Vertex algorithm. Another difference is that the main CPU is no longer interrupted to indicate the completion of message actions. The algorithm employed for the new communication semantics consists of executing asynchronous receive instructions for each of the

edges and corners to be received, gathering the left and right edge data into contiguous buffers, sending each of the four edge buffers and the four corner pixels to its respective neighbors, calculating interior subimage points, polling for reception of edges in order (if successful, calculating the corresponding edge points), repeating the last step until all edges are calculated, then performing steps similar to the last two for the corners.

Again, no time is spent waiting for message data to arrive. Based on data in Chapter 5 the time to transmit the edge data should be less than 22 μsec (after accounting for the NCUBE clock rate). Even after waiting behind outgoing edge data, less than 37 μsec is required to transmit corner pixels. Thus, the time required to process the interior of the message (t_{c22}) is, again, much greater than the time to transmit either the edge or the corner data among neighbors.

The times for the parameters of the execution time model using our new communication system are given in Table 3.7. The predicted total execution time for a 64×64 pixel image distributed across 64 nodes is $T = 18327\mu$sec. This time is about 80% of the time predicted for the standard Vertex algorithm executing on the NCUBE, and about 93% of the time predicted for the Extended Vertex algorithm. The value of T_c is reduced to 379μsec as compared to 5168μsec in the standard Vertex case and 1779μsec for Extended Vertex. The value of T_c could be reduced further to 104μsec if our proposed architecture were modified to handle DMA strides other than one. Such a modification is easy to accommodate, though it is not apparent that it is required often enough to justify the change. With the present proposed architecture the communication time is only 2% of total program execution time. This is substantially better than the 29% for standard Vertex or even the 9% for Extended Vertex. This program does not take advantage of the ability to process data in a message that is still arriving, however, the following program will.

Linpack LU Factorization. The algorithm for the Linpack LU factorization using the communication semantics presented in the following chapter lies between the extended

Parameter	time (μs)	Comment
t_{s1}	4	queue 1st edge receive request
t_{s2}	4	queue 2nd edge receive request
t_{s3}	4	queue 3rd edge receive request
t_{s4}	4	queue 4th edge receive request
t_{s5}	4	queue 1st corner receive request
t_{s6}	4	queue 2nd corner receive request
t_{s7}	4	queue 3rd corner receive request
t_{s8}	4	queue 4th corner receive request
t_{s9}	275	data gather
t_{s10}	4	send 1st edge
t_{s11}	4	send 2nd edge
t_{s12}	4	send 3rd edge
t_{s13}	4	send 4th edge
t_{s15}	4	send corner
t_{s17}	4	send corner
t_{s19}	4	send corner
t_{s21}	4	send corner
t_{c22}	10036	process interior
t_{s23}	5	poll for 1st edge
t_{c24}	1743	process edge
t_{s25}	5	poll for 2nd edge
t_{c26}	1743	process edge
t_{s27}	5	poll for 3rd edge
t_{c28}	1743	process edge
t_{s29}	5	poll for 4th edge
t_{c30}	1743	process edge
t_{s31}	5	poll for 1st corner
t_{c32}	235	process corner
t_{s33}	5	poll for 2nd corner
t_{c34}	235	process corner
t_{s35}	5	poll for 3rd corner
t_{c36}	235	process corner
t_{s37}	5	poll for 4th corner
t_{c38}	235	process corner

Table 3.7. Sobel Execution Profile, Proposed Scheme.

and standard Vertex versions of the code in complexity. The pseudo-C code description of this algorithm is given in Fig. 3.12. Many of the operations necessary to setup and forward the broadcasts of the column multiplier in the Extended Vertex case are no

```
do initial receive buffer setups (first two)
for(i = 1; i <= matrixorder; i++)
{
    determine active buffer
    if (this node has ith column)
    {
        find pivot index
        calculate multiplier column
        broadcast to neighbors
    }
    else
    {
        await reception of broadcast
        setup receive (for the one after next)
    }
    do row elimination
}
```

Figure 3.12. Linpack LU Factorization (Proposed Scheme).

longer needed. This results in two fewer entries in Table 3.8 which gives the times for the parameters of the execution time model using our proposed communication system. However, the repetition and structure of the parameters in this table are similar to those of Table 3.6 which were discussed in the previous section. A notable difference is in the t_{w9} term. First of all, the per hop portion of the term is multiplied by $\frac{3}{4}$ of the cube diameter instead of $\frac{1}{2}$ of the diameter. This additional amount of time accounts for the chance that the broadcast message for a particular node may have to wait behind a copy of the same message that is destined for some other node. The specific actions of the broadcast message for this communication scheme are described in Chapter 5. Secondly, the main CPU no longer spends time servicing communication interrupts nor does it incur the overhead of forwarding the broadcast message. Finally, our proposed scheme permits the processing of message data to begin after the first packet (containing seven data bytes) of the message has arrived. The execution times of all steps that include execution of communication instructions are significantly reduced.

Parameter	time (μs)	Comment
t_{s1}	$2(4)+529$	setup initial receive requests
t_{s2}	92.8	determine active buffer
t_{c3}	$21.26z+280$	determine pivot, calculate multiplier
t_{w4}	0	no wait when we start broadcast
t_{s5}	$2x+2$	broadcast
t_{c6}	$y(16.5z+92)+3$	row elimination
t_{s7}	92.8	same as t_{s2}
t_{c8}	0	skip, don't have pivot column
t_{w9}	$t_{ci-6}+\frac{3(d+1)}{4}7(5.2)$	wait to receive broadcast
t_{s10}	$4+100$	setup receive (one after next)
t_{c11}	$y(16.5z+92)+3$	same as t_{c6}

Table 3.8. Linpack Execution Profile, (Proposed Scheme).

The modelled results for the standard and Extended Vertex codes, along with the results using our proposed communication scheme are given in Table 3.9. Recall that d is the dimension of the cube and m is the order of the matrix. The model indicates that the Extended Vertex code executes from 1.29 to 1.7 times faster than the code that used the standard Vertex calls. The improvements for the proposed communication scheme range from 1.53 to 2.23.

Another performance metric that is useful for evaluating parallel computer systems is the effective processor utilization, U. Effective processor utilization provides a rough indication of the price/performance value of a given system. There are two common definitions for effective processor utilization. One definition is given as follows, let S_N be the speedup achieved by solving a problem on N processors as compared to the time to solve the same problem one processor, U is then given by $U = \frac{S_N}{N}$. This definition of U has two limitations, one is that parallel algorithms are often structured differently for various ranges of N. This is certainly the typical case for $N = 1$ as compared to $N > 1$; however, it is also often the case for other ranges of N (see [FO88,FMO*87], for example). The second problem is that the data set of a large problem is often distributed across the memories of all of the processors in a large multiprocessor sys-

Version	d	m	$T_s(\mu s)$	$T_w(\mu s)$	$T_c(\mu s)$	$T(\mu s)$	Speedup
standard	4	50	28.1×10^3	146×10^3	88.4×10^3	263×10^3	1.00
extended	4	50	28.0×10^3	68.4×10^3	88.4×10^3	185×10^3	1.42
proposed	4	50	10.7×10^3	42.8×10^3	88.4×10^3	142×10^3	1.85
standard	4	100	65.3×10^3	435×10^3	498×10^3	999×10^3	1.00
extended	4	100	55.6×10^3	136×10^3	498×10^3	772×10^3	1.29
proposed	4	100	20.9×10^3	8.12×10^3	498×10^3	655×10^3	1.53
standard	7	500	837×10^3	13.9×10^6	6.66×10^6	21.4×10^6	1.00
extended	7	500	329×10^3	5.63×10^6	6.66×10^6	12.6×10^6	1.70
proposed	7	500	106×10^3	2.88×10^6	6.66×10^6	9.65×10^6	2.22
standard	7	1000	2.67×10^6	52.4×10^6	49.0×10^6	104×10^6	1.00
extended	7	1000	657×10^3	21.7×10^6	49.0×10^6	71.4×10^6	1.46
proposed	7	1000	212×10^3	11.0×10^6	49.0×10^6	60.3×10^6	1.72
standard	10	5000	59.7×10^6	1.50×10^9	793×10^6	2.36×10^9	1.00
extended	10	5000	3.74×10^6	628×10^6	793×10^6	1.42×10^9	1.66
proposed	10	5000	1.06×10^6	268×10^6	793×10^6	1.06×10^9	2.23
standard	10	10000	219×10^6	5.97×10^9	5.97×10^9	12.2×10^9	1.00
extended	10	10000	7.48×10^6	2.50×10^9	5.97×10^9	8.48×10^9	1.44
proposed	10	10000	2.12×10^6	1.07×10^9	5.97×10^9	7.04×10^9	1.73

Table 3.9. Linpack Performance for Standard and Extended Vertex and the Proposed Communication Scheme.

tem [MWP*87,CT87,Wal88,GMB88]. The significance of many programs is derived from the size of their data sets. Restricting the program size to fit on a smaller system can lead to uninteresting results, especially for programs whose execution times scale in a complex manner with either data size or the number of processors. Many large simulation programs fall into this group. Therefore, we prefer to define effective processor utilization as $U = \frac{T_c}{T}$, which corresponds to the fraction of total program execution time that is spent calculating results.

Effective processor utilization values for the standard and Extended Vertex codes, along with the results using our proposed communication scheme are given in Table 3.10. Our proposed communication scheme yields roughly twice the processor utilization of the standard Vertex code. The processor utilization for the Extended Vertex cases typically lies about halfway between that of standard Vertex and our proposed scheme.

Version	d	m	Utilization
standard	4	50	.34
extended	4	50	.48
proposed	4	50	.62
standard	4	100	.50
extended	4	100	.65
proposed	4	100	.76
standard	7	500	.31
extended	7	500	.53
proposed	7	500	.69
standard	7	1000	.47
extended	7	1000	.69
proposed	7	1000	.81
standard	10	5000	.34
extended	10	5000	.56
proposed	10	5000	.75
standard	10	10000	.49
extended	10	10000	.70
proposed	10	10000	.85

Table 3.10. Linpack Effective Processor Utilization for Standard and Extended Vertex and the Proposed Communication Scheme.

3.7 Chapter Highlights

We have shown that improvements to the communication system software can yield significant performance improvements on an existing architecture. The primary cost of this improvement is that it is limited to programs or program phases that are C-deterministic. On the NCUBE architecture, C-determinism requires that both the order and the length of expected messages be known before their arrival. The modest changes to the existing architecture that were noted in Sec. 3.3 could eliminate the restriction of this first class of performance improvements to C-deterministic programs. However, as will be discussed in the following chapter, the message transport strategy that is employed by first generation hypercube systems such as the NCUBE has other severe performance limitations. The communication architecture that we propose in Sec. 4.5 employs a novel message transport strategy that provides significantly better performance and provides the

robustness necessary to handle non-C-deterministic programs.

As a final observation we note that the asynchronous communication semantics that are available in both our Extended Vertex and the communication scheme that we propose in Sec. 4.5 allow many programs to be restructured. Such restructuring can lead to further performance improvements. As an example, a large part of the execution time of the Linpack code is spent waiting to receive the pivot column broadcast. Much of this time can be attributed to all but one processor idly waiting for the pivot column multipliers to be calculated and broadcast. With asynchronous communication primitives, the broadcast waiting cost can be mitigated by effectively overlapping the calculation of the pivot column multipliers and broadcast with the row elimination step of the previous iteration. Consider the actions of a node that has the pivot column for iteration k. Once this node performs the row elimination step for its first column on iteration $k-1$, it can begin to calculate the pivot column multipliers for iteration k. These multipliers can be broadcast, and then the remainder of the row elimination steps (for the remaining columns) for iteration $k-1$ can be completed. Meanwhile, the other nodes are performing row elimination steps for all of their active columns. As long as the time to perform the row eliminations on all of the active columns exceeds the time to perform the elimination on the first column plus the time to calculate and start the broadcasts of the pivot column multipliers, the cost of waiting on the broadcast is substantially reduced. The pseudo-C code for this modification is shown in Fig. 3.13.

```
do initial receive buffer setups (first two)
if (this node has first column)
{
    find pivot index
    calculate multiplier column
    broadcast to neighbors
}
for(i = matrixorder; i > 0; i--)
{
    switch
    {
    case (this node has ith column)
        do row elimination
        break
    case (this node has (i+1)th column)
        await reception of broadcast
        setup receive (for the one after next)
        do row elimination on first column
        find pivot index
        calculate multiplier column
        broadcast to neighbors
        do row elimination on remainder of columns
        break
    case (any other)
        await reception of broadcast
        setup receive (for the one after next)
        do row elimination
        break
    }
}
```

Figure 3.13. Linpack LU Factorization (Asynchronous Broadcast, Proposed Scheme Semantics).

CHAPTER 4

RANDOM COMMUNICATIONS

We consider the more general communication issues involved in supporting messages that may pass by several nodes *en route* to their destinations in this chapter. A description of software interface requirements to support RA (random) and BC (broadcast) communications is given first. Several different message transport mechanisms are then discussed. This is followed by descriptions of representative existing systems. An approach for investigating the performance of different system design tradeoffs is developed and used. The results of this performance investigation are analyzed and a new design is offered and evaluated. The implementation of this new design is discussed in Chapter 5.

4.1 Software Communication Environment Issues

We identify three requirements of parallel software that we would like to see met by the underlying communication architecture in this section. The first of these is support for arbitrary length messages. This is a basic requirement of most communication systems. At the level of the application program, one program statement should suffice to send a message of any size to any destination node. System software must take the responsibility for providing the appearance of this to the application program if support for such messages is not provided by the architecture. This would then require the system software on the sending node to break the message into pieces that are small enough for the underlying architecture—system software on the receiving side would then have to

reassemble the pieces of the message. Several time consuming context switches would be required. Thus, it is highly desirable that support for arbitrarily large messages be provided by the architecture.

The second requirement is that communication operations be handled with a minimal number of user executable machine instructions. Typically such operations are performed in an operating system call. The motivations for performing operations within the context of operating system calls are to provide security and to facilitate the sharing of resources. Neither security nor sharing are requirements of single user operating systems. If multitasking is demonstrated to be cost effective in a hypercube environment, the communication instruction semantics that we offer below can handle multiple simultaneous communications without direct operating system intervention. The architectural support for multitasking consists of informing the operating system at the end of message send and receive operations so that task scheduling operations may be performed if they are necessary. In essence, we would like to have communication operations treated like coprocessor floating point instructions. This is motivated by a desire to eliminate context switch and call/return overhead for communications that is incurred on existing systems.

Finally, we would like to be able to use message data as it arrives, rather than having to wait until the entire message is received. Many parallel algorithms access message data in sequential order. For such algorithms, especially those that receive large messages, this allows us to further overlap calculation with computation. The major design consequence of this scheme is that the communication system need only maintain a point-to-point bandwidth that is equal or greater than the consumption rate of data. Once this bandwidth requirement is met, the remaining design decisions can be made to minimize message latency. Clearly, though, for such a scheme to be useable, we must have a mechanism to prevent the accessing of data before it has arrived.

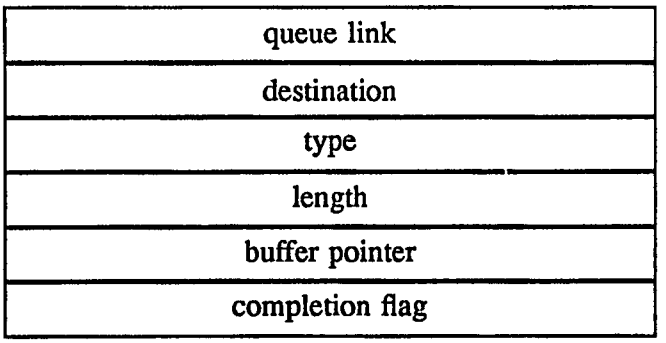


Figure 4.1. Send header block.

4.1.1 Communication Instruction Semantics

The instruction semantics that we propose below assume that the communication processor is implemented as a coprocessor meeting an interface specification like that of the Motorola 68020 [MC685] family. The instruction set architecture of the communication coprocessor provides a basic and efficient interface. This interface is intended to be useable with a minimum of compiler and library support.

Send. The send instruction requires a pointer to a message header block in which the user specifies message destination, type and length values, and a pointer to the message buffer. This block should also contain space for a completion flag and a queue link for use by the communication processor. If the communication processor cannot immediately handle the request, the message header block is enqueued in a wait list for later processing. The completion flag is set to 1 once the message send is in progress, and to 3 once the send is complete. A diagram of the message header block for the send instruction is shown in Fig. 4.1.

Broadcast. Broadcasts occur frequently in numeric codes. Several efficient algorithms are known [HJ86,SWar]. However, none have yet been incorporated into hypercube hardware designs. An integrated architectural solution would eliminate the need to involve

queue link
type
length
buffer pointer
completion flag

Figure 4.2. Broadcast header block.

node CPUs in broadcasting messages. It would also lessen the bandwidth requirement between the communication network and the node memories because the need to forward the message from the node memory is eliminated. The fields within the message header block and the completion flag actions for the broadcast are similar to those for the send instruction, except that a destination field is no longer required. Message transmission will proceed in approximate synchrony as outbound ports become available. This synchrony arises from the fact that the transfer from node memory to communication chip occurs only once, regardless of the number of communication processor output ports that the message is sent through. Broadcast messages will have specially tagged headers so that later communication chips will be able to recognize the potential need to forward the packet through more than one output port. A diagram of the message header block for the broadcast instruction is shown in Fig. 4.2.

The special case of broadcasts to near-neighbors will be handled with a separate instruction. Near-neighbors are nodes that lie one link away from each other. The number of near-neighbors of a node is equal to n, the degree of the cube. For this instruction, we require a list of near-neighbor destinations and a value ($\leq n$) indicating the number of entries in the list, rather than the implicit destinations assumed for the cube-wide broadcast instruction described above. Therefore, our NN broadcast can send messages to a subset of its immediate neighbors. The remainder of the instruction arguments and the completion flag actions are similar to the send instruction. Message transmission will,

queue link
type
length
buffer pointer
completion flag
number of destinations
destination 1
...
destination n

Figure 4.3. Near neighbor broadcast header block.

again, proceed in approximate synchrony as described above for the general broadcast case. A diagram of the message header block for the near-neighbor broadcast instruction is shown in Fig. 4.3.

Receive. The receive instruction also requires a pointer to a message header block. In this block, the user specifies message source and type values (either of which may assume the value *ANY*), the length of the message buffer, and a pointer to the message buffer. This block should also reserve space for the completion flag, two queue links, and two extra data fields for use by the communication processor. The use of the second queue link and the extra data fields is explained in Sec. 5.1.2. A null message pointer indicates that the user would like to use a system allocated message buffer. This is useful in a few cases: when the user suspects that the message may have already begun to arrive and does not want to incur a block move overhead; when the user does not want to be burdened with managing message buffers; and when the length of the incoming message is unknown. In such cases, the communication processor writes the address of the message buffer and its length into the message header block and the user assumes responsibility for returning the buffer space, via the buffer release instruction, to the system. When a valid message

buffer pointer is provided, the specified message buffer will be used. If the message has already begun to arrive, it will be copied from the system buffer to the user specified buffer. The completion flag is set to 1 once the message receive is in progress and to 3 once the reception is complete.

The completion flag and message overrun exception mechanism allow the user to begin processing arriving data before the message reception is complete. This is accomplished by waiting for the completion flag to indicate receive-in-progress before starting to process the data. An exception will occur if a memory request is made for data that has not yet arrived. This request may be continually retried until the data arrives. The more conservative user may just wait for the message completion flag to indicate that the reception is complete before processing the message. The flags are specified such that a process waiting for the receive-in-progress indication may also be released by the reception-completed flag. In a multitasking environment, repeatedly retrying a memory request or repeatedly testing a flag may not be acceptable mechanisms for waiting on asynchronous events like message reception. In such a case, the operating system should provide a signalling service that allows processes to suspend themselves pending the occurrence of an asynchronous event (for example, see [Bac86]). A diagram for the message header block for the receive instruction is shown in Fig. 4.4

Buffer Release. System buffer space is returned via the buffer release instruction. The instruction requires only a single operand that points to the buffer.

4.2 Message Transport Strategies

Perhaps the most fundamental architectural design decision to be made in support of RA communications is the choice of message transport strategy. At the highest level of abstraction, there are two choices to consider: store-and-forward and circuit switching. The prime conceptual example of circuit switching is the telephone system. First, a circuit is established (dial a number); then an arbitrary amount of information is exchanged

queue link
source
type
length
buffer pointer
completion flag
queue link
extra field 1
extra field 2

Figure 4.4. Receive header block.

(talk); finally, the circuit is released (hang up). Once the circuit is established, all subsequent data follows the same path through the network, thus incurring no further routing overhead. Physical resources, chiefly in the form of network bandwidth, are reserved for the user from the time the circuit is established until it is released, even if there are long pauses in the conversation. Circuits can be very wasteful unless they are established and released on a message by message basis. This is especially true for multicomputer network communications which tend to be rather bursty. Another problem is that as the average circuit length increases, fewer circuits can simultaneously co-exist. Therefore, the performance of circuit switching under conditions of non-local communications will be significantly affected as message traffic loads get heavy. Further information on circuit switched networks can be found in [Joe79,MGN79].

Store-and-forward systems can be divided into three specific transport mechanisms: datagram, packet switching and virtual circuits. Datagram transport systems require that a buffer be allocated at each intermediate node along the path from source to destination. Datagrams (messages) of varying sizes can be accommodated. The sending node must be notified when a receiving buffer of the appropriate size has been allocated so that message transmission can begin. The requirement to handle variable size messages adds

significantly to the complexity of buffer management. Typically, a three way handshake protocol like that described in Sec. 3.1.1 is used to provide notification that the buffer allocation has been completed. Routing decisions are made on a hop-by-hop basis for each datagram. The routing decisions at each hop may be either fixed or adaptive. The next hop is determined entirely as a function of the current node and destination node in the case of fixed routing. For adaptive routing, the decision of which node to route to next may be additionally influenced by the perceived traffic loads of the various paths to the destination. Most of the first generation of commercially available large scale hypercube multiprocessors employ a datagram transport mechanism.

Packet switching is very similar to datagram transport. The chief difference is that packet switching uses fixed sized packets and may require several packets to deliver one message. This eliminates much of the buffer management problem associated with datagrams; however, it incurs an additional routing expense for each of the *extra* packets required for the message. If adaptive routing is used, packets from the same message may arrive out of order, requiring an additional reassembly expense. In Sec. 4.5, we propose a variation of packet switching that is shown to perform very well.

Virtual circuits have been suggested as a hybrid approach to message transport. They combine the routing efficiency of circuit switching with the dynamic bandwidth allocation of packet switching while incurring an acceptable buffer management overhead [RF87]. Data is divided and sent in fixed sized packets as in packet switching. However, the route (virtual circuit) that the packets will take is established by an initial header packet. The links between adjacent nodes are divided into a set of *virtual channels*. This division may occur via preallocated time slots or it may be demand driven by prepending a channel identification tag on each data packet. Arriving packets are temporarily buffered, then dispatched based upon their channel tag. Issues of routing, flow control and buffer management must be dealt with by the switch architecture. The X-Tree project at the University of California at Berkeley yielded a design and partial implementation of a virtual circuit based multicomputer system [DP78,Fuj83].

An idea known as *virtual cut-through* can lead to significant improvement in any store and forward based message system. The idea is to begin forwarding an arriving packet via an output link as soon as the link is ready. The consequences of using virtual cut-through are that the communication links must be synchronous and that error checking and retransmission must be left to an end-to-end protocol. Because massively parallel systems are typically compact, with physical dimensions on the order of a few feet, synchronous communication links can be easily provided and the expectation of transmission errors is very low. A queuing system analysis of virtual cut-through is given in [KK79]. The simulation results that we provide in Sec. 4.4 verifies the performance improvement.

4.3 Existing Implementations

We will describe two of the better performing commercially available RA message handling systems, the NCUBE and the Intel iPSC/2. The NCUBE is among the best of the first generation of commercially available hypercube computers. The iPSC/2 is the first machine of the second generation of commercially available hypercubes computers. We also consider the Ametek 2010 which is based in part on the Torus Routing Chip project at the California Institute of Technology and Hyperswitch Network project at the Jet Propulsion Laboratory. Though systems incorporating the Hyperswitch are not yet available, several detailed simulations have been run to predict the performance of its various transport schemes [CMP87a,CMP*87b,CMP*88,GR88]).

4.3.1 NCUBE

The key architectural feature of the NCUBE is its 22 on-chip, bit serial, DMA channels. These 22 asynchronous DMA channels provide 11 inbound and 11 outbound communication lines. These lines are paired to provide one bidirectional connection to the host, with the remaining 10 bidirectional connections available to be configured into a cube of up to 10 dimensions. Each DMA channel has independent count and address registers. When a count register is decremented to zero, the DMA activity for that chan-

nel ceases and an interrupt is signalled. An interrupt service routine that is unique to the interrupting channel may then be selected and executed via a vectored interrupt facility. The DMA devices communicate with memory via a common memory interface unit that is shared with other on-chip functional units. Due to pin limitations, the path between the memory interface unit and memory is 16 bits wide. Thus, at a message transmission rate of 7 MHz, an active channel will write or fetch a memory word about every 2.3μs. NCUBE claims that up to nine channels can be simultaneously receiving data without experiencing overruns due to memory contention [HMS*86]. The high scale of integration achieved by placing the communication hardware on chip allows 64 nodes (processors and memory) to be placed onto each board. Thus, a 1024 node system is contained within a single small cabinet. The chief limitation is that the nodes are not expandable to include more memory (512 K-bytes is standard) or additional functional units.

The instruction set interface to the DMA communication capabilities is via four instructions, seven processor status registers, and the vectored interrupt facility. The LPTR, load pointer (address), instruction is used to indicate the starting memory location at which the DMA fetches or stores are to occur. The LCNT, load count, instruction is used to indicate the number of bytes to be fetched or stored. The loading of a non-zero value in the count register has the effect of starting the DMA operation. The BPTR and BCNT (broadcast) instructions are provided to reduce DMA activity. They allow for the transmission of the same set of memory values to an arbitrary set of output channels in a single DMA operation. In all four of these instructions the second operand is used to select the channel or set of channels to be operated on.

A datagram transport service is used for communications. The routing is handled by the software in the end-of-receive DMA interrupt service routine. If the message is not yet at its destination, it is routed to a node that is one step closer. The routing is statically determined. Messages are sent in the direction of the first differing bit in the result of the exclusive-or of the current and destination nodes. For example, a message from node 011010 to node 110100 would always travel, in order, through nodes 011000, 011100

and 010100 before arriving at node 110100. Since virtual cut-through is not employed, messages always arrive in their entirety at each intermediate node before any action is taken on them. It is a further consequence of the three way handshake employed by standard Vertex that only one message may travel on a bidirectional connection at a time.

4.3.2 Intel iPSC/2

The Intel iPSC/2 implements nodes as either single boards or two board sets. This allows more flexibility in the configuration of nodes. Two board nodes may include, for instance, either additional memory or vector processing hardware. The chief limitation of this scale of integration is that total system size is restricted to 128 nodes in four separate cabinets.

The Intel iPSC/2 employs a fixed route circuit switching mechanism to transport messages [Nug88]. All messages follow a fixed route using the *e-cube* [Lan82] routing scheme. This is the same routing algorithm that was described above for the NCUBE. The message routing system appears as an I/O device to the node CPU. Two separate wires are used to provide bidirectional data links between each pair of adjacent nodes as in the NCUBE. The routing mechanism employs a short probe message that attempts to establish a circuit. An acknowledgement is relayed back to the initiator if the circuit is successfully established. Each 16 bits of data that is transmitted is actually wrapped in a 20 bit packet with the additional 4 bits providing control information. Thus, important control information does not wait behind any messages for delivery. One purpose of the control information is to indicate whether the paired link (in the opposite direction) is part of a circuit that has been successfully established or is part of a failed circuit that must be released. The control bits may also indicate that a circuit has completed its function and may be released.

4.3.3 Ametek 2010

The Ametek 2010 system implements a 2-dimensional mesh connected multiprocessor [Ame87a,Ame87b]. Each node may consist of multiple boards incorporating several functional units such as a CPU and floating point coprocessor, vector floating point accelerator, disk interface, memory, and communications interface and routing chip. The maximum system size is claimed to be 1024 nodes. However, a more reasonable maximum size system, in terms of incremental performance gains, is likely to be 256 nodes. Two large cabinets are required to house a 256 node system. A modified wormhole message transport is implemented on top of a routing chip that is partially based upon the torus routing chip (TRC) [DS87,DS86]. Technical details of the Ametek routing chip were not available at the time of publication. In lieu of this, the details of the TRC are given below—after the explanation of wormhole routing.

Wormhole routing can be thought of as a persistent form of circuit routing. As the head of the message is routed through the network, communication resources (i.e., links and buffers) are accumulated. The message data (or body of the worm) immediately follows the head. If the progress of the head of the message is blocked, the flow of the message is halted; the communication resources are not released (attempts at establishing the circuit are persistent). Since communication resources are not released until the message delivery is completed, care must be taken to avoid routing decisions that can lead to deadlocks. Any blockage that the head of the message encounters must be guaranteed to be temporary.

The chief attributes of the torus routing chip are deadlock avoidance and speed. A pair of unidirectional 8-bit wide data paths provide input/output connections between the communications processor and the TRC. The TRCs are interconnected by four unidirectional, 8-bit wide data paths. There is one inbound and one outbound path in each of the X and Y directions. Additionally, four control lines provide flow control for each of the data paths between the TRCs. Two additional control lines provide flow control for the data path between the TRC and node processor. The TRCs function as intelligent

3×3 crossbar switches. The message packets begin with two relative address bytes, one each for the X and Y directions, followed by an arbitrary number of non-zero data bytes. They are terminated by a tail byte of value zero. Messages are routed in the direction of decreasing addresses, first along the X dimension, then along the Y dimension. The destination addresses are specified as a relative distance from the source and are decremented at each routing step. When the correct X coordinate is reached the relative X dimension address is stripped from the packet and routing is initiated along the Y dimension. When the packet arrives at its final destination the relative Y dimension address is stripped and the data and tail byte are routed to the processor.

The TRC internally consists of a 5×5 crossbar with each of the X and Y inputs being internally demultiplexed into one of two internal channels by a self-timed internal state machine. The existence of the two channels facilitates deadlock avoidance by providing routing paths that are free of cycles. The pairs of X and Y output signals are multiplexed into single X and Y outputs in a similar manner. The four control lines for the data paths between TRCs provide separate request and acknowledge lines for each of the two logical internode channels. Only two control lines are needed for the data path between the TRC and node processor since there is only one logical channel on this data path. TRCs have been designed with a routing latency of about 150ns per step.

The Ametek equivalent of the TRC is called the Automatic Message Routing Device (AMRD). Each AMRD is claimed to be able to transmit messages at the rate of 80 M-bytes per second, 20 M-bytes per second on each of four channels. Latency through the AMRD is less than 1 μs. A small amount of "glue" logic sits between the AMRD and the node processor. This logic makes the AMRD appear like intelligent memory to the node processor. Thus, the semantics of the message send operation are very similar to those for freeing a memory buffer. Once the message buffer is freed it is no longer useable by the application program; the system then begins to transmit the message. When the message transmission is complete, the buffer is returned to the pool of free memory space. A system of 256 nodes represents the largest size system that is commonly thought to favor

the faster but longer average message paths that are feasible with the low dimensionality and wide data paths of mesh interconnections. Ametek 2010 systems are expected to be available by mid-1988.

4.3.4 JPL Hyperswitch

The current generation hypercube computer developed at the Jet Propulsion Laboratory is the Mark III [PTLP85]. The Mark III can be configured with up to 128 nodes. Each node has a dedicated I/O control processor (IOCP). Even with the IOCP, the software overhead limits the maximum attainable communication bandwidth between neighboring nodes to 20% of the hardware bandwidth of 100 M-bits per second. Additionally, a large latency (40 μsec) is incurred for each message hop and only one message channel per node can be served at any one time. In order to break these limitations the hyperswitch network (HSN) was developed.

The HSN [CMP87a,CMP*88] is designed to support both packet and circuit switched message transport. This system is designed to optimize the performance of circuit switching, therefore, we will not consider the packet switching operation in this discussion. The HSN has two operating modes: path setup and data transmission. A path setup is required for each message transmission. During this time no data is moving, thus, path setup time is a component of message latency. Two heuristically driven dynamic route finding algorithms are available: K and $K(K-1)$. These algorithms differ in the tenacity with which they will search for an available route. The K algorithm will make at most one attempt to reach all of the penultimate nodes, resulting in an execution time complexity of $O(K)$, where K is the order of the cube. The $K(K-1)$ algorithm will make every possible attempt to reach all of the penultimate nodes, resulting in an execution time complexity of $O(K^2)$. The $K(K-1)$ scheme attempts to take maximum advantage of the high degree of link connectivity, $L!$, that is available between two nodes that lie L hops apart.

Each hyperswitch node contains a bit-wide crossbar switch, K communication channel

interface chips, K hyperswitch chips and three buses. The channel interface chips are implemented in ECL technology and the hyperswitch chips in CMOS. Full connectivity is provided between communication channel inputs and outputs by the crossbar switch. A bidirectional status line accompanies each data link between neighboring nodes. In path setup mode, the status indications are: 1) ready-to-receive header; 2) ready-to-receive data; 3) header backtracked to previous node and 4) header error, retransmit. The original state of all hyperswitches places them in path setup mode. During path setup, the header backtrack signal is raised for the preceding node if the current node cannot acquire the next link in the route. The node that raises the signal will release its circuit resources. The node that receives the signal will either try to route through another neighbor or will propagate the signal to its predecessor. Path setup mode is maintained until a header progresses to its destination. At that time, the status is changed to ready-to-receive data which indicates a transition into data transmission mode. The receive data signal is propagated through the established route to the source. In data transmission mode, the status indications are: 1) break-pipeline to source; 2) ready to receive data; 3) wait for next data packet and 4) data error, retransmit. For flow control purposes, the transmitted data is packetized into packets that range from 4 to 256 bytes in length. The flow of this data is controlled by the ready and wait status signals. When the last byte of a message has been received the destination node begins the propagation of the break-pipeline signal back to the source.

4.4 Message Transport Comparisons

We present the results of a general investigation into design alternatives for message transport mechanisms in the following two sections. We begin by describing our method of investigation.

4.4.1 Investigative Approach

The very nature of programs that employ RA communications typically causes them to fall into either the loosely coupled SCMD or the MCMD classification. These algorithms often lack the tight synchronization that is inherent in most NN and BC programs. While it may be possible to draw certain generalizations about the communication patterns of such programs, the specific destinations and time ordering of communications is typically not known much in advance of their occurrence. Therefore, RA programs lend themselves well to simulations that capture the general pattern of communications by modelling destinations and message initiation times in a pseudo-random fashion. This approach has been used by Reed and Grunwald and is discussed further in [RG87]. We extend the nature of Reed and Grunwald's work by developing an event driven simulator [SC81] that allows us to quickly vary the communication parameters of the machines under test. This allows us to predict the performance of architectural changes such as the use of different message transport schemes (e.g., datagram), or changes that effect the speed of basic communication parameters.

The message transport mechanisms that we compare are datagram, datagram with cut-through [KK79], and circuit switching. Throughout the rest of this section we will refer to *datagram with cut-through* as simply *cut-through.* The specific protocol that we simulate for the datagram with cut-through case is analogous to the one described in [ABG85]. Our implementation of circuit switching is persistent. That is, blocked circuits do not surrender their resources and try again. Messages wait until the blocking condition disappears. In this sense, it is similar to wormhole routing [Dal86a]. With adaptive routing schemes such behavior can lead to deadlock. Several deadlock-free routing schemes exist for store-and-forward transport schemes [Gel81,Gun81,MS80,TU79,Tou80]. A deadlock-free routing scheme for k-ary n-cubes employing wormhole routing has also been developed [DS87]. For our simulation we break potential deadlocks by assigning priorities to messages in order of their creation and requiring that a message never be allowed to block another message of higher priority (earlier creation) unless it is in the process of being received

at the destination. Messages that are being actively received cannot cause deadlocks because they will only hold their resources for a limited amount of time. Though there are several methods for avoiding or breaking deadlocks, we feel that this scheme should yield simulation results that provide a fair basis for comparison. We additionally consider the effects of adaptive routing and independent simultaneous bidirectional communications.

Hypercube communication protocols typically employ some form of handshaking that requires acknowledgement information to be passed from a message destination back to its source. Since the original sending node may not continue to process messages on a given link until any expected acknowledgements are received, true bidirectional communications may be difficult to achieve if the acknowledgement information is blocked behind another message on the return link. For further discussion on this topic see [MBA87]. By allowing independent simultaneous bidirectional communications in our simulation model we can evaluate the benefit of removing acknowledgement or confirmation messages from the link that runs in the opposite direction of the the original message. Such a scheme may be implemented at the expense of additional circuitry at the I/O pads of a custom chip (details are discussed in Sec. 5.4.8). Alternatively, Intel circumvents this problem by interspersing small amounts of control information, which may include acknowledgements, into the data stream of the link that runs in the opposite direction of the original message. JPL avoids this problem at the expense of dedicating separate status lines to carry control information. The simulation results that are described throughout the remainder of this section compare the three transport mechanisms mentioned above. These comparisons are made with both adaptive and fixed routing, and unidirectional and bidirectional links.

In order to save space on the figures, part of the legend used in Figs. 4.6 through 4.27 is described in Fig. 4.5. The results reported in Figs. 4.6 through 4.27 indicate the minimum, average, average plus one standard deviation and maximum times for the set of random test messages to reach their destinations. The average length of messages is abbreviated in the figure captions. It is either a fixed constant value, an exponentially distributed

DCAB	Datagram Cut-Through Transport, Adaptive Routing Bidirectional Links
DCAU	Datagram Cut-Through Transport, Adaptive Routing Unidirectional Links
DCFB	Datagram Cut-Through Transport, Fixed Routing Bidirectional Links
DCFU	Datagram Cut-Through Transport, Fixed Routing Unidirectional Links
CSAB	Circuit Switched Transport, Adaptive Routing Bidirectional Links
CSAU	Circuit Switched Transport, Adaptive Routing Unidirectional Links
CSFB	Circuit Switched Transport, Fixed Routing Bidirectional Links
CSFU	Circuit Switched Transport, Fixed Routing Unidirectional Links
DTAB	Datagram Transport, Adaptive Routing Bidirectional Links
DTAU	Datagram Transport, Adaptive Routing Unidirectional Links
DTFB	Datagram Transport, Fixed Routing Bidirectional Links
DTFU	Datagram Transport, Fixed Routing Unidirectional Links

Figure 4.5. Legend Explanation for Simulation Figures.

random value with the specified mean or a normally distributed random value with the mean and standard deviation expressed as *mean,destination.* The message destination is typically chosen uniformly from the the set of all nodes excluding the source. The exceptions are explicitly discussed.

We must establish a reasonable standard for the execution times of the basic communication operations before attempting to compare the performance of the different message switching strategies that rely on them. For this purpose, there are three basic times that are of interest. One is the amount of time required to transmit a byte of data. The second is the time required to setup the communication links between adjacent nodes. The third is the time required to allocate a storage buffer to hold a message while it is being transmitted. We will use times of 1, 1, and 40 ticks, respectively. The relative transmit and buffer allocation times are similar to times measured on existing NCUBE systems. The link setup time is taken to be a few times slower than the times expected by Dally and Seitz for their TRC chip. These times are intended to provide a reasonable

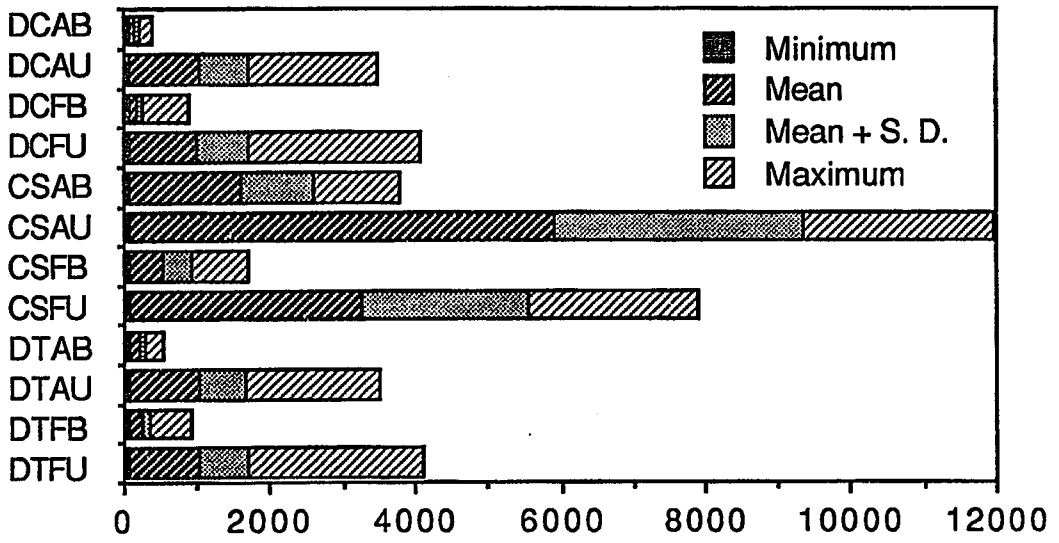


Figure 4.6. Message Time: Flooded Traffic, Len = 16, Dest = uni().

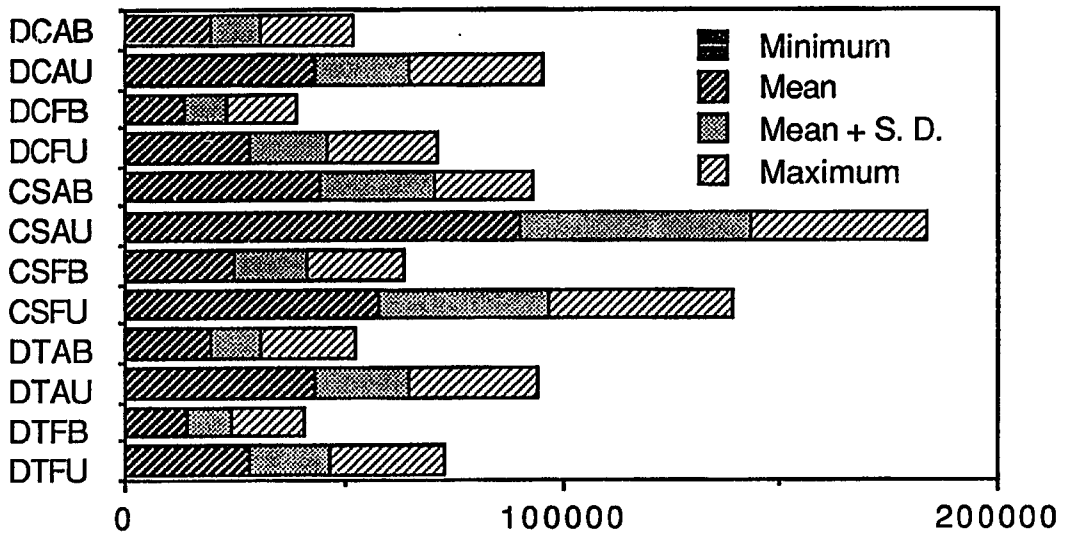


Figure 4.7. Message Time: Flooded Traffic, Len = exp(512), Dest = uni().

basis of comparison for the evaluation of general design alternatives.

4.4.2 Comparitive Observations

We first consider cases in which the communication system is flooded with messages. This situation is defined by a communication traffic load in which the average number

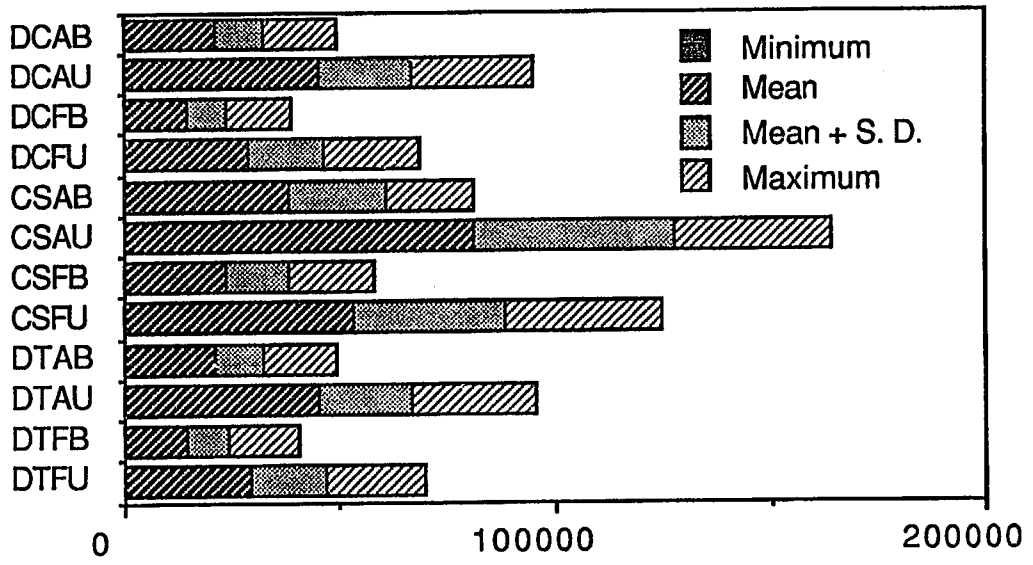


Figure 4.8. Message Time: Flooded Traffic, Len = nor(512, 256), Dest = uni().

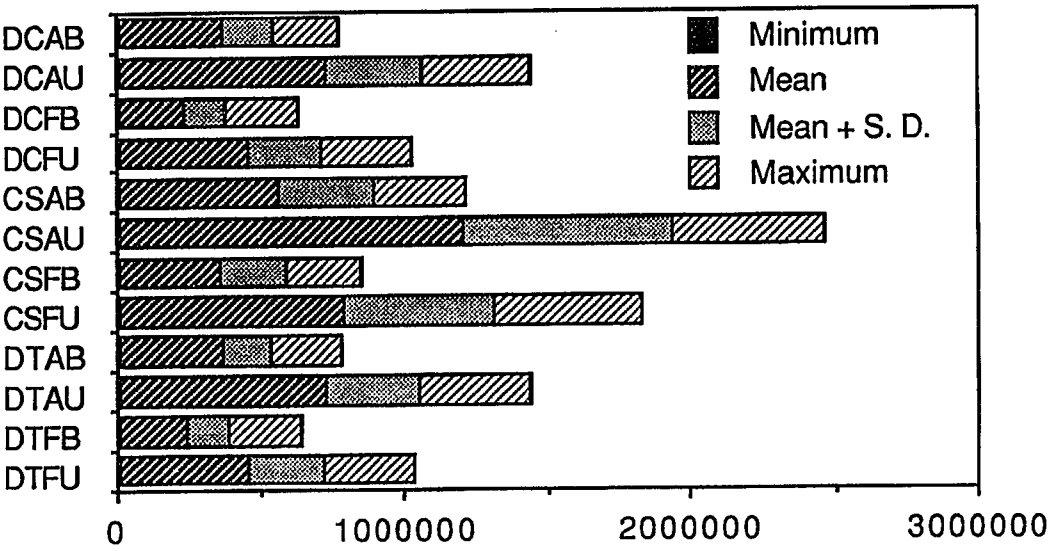


Figure 4.9. Message Time: Flooded Traffic, Len = 8192, Dest = uni().

of messages being processed or awaiting processing is greater than N, the number of processors in the system. Flooding can be achieved by having the message production rate exceed the message delivery rate on each node. In Figs. 4.6–4.10 we compare the performance of datagram, circuit switching, and cut-through transports for messages of

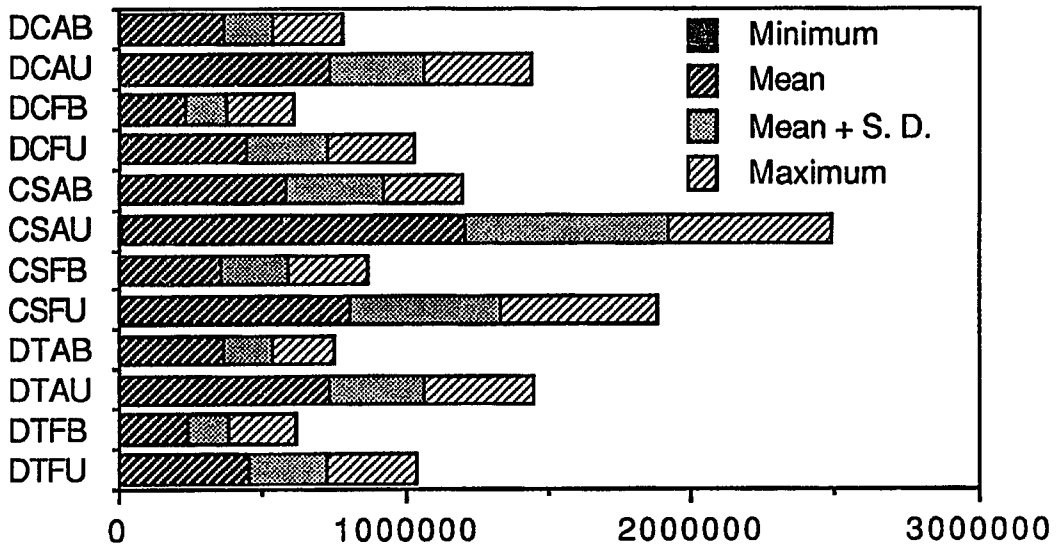


Figure 4.10. Message Time: Flooded Traffic, Len = nor(8192, 2048), Dest = uni().

various size distributions. For each type of transport we consider the effects of adaptive routing and non-interfering bidirectional links. In all cases, a new message is generated every 50 ticks and the message destination distribution function is uniform over all nodes (with the source and destination always being different nodes). Unless otherwise noted, all simulations in this chapter are for hypercubes of degree 6, $N = 64$.

A few major trends can be noted. First, there is essentially no performance difference between datagram and cut-through switching. This is expected since, in the case of extremely flooded conditions, the cut-through algorithm is rarely able to establish a connection of length greater than one. Both cut-through and datagram transports perform consistently and significantly better than circuit switching. This occurs since under heavy message traffic many links are idle but reserved while waiting for a circuit to become fully established. The negative effects of circuit routing are most noticeable for small messages. When comparing the effects of message lengths, there is only a slight performance difference for any of the cases between fixed length messages and those with a normal distribution centered about the length of the fixed messages, see Figs. 4.9 and 4.10 for example. The same is not true when either fixed or normal distributions are

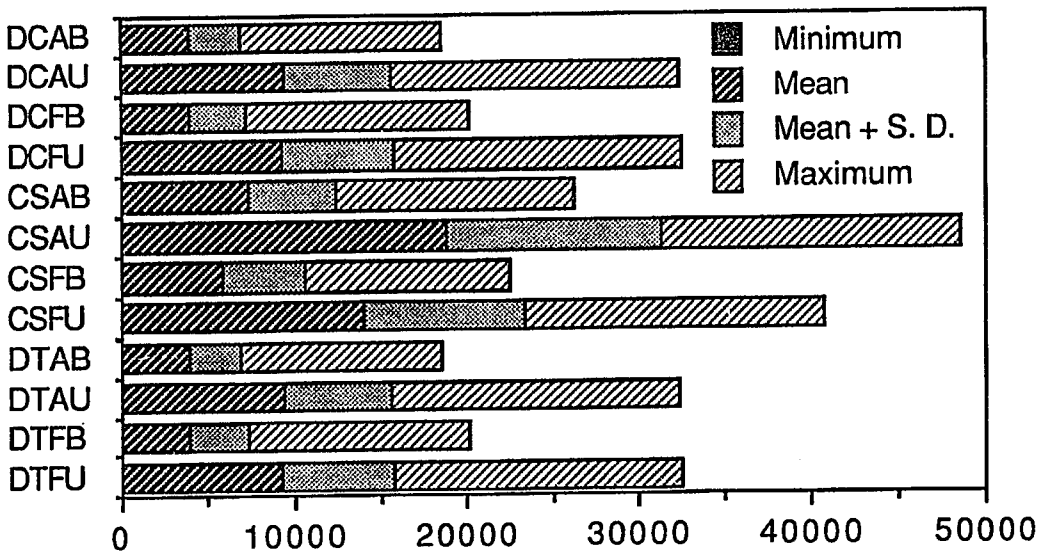


Figure 4.11. Message Time: Flooded Traffic, Len = exp(512), Dest = dpf(0.2).

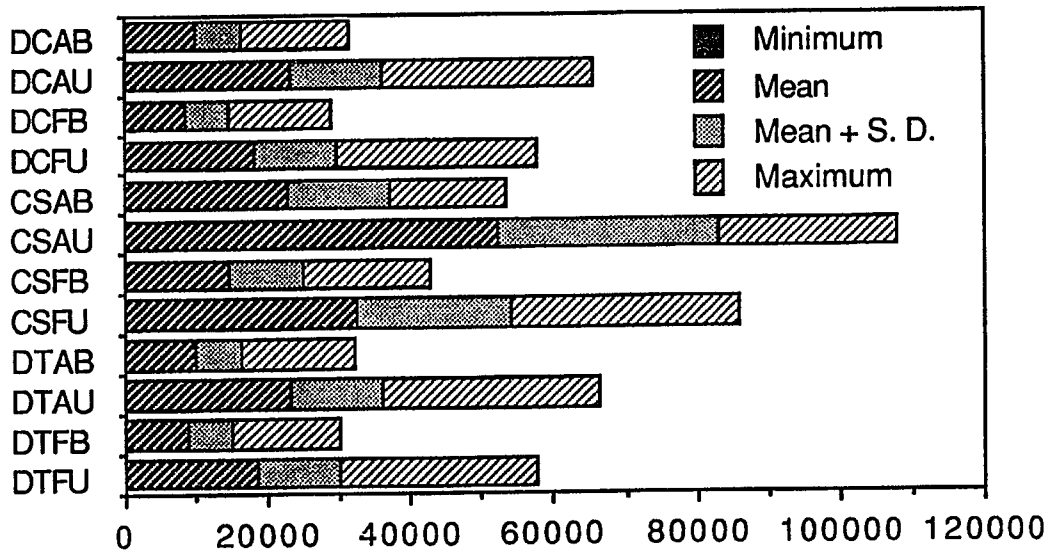


Figure 4.12. Message Time: Flooded Traffic, Len = exp(512), Dest = sl(2, 0.8).

compared with exponential distributions of the same mean, as in Figs. 4.7 and 4.8. This is because the message traffic in the exponential case is dispersed more evenly in time; whereas, with fixed or normal cases we have distinct phases of communication activity.

The use of non-interfering bidirectional links directly contributes to about a doubling in performance in all cases. The performance increase is even better for small mes-

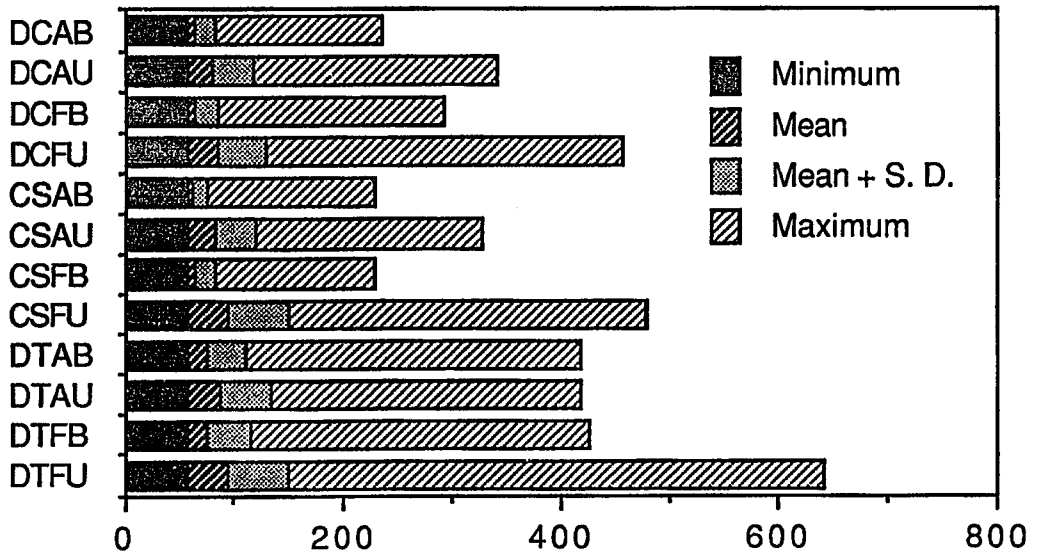


Figure 4.13. Message Time: Flooded Traffic, Len = 16, Dest = dpf(0.2).

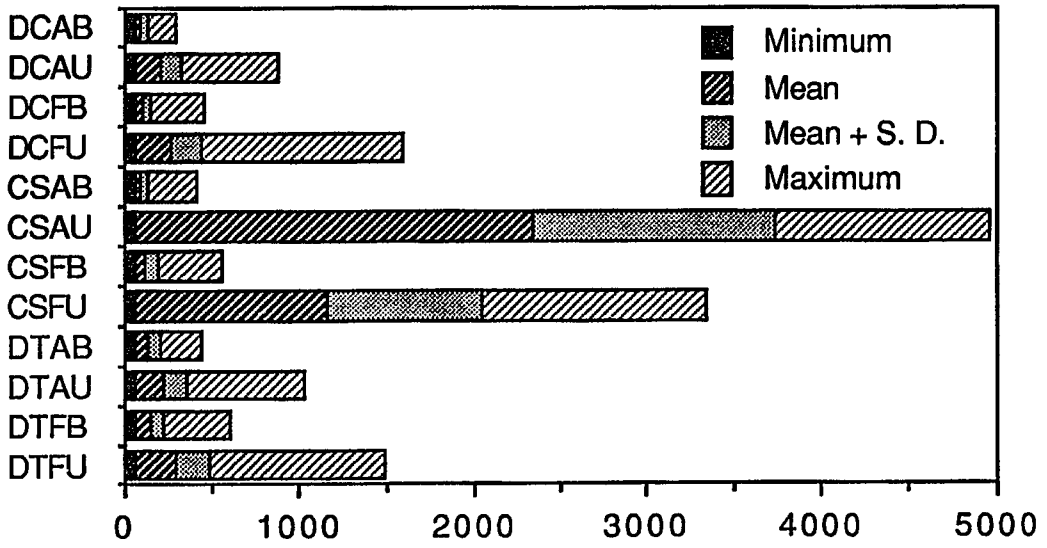


Figure 4.14. Message Time: Flooded Traffic, Len = 16, Dest = sl(2, 0.8).

sages. Conversely, adaptive routing leads to marginal performance improvements at best and often leads to significant performance decreases. This is especially true for circuit switching. Changing the message destination distribution function to exhibit more locality yields results similar to the above for the large average message size cases. For

small messages, circuit switching looks comparatively better, especially for the cases with non-interfering bidirectional links or when the locality is extremely tight. The latter case is demonstrated in Fig. 4.13. In Figs. 4.11 and 4.13, a decreasing probability function with a decay parameter of 0.2 is employed. Both Figs. 4.12 and 4.14 use a sphere of locality function with a radius parameter of 2 and probability parameter of 0.8. These two distribution functions are described below [RG87].

Sphere of Locality Probability Function. Iterative partial differential equation solving algorithms communicate mostly between neighbors, but they periodically send information to a global convergence checker. Communication traffic in algorithms with characteristics similar to this can be modelled by having each node centered within its own sphere of locality. The local region is parameterized by the radius of the sphere (r), that is, the number of communication links a message can cross and still be within its locality. The chance of communicating with a node within your locality is given a rather high probability (say, ϕ), nodes outside of your locality have a $1-\phi$ chance of being communicated with. In a hypercube of degree n, the number of nodes within a sphere of locality is given by:

$$R(r,n) = \sum_{l=1}^{r} L(l,n) \qquad \text{where} \qquad L(l,n) = \frac{n!}{l!(n-l)!}$$

In a hypercube with N nodes, each nodes lies within the locality of R other nodes and outside of the localities of $N-R-1$ nodes. Thus, given values for ϕ and r the message routing probability distribution in terms of l, the distance of the destination from the source, is given by:

$$\phi(l) = \begin{cases} \frac{\phi L(l,n)}{R} & 1 \le l \le r \\ \frac{(1-\phi)L(l,n)}{N-R-1} & r < l \le n \end{cases}$$

One of the L nodes that lies at a distance l from the source is chosen with uniform probability to be the message destination.

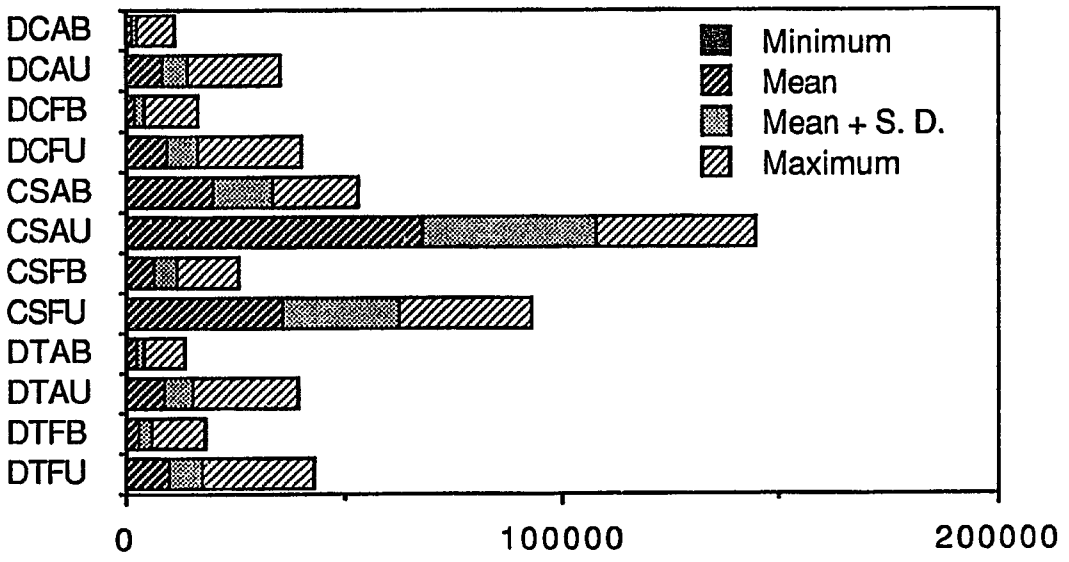


Figure 4.15. Message Time: Freq = exp(512), Len = exp(512), Dest = uni().

Decreasing Probability Function. Another useful message destination distribution function is one in which the probability of sending a message to a node decreases as its distance from the message source increases. A simple and straightforward function of this type is given by:

$$\phi(l) = D(d,n) \cdot d^l \qquad 0 < d < 1,$$

where $D(d,n)$ is a constant to normalize ϕ. This constant is chosen such that:

$$D(d,n) \sum_{l=1}^{n} d^l = 1$$

Again, one of the L nodes that lies at a distance l from the source is chosen with uniform probability to be the message destination.

Performance tradeoffs are next studied under conditions with less intense communications traffic. These comparisons are structured similarly to those above, the chief difference is that the time between the generation of successive messages is varied from a rather heavy communication load to a light, but significant, level.

For moderate and light communication loads (Figs. 4.16, 4.17, 4.19, 4.20, 4.22, and

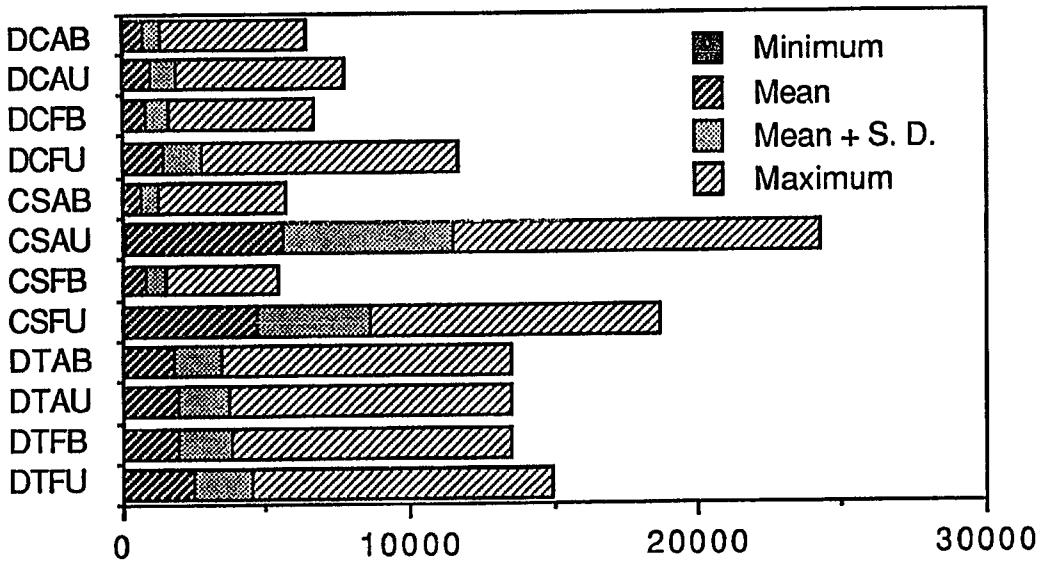


Figure 4.16. Message Time: Freq = exp(1536), Len = exp(512), Dest = uni().

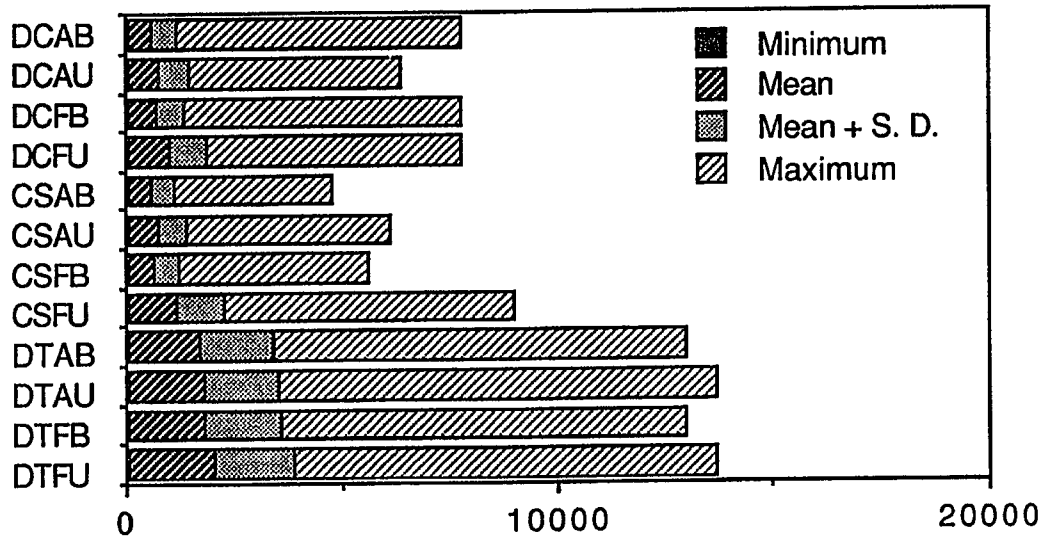


Figure 4.17. Message Time: Freq = exp(2560), Len = exp(512), Dest = uni().

4.23) we see that adaptive routing is useful, though its effect is frequently small. The chief exceptions occur with small packets (Figs. 4.21–4.23). In this case, adaptive routing, in conjunction with circuit switching, again consistently leads to decreases in performance. Circuit switching approaches cut-through performance as traffic loads decrease, as we expect. However, circuit switching performs consistently poorly on small messages,

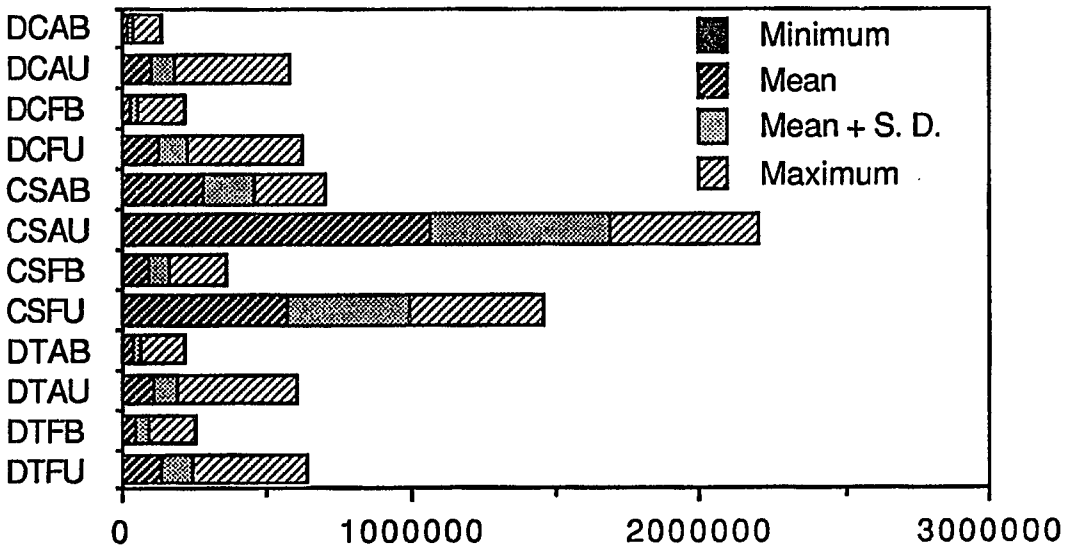


Figure 4.18. Message Time: Freq = exp(8192), Len = exp(8192), Dest = uni().

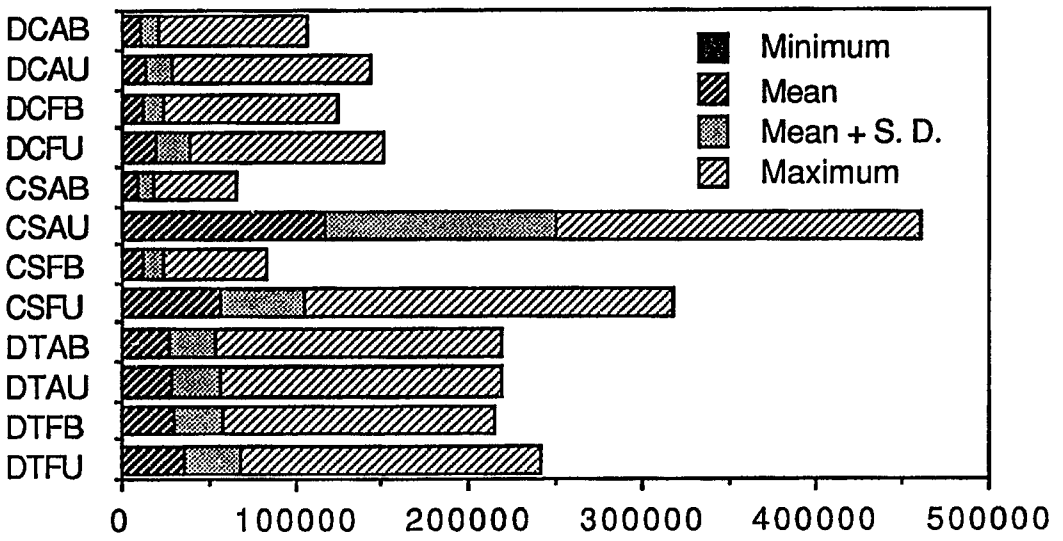


Figure 4.19. Message Time: Freq = exp(24486), Len = exp(8192), Dest = uni().

especially with unidirectional links. An interesting side note is that the maximum values for communication times using circuit switching is less than that for cut-through under moderate traffic conditions for moderate length messages. This anomaly, which can be seen in Fig. 4.16, appears to indicate a point where it is better to patiently wait a small time for a circuit path to clear rather than hastily allocating a buffer and paying the store

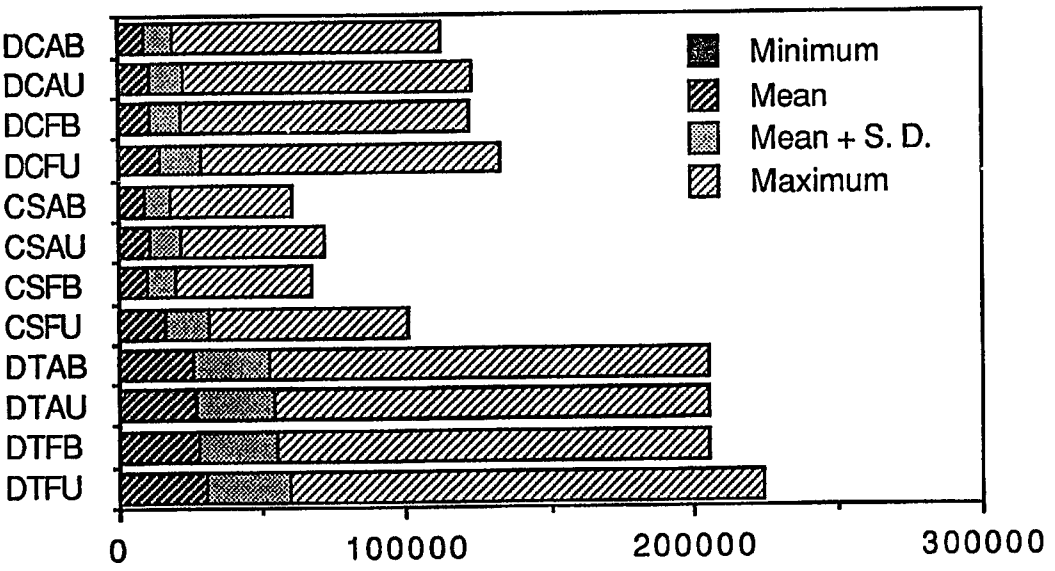


Figure 4.20. Message Time: Freq = exp(40960), Len = exp(8192), Dest = uni().

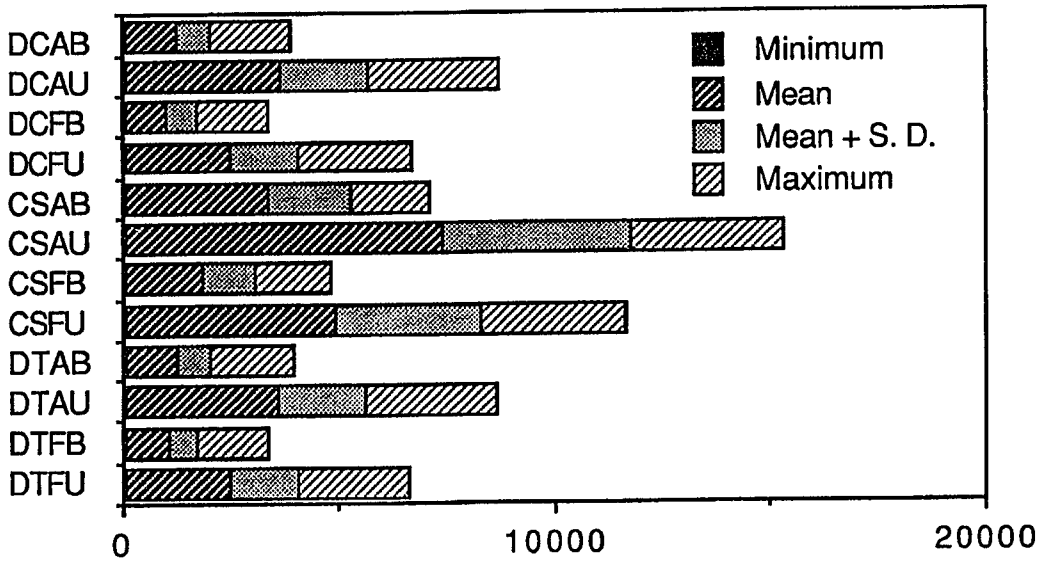


Figure 4.21. Message Time: Freq = exp(16), Len = exp(16), Dest = uni().

before forwarding overhead. This anomaly may be avoided with a more sophisticated switch design that allows cut-through even after a store operation has begun. Datagrams without cut-through generally trail in performance. This is due to the cost associated with storing the data at each intermediate node before it is forwarded on to the next node.

The sensitivity of circuit switching is readily apparent under bursty communication

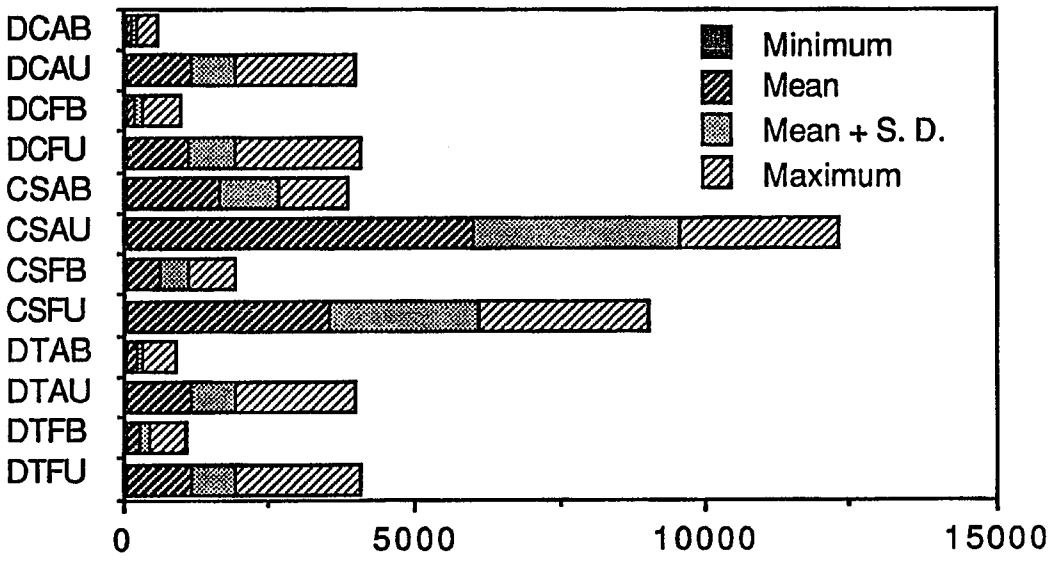


Figure 4.22. Message Time: Freq = exp(48), Len = exp(16), Dest = uni().

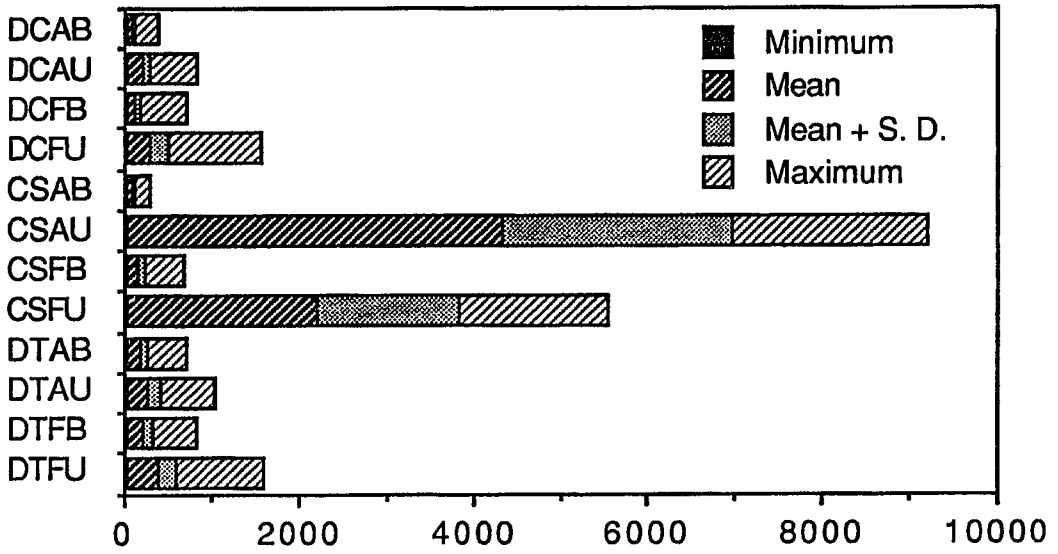


Figure 4.23. Message Time: Freq = exp(80), Len = exp(16), Dest = uni().

loads as illustrated in Figs. 4.24–4.27. In these cases, we have made the time gap between the generation of successive messages constant. This is similar to the type of behavior that we would expect to see in closely synchronized SCMD programs. Circuit routing is, by far, the most sensitive to the onset of periods of heavy message traffic.

All three of the transport schemes that we have evaluated have significant drawbacks when considered by themselves. The primary purpose of this particular investigation was to determine the impact of various design decisions; we did not intend to advocate the adoption of any specific case. Nevertheless, the following general conclusions can be drawn. The inability of circuit switching to gracefully degrade in performance under heavy traffic loads renders it a poor choice for environments where moderate to heavy bursts of traffic are likely to occur. Our simulations show that datagrams, both with and without cut-through, require buffers that are many times larger than the largest message to maintain reasonable performance. Without flow control, maximum buffer requirements can quickly become excessive: ranging from about 100 K-bytes for light loads to in excess of 200 K-bytes for heavier traffic for messages averaging 8192 bytes in length. Clearly, for any store-and-forward based transport scheme to be viable it must employ flow control. Even with flow control, however, datagram based schemes require buffers at least as large as the largest allowable message size at each node. The advantages of cut-through are significant for all but the heaviest traffic conditions. For both of the datagram cases, as well as for circuit switching the most significant performance improvement under moderate to heavy traffic conditions is gained by providing non-interfering bidirectional links. Adaptive routing was also helpful for datagrams for all but heavy traffic cases with small messages. This was particularly the case when adaptive routing was used in conjunction with non-interfering bidirectional links. For small messages under heavy traffic loads, adaptive routing lead to performance decreases which were particularly significant for the uni-link case. The benefits of adaptive routing were typically much smaller than those of bidirectional links for all but the light traffic cases. Adaptive routing was not as helpful when used in conjunction with circuit switching. Performance decreases occurred for small messages under all traffic loads. When used without bidirectional links, performance decreased for medium and large messages under moderate and heavy traffic loads as well.

Message transport tradeoffs are also being studied by Reed and Grunwald [GR88].

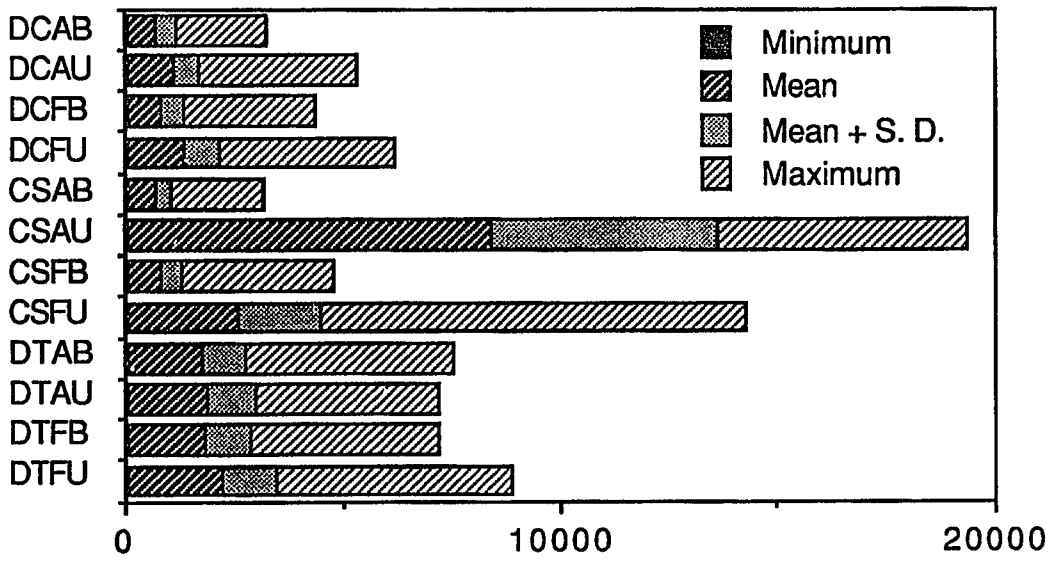


Figure 4.24. Message Time: Freq = 1536, Len = nor(512,256), Dest = uni().

Their work includes evaluating the different routing mechanisms that are possible with the hyperswitch in JPL Mark III computers. The general observations from their preliminary results appear to coincide with ours. In particular, for moderate to light traffic loads they show the best results for wormhole and an adaptive form of circuit routing known as $K(K-1)$. The $K(K-1)$ scheme provides a small but consistent performance improvement over wormhole routing.

4.5 Communication Processor Design Issues

We consider communication processor design issues to encompass the instruction set specification, choice of message transport mechanism, the division of labor and level of interaction with other node components, and the design of the internal architecture. The instruction set specification was given in Sec. 4.1. The design of the internal architecture, along with the details of our proposed design, are discussed in Chapter 5.

Message transport performance for RA communications is improved the most by providing non-interfering bidirectional communications in moderate to heavy traffic and

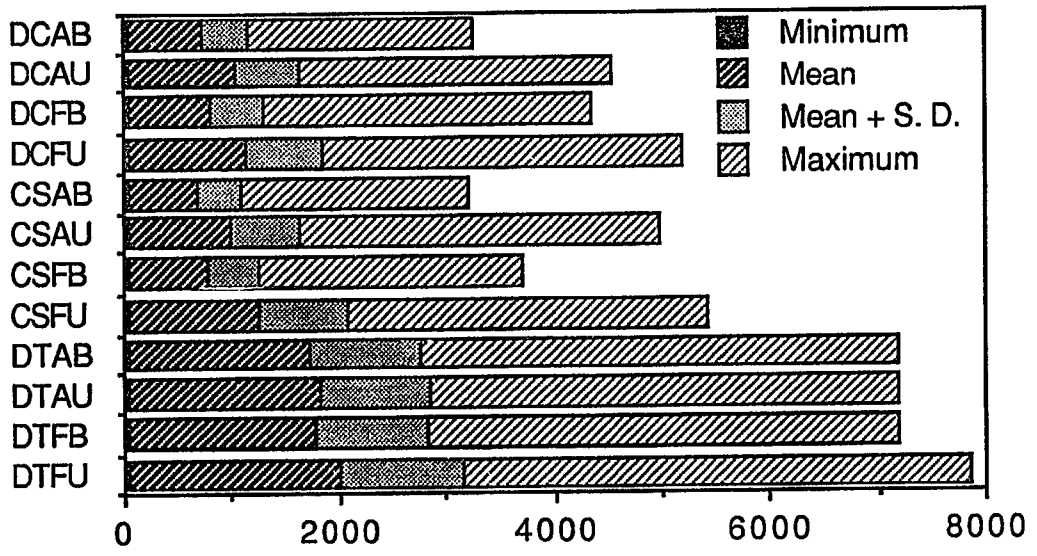


Figure 4.25. Message Time: Freq = 2560, Len = nor(512,256), Dest = uni().

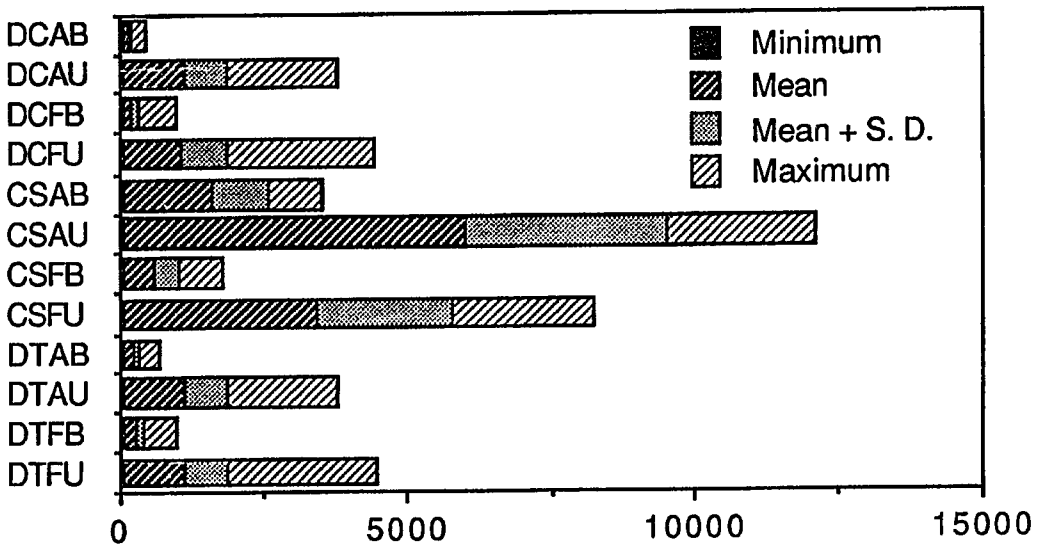


Figure 4.26. Message Time: Freq = 48, Len = nor(16,8), Dest = uni().

by avoiding the store-and-forward overhead in light traffic. Non-interfering bidirectional communications can be provided by the underlying architecture for any of the transport schemes that we are considering. Avoiding store-and-forward overhead requires a transport strategy based upon some variant of circuit switching, or datagram with cut-through.

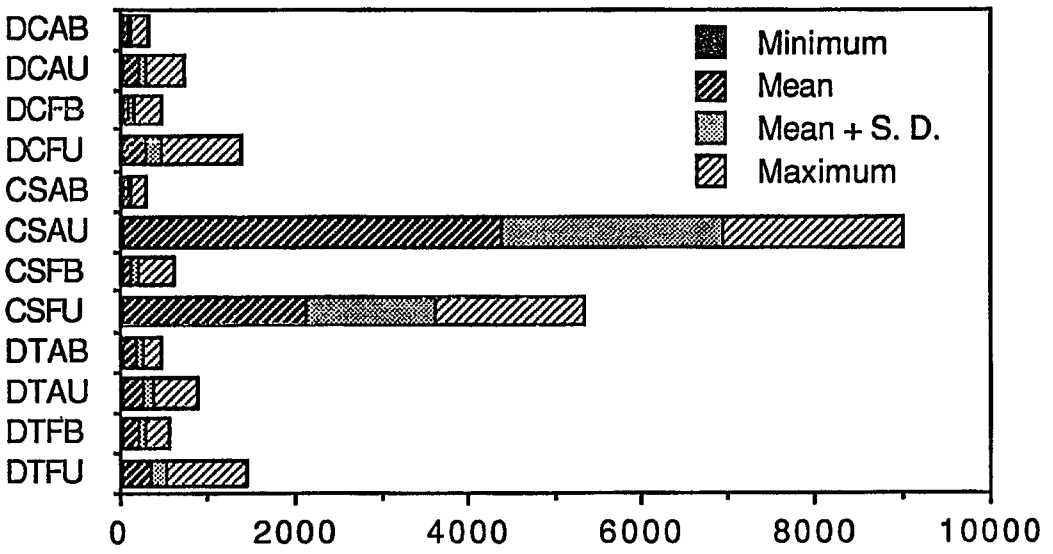


Figure 4.27. Message Time: Freq = 80, Len = nor(16,8), Dest = uni().

However, the buffering requirements of cut-through conflict with the desire to support arbitrary length messages. Packet switching, in which individual messages are broken into fixed sized packets, provides a reasonable solution to this conflict by limiting the buffering requirements while maintaining the ability to handle arbitrarily large messages by using a similarly large number of packets. In order to preserve the ordering of packets within a message, all packets must follow the same route from source to destination. Alternatively, one could attach sequence numbers to each packet and reassemble them in the correct order at the destination. However, reassembly is a very expensive operation whose cost cannot be justified in this context. This is especially true when trying to allow the processing of message data to proceed concurrently with the arrival of later portions of the same message. The limitations placed on packet switching by the reassembly problem will be shown later to be surmountable. The task of coordinating the processing of message data with the arrival of the later portions of the same message is made easier when messages arrive in a contiguous non-interleaved fashion on a particular channel. Message transport mechanisms that are based on circuit switching deliver their messages in such a manner, packet based schemes do not. Packet based transport schemes may

still accommodate the overlapped processing and reception of message data; however, doing so requires additional architectural complexity. This additional complexity can be seen in Sec. 5.4.6.

We further investigate two schemes. The first is wormhole routing, a variant of fixed route circuit switching. Wormhole transports generally compare favorably with other existing schemes. However, they have two drawbacks. One is their relatively poor performance for small messages in heavy traffic. We expect that for several algorithms a large proportion of messages under conditions of heavy traffic will be of relatively short length. Certainly, most request messages in algorithms that use a request/response communication paradigm will be short. Also, there are many amorphously structured algorithms, such as chess [FMO*87] or programs incorporating branch-and-bound techniques [AM88,AC87,Qui87,WLY85], that would like to make a random accesses to small amounts of global data. The second drawback is that the presence of a long message can effectively delay the delivery of other messages.

A simple example of the second drawback is illustrated in Fig. 4.28. In this example, assume that with the two messages starting at about the same time, the circuit for the larger message gets established first, forcing the shorter message to wait until the larger message completes for its circuit to be established. At a high level, we can view the message transmission time of a circuit or cut-through based transport scheme as being proportional to $M + h\beta$, where M is the length of the message, h is the number of hops from source to destination, and β reflects the speed with which the communication links can be acquired. The magnitude of β is dependent upon the traffic load (i.e., number and size of active messages), and the availability of communication resources—in this case, links. Alternatively, if we could break messages into packets, then interleave the packets of the two messages, transmission time would be proportional to $\frac{M}{p}(p + h\alpha)$. Where α reflects the apparently reduced bandwidth of the link as viewed from the message level due to the multiplexing and additional overhead of packet headers, and p is the size of the packets. This, of course, can be rewritten as $M + \frac{M}{p}h\alpha$. When we take into account

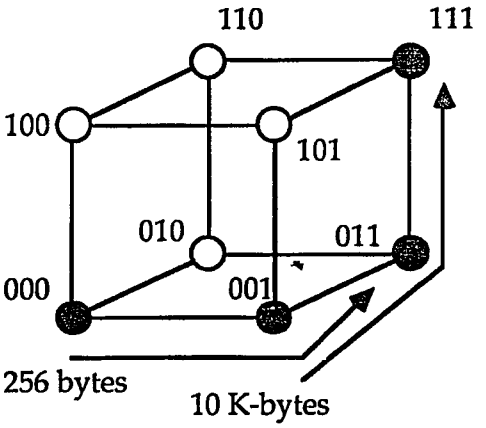


Figure 4.28. Wormhole Routing Blockage.

the fact that the message packets are pipelined through the network, the multiplicative effect of the number of packets ($\frac{M}{p}$) on $h\alpha$ is reduced to one. Thus, the better performing transport scheme will depend on which of α or β is lower.

Returning to our simple example, we see that with wormhole routing the transmission times for the long (t_l) and short (t_s) messages are:

$$t_l = M_l + h$$

$$t_s = M_s + ht_l$$

And with a packet based scheme:

$$t_l = M_l + h\alpha_l \qquad \text{where, } \alpha_l \approx 1$$

$$t_s = M_s + h\alpha_s \qquad \text{where, } \alpha_s \approx 2$$

The effect of α_l is negligible due to the size of M_l relative to M_s, and α_s is slightly larger than 2 to account for the additional overhead of packet headers. In general, the outlook is not quite this bleak for wormhole transports. However, it is important to note that while α tends to be, at worst, linearly proportional to the average number of packets

vying for a link, β suffers additionally from the effect of link starvation. This occurs as a result of messages being blocked after having acquired, possibly several, links. Thus, in addition to waiting on a given link, the waiters also hold out of service all of the links that they have already acquired. Similar effects with packet buffers, in the case of packet based transports can be easily avoided with a modest number of buffers. These effects can be seen in the results reported in Sec. 5.7.

The second scheme that we consider employs a packet based transport mechanism that allows packets from different messages to be interleaved. It uses a combination of fixed and adaptive routing in which the first packet may adaptively select its first routing step in any direction that will take closer to its destination. All subsequent packets from the message are routed in the same initial direction. All routing decisions, other than the first, are determined by a fixed routing algorithm that depends only on the current and destination nodes. Thus, packets from the same message will always arrive in order at their destination. There are two primary costs associated with this scheme: 1) each packet must now carry its source and destination node numbers, thus lowering the effective bandwidth for message data as discussed above; and 2) the architectural scheme for detecting if the node CPU requests part of a message before it arrives is more complicated than the comparable implementation for wormhole routing. Implementation restrictions limit the number of concurrently arriving messages that a communication processor can check. These checks are made by the hardware by employing multiple pairs of comparators to monitor the address bus for addresses that fall between the address of the most recent byte received and the address of the last byte expected for messages that are actively being received. Since wormhole messages do not interleave on individual channels, the maximum number of concurrently arriving wormhole messages is at most n, the degree of the cube. Thus, potential overruns can can be detected for all arriving wormhole messages with n pairs of comparators. For packet routing, which can concurrently receive a far greater number of packets, we have to maintain a *cache* of addresses that track the progress of the first m messages that arrive concurrently. This

cache simultaneously feeds m pairs of comparators that check these addresses against the node CPU memory requests that appear on the node address bus. Messages that have their arrival tracked by entries in the *comparator cache* (which is referred to as the arrival tracking unit in the following chapter) have their completion flags marked to indicate that message arrival has begun and that monitoring for overruns will be performed. Upon completion of message arrival their completion flags are marked to indicate reception-completed. Any message that begins to arrive while the comparator cache is full will have its completion flag marked only upon completion of message arrival, thus indicating that overrun checking is not being performed for this message. While not all arriving messages may be able to be processed concurrently with their reception, this should not be a problem in practice since the node CPU will still have several arriving messages with which to overlap computations.

The responsibilities of the communication processor include the execution of all communication instructions that are encountered in the node CPUs instruction stream. These instructions were described in Sec. 4.1. The only direct communication from the communication processor back to the the node CPU occurs via the interrupt mechanism. The communication processor can request a node CPU interrupt for several reasons, including:

- errors encountered during the execution of a communication instruction (e.g., invalid message buffer address)
- node CPU memory requests for message data that has not yet arrived and
- to indicate that a message of type system has been delivered.

Interrupts are not generated upon the delivery of messages of type user. It is assumed that programs will poll for the delivery status of such messages. In a multitasking environment it may be desirable to eliminate the need for repeated polling of message delivery status. This can be accomplished by mapping the user message types into some subset of the system type range. In order to transfer message data between user specified buffers and the communication network the communication processor needs to be able access all of

the user accessible memory. The communication processor also reserves a portion of the node memory for its own internal use.

4.6 Chapter Highlights

We have identified the relative strengths and weaknesses of several design alternatives that may be used in the implementation of various routing schemes in this chapter. We also developed a set of communications instructions and semantics based upon a discussion of general communication environment issues. These instructions and semantics were then combined with the results of our routing design comparisons to outline a new communication processor design.

The key features of our new design include:

- the handling of all communication system interrupts by the communication processor.
- communication instructions that are directly executable by user processes
- architectural support for broadcasts
- true bidirectional communications
- packet based message transport service
- partially adaptive message routing
- the ability to overlap the processing of message data with its arrival.

The first three items reduce the communication processing demands that are made on the main node processor. The next three items will reduce message latency in environments with with large messages or bursty traffic patterns. Our packet based scheme also avoids store-and-forward overheads by *cutting-through* packets that are being forwarded and by storing any packet data that is accumulating at an intermediate node in a high-speed

buffer memory. The final item allows many programs to take advantage of the reduction in message latency times even when total message times may have increased. This is possible because there is typically both a large excess in communication system bandwidth and a very small number of accesses to message data that do not occur in order. The greatest departure from existing designs is provided by the last three items. The design for our new *quasi-adaptive packet routing* communication system will be developed further in the following chapter.

CHAPTER 5

COMMUNICATIONS SYSTEM ARCHITECTURE

In this chapter, we further develop and evaluate the performance of the packet based message transport that we introduced in Sec. 4.5. We begin by presenting an architectural overview of the communication processor. This is followed by a discussion of the analysis that lead to the development of our quasi-adaptive routing scheme and an examination of the deadlock issue that it introduced. Functional descriptions of the major components of the communications processor are then given. Finally, we discuss the results of our simulation based performance analysis.

5.1 Architectural Overview

Several different implementations of a communications processor which meets the criteria discussed in Sec. 4.5 are possible. We discuss one such implementation in this chapter. Our intent is to show that it is feasible to build the communication processor architecture that we describe in a manner that is consistent with our simulation model. This is done by offering a functional description for the major components of the processor and analyzing the critical path time. We attempt to minimize the number of connections required across major functional boundaries in the architecture that we specify. This is done so that the implementation may be divided across multiple chips. We have also attempted to minimize the number of functional units that are on the critical timing path of packets that are passing by a node. We do not claim that the specific design that

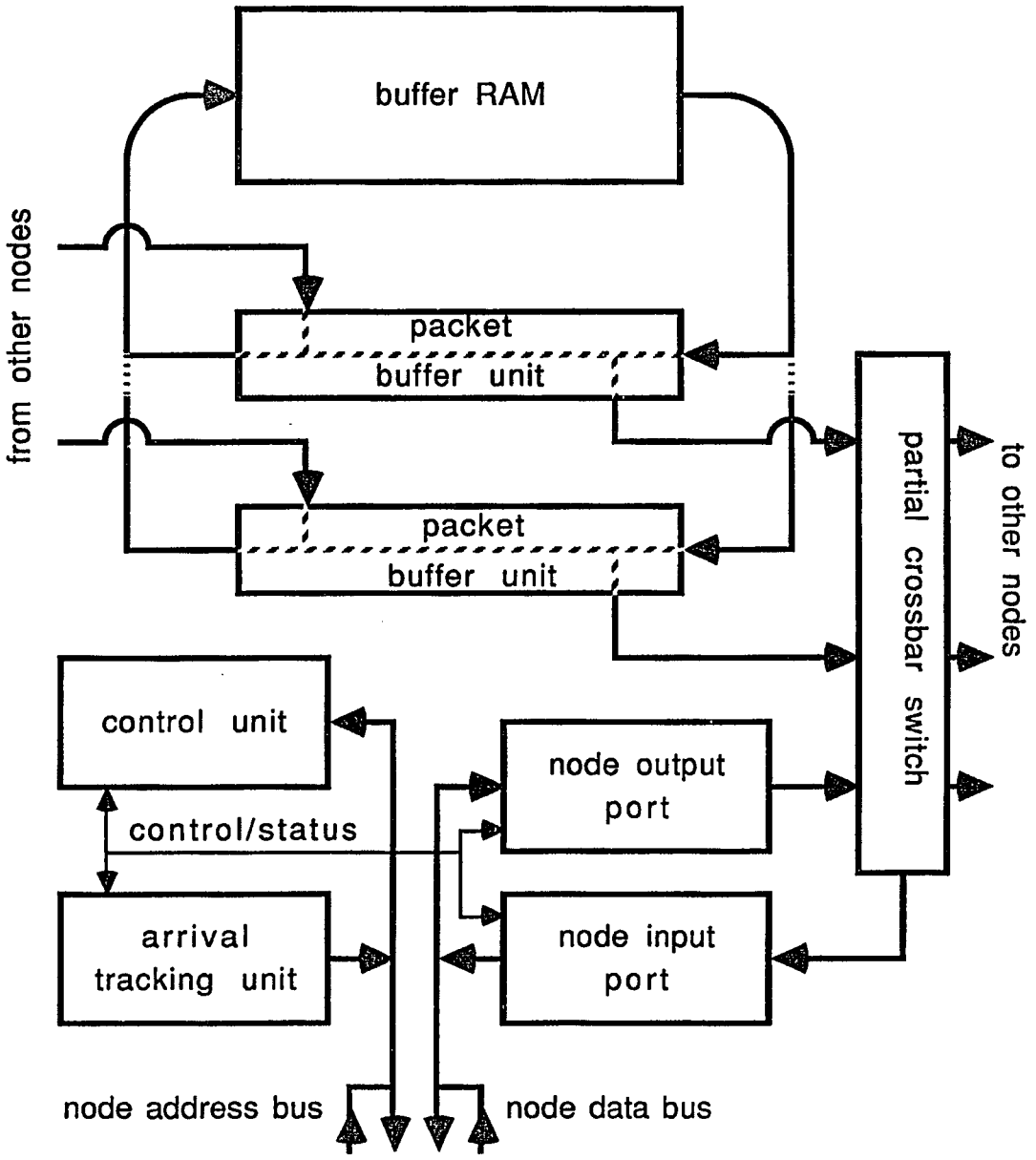


Figure 5.1. Major Functional Units of Communication Processor.

we offer is optimal in any sense. A thorough analysis that carefully considers all major design alternatives is considered to lie beyond the scope of this thesis.

Our implementation consists of eight major functional units. These units are listed below. A block diagram of the communication processor that shows seven of the eight units is given in Fig. 5.1.

The buffer RAM provides space efficient storage for packets passing through the communication processor. The RAM is dual ported, with reads and writes using separate ports, to reduce contention. Space in the buffer RAM is divided evenly among the packet buffer units described below.

Packet buffer units serve five functions: 1) they handle the reception of packets from preceding nodes; 2) they store in their *fast access memory* the first few bytes of packets that pass through the communication processor; 3) they hold the address of the buffer RAM location at which the remainder of the packet is stored; 4) they arbitrate for packet buffer space on the next node on behalf of each of the packets they are holding; and 5) once the arbitration is successful they oversee the delivery of the packets. Packets from the same message are always forwarded in order. However, packets from different messages may be interleaved to allow them to progress at independent rates that are governed only by the available resources on their specific paths. When resources are not available, a flow control mechanism (described below) is used to temporarily halt the incoming packets until more resources become available. The packet buffer units are active entities that each have there own autonomous controllers to sequence their operations. There is one packet buffer unit on each node for each dimension in the cube.

The crossbar switch provides connections from any of the packet buffer units and the node output port to the node input port and any of the output links that lead to a neighboring node. The $(n+1)\times(n+1)$ crossbar allows multiple simultaneous connections to exist. Like the buffer RAM, the crossbar switch is a passive entity.

The node output port receives packets that are generated on the local node and introduces them into the network through the crossbar switch. It is an active entity that performs several of the functions of the packet buffer unit: providing partial packet storage, arbitrating for further resources on behalf of its packets and forwarding the packets when the resources have been acquired. The number of partial packets that may be stored in the node output port is equal to the degree of the cube. Storage for a partial packet is referred to as a buffer slot. There is one buffer slot allocated for each link to

a neighboring node. The *main* control unit (described shortly) is responsible for moving data from the node memory to buffers within the node output port.

The node input port is a FIFO buffer that holds the packets that have arrived at their destination node until the main control unit can store them at the appropriate location in node memory. The incoming data that is stored in the FIFO buffer arrives from a packet buffer unit via the crossbar switch. The node input port has a simple controller to handle write operations on the FIFO. This enables it to emulate the flow control actions of a receiving packet buffer unit. Thus, the node input port will look just like another downstream packet buffer unit to the packet buffer unit that is forwarding the packet.

The arrival tracking unit tracks the progress of arriving messages. It maintains the node memory addresses of the current and final byte of the arriving messages. The node CPU's memory requests are monitored and compared against these address pairs. If the CPU requests part of a message that has not yet arrived, this unit requests a CPU interrupt. The current node memory address of the arriving message indicates the location at which the control unit should place the arriving packet that is currently being buffered in the node input port. This unit can request a CPU interrupt, but has no controller of its own.

The (main) control unit continually checks the node output and input ports to see if any data needs to be transferred between either port and the node memory. It also executes all of the communication instructions that the node CPU encounters and maintains data structures, both on-chip and in node memory, that track the state of communications. The communication instructions were described in Sec. 4.1.1 and the communication data structures are discussed in Sec 5.1.1.

The flow control unit is the only major unit not shown in Fig. 5.1. The flow control unit is comprised of a sending section and a receiving section that reside on separate nodes at opposite ends of each communication link. The purpose of the receiving section is to prevent a packet buffer unit or a node output port from overrunning the input capabilities of a neighboring downstream packet buffer unit. The receiving section may also be used

by a receiving packet buffer unit to refuse a *contrary packet* from an upstream node output port if the packet may lead to deadlock—contrary packets are defined in Sec 5.3. The node input port emulates the actions of the receiving section so that it also does not receive more data than it can handle. The sending section of the flow control unit continually polls the status of its corresponding receiving section. When the receiving end is ready to accept data, the sending section can relay this information to a unit that has data to send. The specific actions of this unit and all of the others described above are discussed separately in Sec. 5.4.

The three units located toward the top of the illustration in Fig. 5.1 comprise the packet switching section of the chip. Packets that pass by a node are handled exclusively by these three units. After arriving at a packet buffer unit, a small portion of the packet is stored directly at the buffer unit. This allows the packet to be moved rapidly to the next node if the resources are immediately available. The remainder of the packet is stored in the buffer RAM. As resources at the next node become available, the packet buffer unit forwards packets through the crossbar switch to the packet buffer unit at the next node. Packets arriving on their final node are handled in a similar manner. The chief exception is that they are passed via the crossbar switch to the node input port instead of being sent to a packet unit on a neighboring node. Packets on their originating node are primarily handled by the interface section of the communication processor which will be discussed below. For broadcast packets, which are denoted by having same value for both their source and destination fields, the packet buffer units must forward a copy of the packet to all higher order units (the specific routing requirements are specified in Sec. 5.4). In such a case, the packet buffer unit will send a copy of the packet to each higher order unit in order, holding the packet in its buffer until all copies have been sent.

The four remaining units that are illustrated in Fig. 5.1 comprise the node interface section of the communication processor. All messages that originate on a given node pass from the node memory, through the node output port and crossbar switch, to a packet buffer unit on an adjacent node. Conversely, all messages that are destined for a given

node are passed from the packet buffer unit on that node, through the crossbar switch and node input port, to the node memory. The node input and output ports each have their own autonomous controller to handle their interactions with the packet switching section of the communication processor. The actions of the node interface side of both of these ports, however, are directed by the main control unit. FIFO buffers separate the two sides of these units. The actions of the arrival tracking unit are directed by the control unit. A description of the communication system queues that are maintained by the main control unit and an overview of the functions performed by the node interface section of the communication chip are given below.

5.1.1 Communication System Queues

The control unit of the communication processor maintains five different groups of queues in node memory to support communication activities. The message header blocks that were described in Sec. 4.1.1 are manipulated among four of these five queues. The message header blocks serve as repositories of information about messages when the messages are located in node memory. Unlike the message data, the header blocks remain on the same node. Users may allocate memory for header blocks as needed or a pool of header blocks may be provided by the operating system. When messages are attached to the header block, or while the header blocks are being manipulated among the communication queues, they contain information about their current or expected messages. For the remainder of their existence, they convey no special meaning and may be treated like any other allocated data object.

The first group of queues are the message send queues. These queues are used to hold newly created messages until they can be introduced into the communication network. There is one queue for each buffer slot in the node output buffer. Thus, the number of queues is equal to the degree of the cube. A table of pointers is maintained to point to the message header block of the first message in each queue. This table is indexed by the number of the node output buffer slot to which the queue is assigned. The message

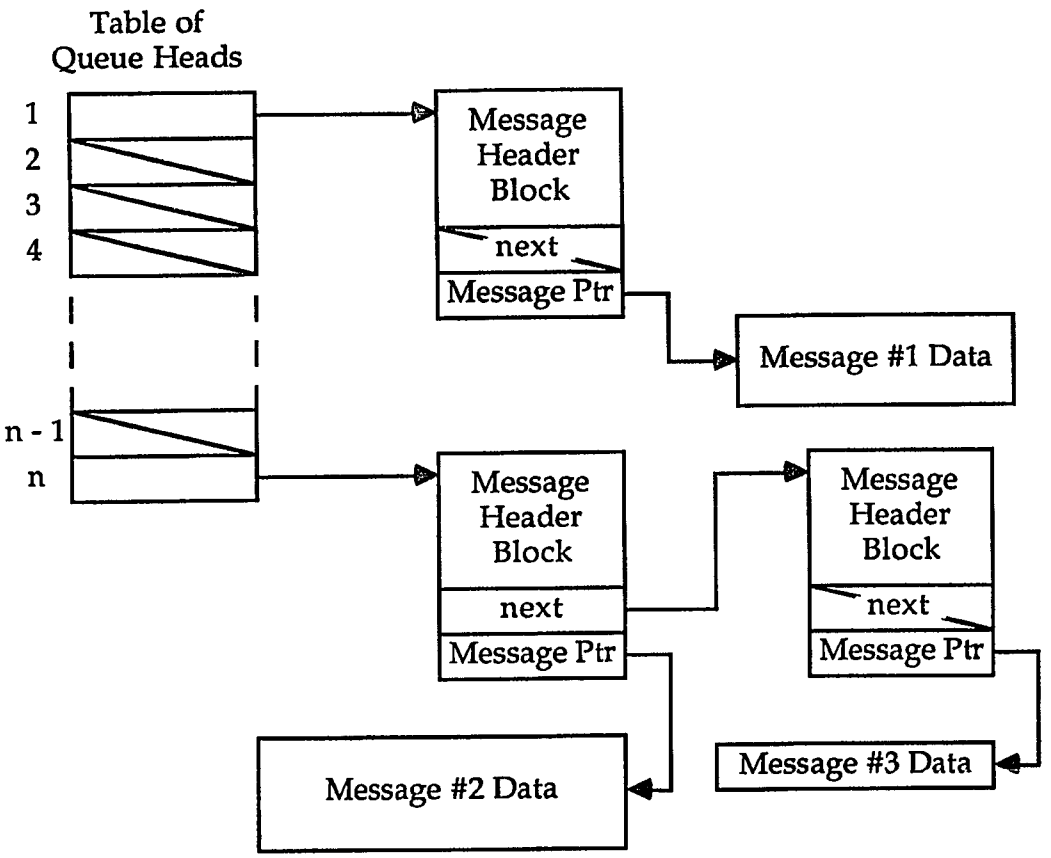


Figure 5.2. Message Send Queues.

send queues are illustrated in Fig. 5.2

The second group of queues are for the header blocks of *expected* messages that have not yet begun to arrive. Queues in this group will be referred to as expected message queues. An *expected* message is a message for which a receive instruction has been executed prior to the beginning of its arrival. Once such a message has begun to arrive it may still be referred to as an expected message, but it will no longer reside in an *expected message* queue. Several expected message queues are maintained. The pointers to the first message header block in each queue are held in a table. This table is indexed by a hash value that is a function of both the source and type of the expected message. Making the hash value a function of both the message source and type minimizes the chance of a potential search problem in multiuser environments. In a multiuser environment there is a

reasonable chance that two independent programs executing on a given node would each be receiving messages at about the same time from two different processes that also share a common node. This is particularly possible if the two nodes are near neighbors, which in many algorithms have a higher probability of communicating than two distant nodes. If the hash value were a function of only the message source, the communication processor may frequently search through message header blocks queued for messages expected by the first process while looking for header blocks for messages expected by the second process. A hash function that is based on a combination of message source and type would lessen this chance of conflict by spreading such messages among different queues. The handling of requests for messages from *ANY* source represents an exceptional case that is not accommodated by the scheme as described above. This anomaly is rectified by allocating a special table entry for the header blocks of such requests. When a new message arrives, the communication processor first searches the special queue for requests for *ANY* message. If none are found it then looks in the queue corresponding to the hash value generated from the source and type of the arriving message. The communication processor can maintain an on-chip counter to determine when the check of the *ANY* source queue may be omitted. If a header block that matches arriving message cannot be located in either queue, the arriving message is said to be *unexpected.*

The third group of queues form the message arrival table. The on-chip message arrival unit can store only a limited amount of information for tracking the progress of arriving messages. Therefore, it functions as a cache of the *most likely to be needed* subset of information that is available in the node memory message arrival table. The on-chip cache never writes back to the table in node memory. Instead, the information on an arriving message *sticks* in the on-chip cache until the message arrival is complete; at which time information about the message is purged from both the table in node memory as well as the on-chip cache. The table that is maintained in node memory consists of pointers that point to header blocks of messages that are in the process of arriving. This table of pointers is also indexed by a hash value based on both the source node and type

of the arriving message. It is possible for more than one message to be simultaneously arriving from the same source node. This is a side effect of the multiple routes that are possible with our quasi-adaptive routing scheme. Since at most one message will leaving a source node in each direction at any point in time, the arriving packets of multiple messages from the same source can be assigned to the correct message by inspecting the direction of the first routing step. This information is encoded within the unused bits at the high end of both the source and destination fields by the communication processor at the source node.

The fourth set of queues are for user messages that have completely arrived, but have not yet been claimed by a message receive instruction. This queue set is also implemented as a table of pointers to queues of message header blocks. The same hash function described above is used to index this table as well. This group of queues would also be susceptible to searching problems similar to those described for the expected message queue if its hash function were not also a function of both the message source and type.

The size of the pointer tables for the preceding three groups of queues may be modified by changing the specific hashing function that is used. Implementing the hashing function in a manner that allows the range of the hash values to be programmable will be useful to accommodate systems of varying memory space. If there is concern about the possibility of careless users flooding the queues with requests or unclaimed messages to the point where systems communications may be affected, the hash function could be chosen to map messages in the type range reserved for the system into a separate section of the pointer tables.

The final queue that is maintained to support communications is the pool of available system message buffers. This is the pool from which buffers for unexpected messages are allocated. Application programs that receive buffers from this pool are expected to return them by using the buffer release instruction when they are no longer needed. The locations of the queue head of the system message buffer queue and of the pointer tables

for the preceding four groups of queues are stored within the communication processor.

5.1.2 Message Initiation and Reception

Message transmissions are initiated by coprocessor instructions that are encountered by the node CPU. These instructions, either *send, broadcast,* or *near-neighbor broadcast,* are executed by the control unit of the communication processor. The control unit will check the status of the node output port. If the desired buffer slot is available, the required data from the message header block will be marshalled into the buffer slot in the output port. If the message is to be broadcast, the control unit will write the message data to each required buffer slot as space becomes available. If space is not available in the node output port, the header block will be added to one of the send queues in node memory for later processing. A separate queue is maintained for each output buffer slot. The header block will be assigned to the shortest queue that corresponds to a buffer slot that forwards packets in a viable routing direction (i.e., a direction which takes the packet closer to its destination). Once the message progresses through the queue to the node output port, the control unit within the port begins to arbitrate for space in the packet buffer unit of the next node that the message will visit. Once this space is acquired, the node output buffer begins introducing the first packet into the network through the crossbar switch. The preceding two steps will be repeated for each packet of the message. Meanwhile, the main control unit, as one of its usual functions, is keeping the FIFO of the node output port full. After the node output port has introduced the final byte of a message to the network, the main control unit will check the appropriate queue in node memory to see if another message is waiting to be sent.

Messages that have arrived at the packet buffer unit on their destination nodes arbitrate for space in the node input port. When an arbitration request is granted, the control unit within the node input port releases its flow control block and begins to store the arriving packet in the FIFO at the node input port. As one of its usual functions, the main control unit continually checks the status of the node input port. When the main control unit

notices that a packet is present in the FIFO, it consults a table in the arrival tracking unit to determine the node memory location at which to store the arriving data. If this information is not available on-chip in the arrival tracking unit, the control unit will then consult its arrival table in node memory. If a message header block corresponding to the arriving message is found in the node memory arrival table and there is space available in the on-chip cache, the current and final message buffer addresses will loaded into the cache and the completion flag in the message header block can be set to the receive-in-progress value. The receive-in-progress indication can be set because the arrival tracking unit will now keep track of the arrival progress of this message until it has completely arrived. If no message header block which matches the arriving message can be found in the node memory table, the expected message table is searched next. Expected but not yet arriving messages will have had their message header blocks queued into the expected message queue by the control unit when their corresponding receive instructions were executed. Expected messages are not required to specify a message buffer. If they do not specify a message buffer the control unit will allocate the necessary space from the system buffer pool and write a pointer to this space into the message header block. If a matching header block is found in the expected message queue this header block is moved to the arrival table, and if space is available in the on-chip cache, the cache is loaded as described above. If a matching header block is still not found, the arriving packet must be unexpected. If this is the case, the control unit will allocate space for the message data and the message header block from the system buffer area in node memory, update the header block, queue the header block in the arrival table, and load the on-chip cache if space is available. Once the location of the message buffer for the arriving packet is determined, the control unit begins to store the packet in node memory. As the last packet byte is written to memory, the arrival table information is updated as described immediately below.

If the arriving packet is the first packet of the message, there are two possibilities. Either: 1) there is space in the on-chip tracking unit —in this case an entry in the on-chip

arrival tracking unit that contains the locations to which both the next arriving packet and the final byte of the arriving message will be written is created, additionally, the control unit writes the receive-in-progress indication to the completion flag of the message header block in node memory or 2) there is no space in the on-chip tracking unit—in this case the locations to which both the next arriving packet and the the final byte of the arriving message will be written are stored in the arrival table in node memory, and no changes are made to the completion flag. In both cases above, the location to which the final byte will be written is computed from the address of the message buffer and the message length field which is part of the first packet of the arriving message.

If the arriving packet is not the first packet of the message there are also two possibilities. Either: 1) the packet that just arrived belongs to a message that is being tracked by the on-chip arrival tracking unit—in this case the location to which the next arriving packet should be written is updated only in the on-chip arrival table or 2) the packet that just arrived belongs to a message that is not being tracked by the on-chip arrival tracking unit—in this case the location to which the next arriving packet should be written is updated in the node memory arrival table.

If the arriving packet is the last packet of the message, information about it is purged from the on-chip cache and the completion flag in the message header is set to arrival-completed. Additionally, one of the following two actions is taken. If the message header block was allocated from system space and the second queue field is null, the block is moved to the unclaimed message queue. If the message header block was allocated from system space and the second queue field is not null, the message information is copied to the header block that is attached to the second queue field. This case is described further in the next few paragraphs. If the message header block was allocated from user space, the block is pointed to by some user process and can be unlinked from the arrival table. In any of the above cases, if the message type lies in the range reserved for system messages a node CPU interrupt is raised.

Like all of the other communication instructions, the receive instruction is also exe-

cuted by the main control unit of the communication chip. There are two major classes of receives, those that specify the message buffer that is to be used, and those that do not specify a message buffer. The receive instruction can be executed before, during or after message arrival. The first action in the execution of the receive instruction is to check if the message has already arrived. This is done by searching the appropriate unclaimed message queue. If, for receive instructions where a message buffer has been specified, a matching message is found, the message header information and data are copied from the system buffer space to the user supplied header block and message buffer. The message header block and buffer that were allocated in system space are then returned to the system buffer pool. For receive instructions where a message buffer has not been specified, if a matching message is found the message header information and a pointer to the system allocated message buffer is copied from the system buffer space to the user supplied header block. In this case, it is the responsibility of the user to explicitly return the system allocated message buffer when it is no longer needed.

If the requested message is not located in the unclaimed message list, the arriving message queues in node memory are checked for any arriving, unexpected, messages that match the requested message. Unexpected messages can be identified in the arriving message queues because they are the only messages with header blocks that have been allocated from system space. If a matching header block is found here, the message header block that is specified as a parameter to the receive instruction can be attached to the header block that was found in the queue. This attachment can be made via the second queue link field. When the message arrival has completed, the information from the system allocated header block can be copied to the attached header block and one of the two following actions will be taken. Either: 1) the attached message header block has a pointer to a user supplied message buffer—in this case the data in the system allocated buffer and header block is copied to the user specified buffer and header block and the system allocated space is released or 2) the attached message header block does not have a pointer to a user supplied message buffer—in this case the update to the attached header

block includes receiving a pointer to the system allocated message buffer, the user then assumes the responsibility for later returning the system allocated message buffer via the buffer release instruction.

A requested message that is not located in either the unclaimed message queues nor the arriving message queues, has not yet begun to arrive. For such messages the header block that is specified as the parameter to the receive instruction is queued in the appropriate expected message queue. The addresses of the first and last byte of the expected message are entered into the two *extra data* fields of the message header block. If the header block contains a pointer to a user supplied message buffer, if the request is for a message of *ANY* type and if there is space available in the on-chip arrival table, the addresses of the first and last byte of the expected message are also entered into the on-chip table.

5.2 Sequential Bottleneck at Source

Message bottlenecks near busy source nodes can significantly limit the performance of the communication system. The chief challenge in avoiding this problem is to spread the communications load among the many available links while still avoiding deadlock. While techniques for reducing local congestion away from source nodes have been discussed [Val82], our quasi-adaptive routing scheme provides a unique solution to reducing congestion near a busy source.

Originally, we used entirely fixed routing for all packets. However, initial studies revealed that the performance advantage of the packet based transport for heavy message traffic was quickly lost as the average message size was increased. Closer inspection determined that messages were blocking in sequential order on their source nodes, just as they do for the wormhole transport. When average message times were computed from the time messages began to leave their source nodes the performance advantage returned. In fact, the relative advantage was even greater than we had previously noted because the same effect had been occurring with the shorter messages as well.

The primary factor that limits the speed with which messages can be moved off of their source node is that fixed routing schemes direct messages to half of the nodes in the cube out of a single channel, messages to half of the remaining nodes go out the next channel, etc. Under conditions of constant uniform traffic, of course, this scheme makes optimal use of the links by evenly distributing the message load from all nodes across all links. However, in reality we expect message traffic to be bursty and chaotic. This can lead to excessive link contention for the most popular link by messages that are attempting to leave their source nodes. As a simple example, with all other communications quiescent, a pair of simultaneously generated messages on any given node in a cube of dimension n will collide while attempting to acquire their respective initial links with a probability given by:

$$P_f = \left(\frac{2^n}{2^n - 1}\right)^2 \sum_{k=1}^{n} \frac{1}{2^{2k}} \tag{5.1}$$

The $\left(\frac{2^n}{2^n-1}\right)^2$ term accounts for the fact that a node will not send a message to itself. The term in the summation is derived from the $\frac{1}{2^k}$ chance, for each message, that the kth link is chosen.

Rather than interleaving the message with packets that have been queued for transmission, we have chosen to adaptively select the first routing step for the message based upon the shortest queue that takes the message in a direction closer to its destination. All packets of the message still follow the same route; however, the direction of the first step in the route is no longer fixed. In this case, with all the other communications quiescent, a pair of simultaneously generated messages on any given node in a cube of dimension n will collide while attempting to acquire their respective initial links with a probability given by:

$$P_a = \left(\frac{2^n}{2^n - 1}\right)^2 \sum_{k=1}^{n} \left(\frac{1}{2^k}\right)\left(\frac{1}{2^n - 1}\right) \tag{5.2}$$

In this case, there is a $\frac{1}{2^k}$ chance that the first message chooses the kth link, and a $\frac{1}{2^n-1}$ chance that the second message cannot use any of the remaining links. Comparing (5.1) and (5.2), we can show that $\lim_{n\to\infty} P_f = \frac{1}{3}$ and $\lim_{n\to\infty} P_a = 0$, also $P_f \gg P_a$ for all

$n \geq 2$, therefore we conclude that our adaptive packet transport scheme significantly reduces the problem of messages colliding prior to entering the communication network. We also investigated adaptively routed the message by choosing the first step at random. Both analytical and simulation analyses showed this scheme to be inferior to routing in the direction of the shortest queue.

5.3 Deadlock Avoidance

Fixed routing schemes that preclude the formation of cyclical routing dependencies, such as the *ecube* scheme are inherently deadlock free. However, the use of fixed routing specifies a single path between source and destination nodes. Our quasi-adaptive routing scheme allows any of d (where d is the distance between the source and destination) paths to be used. As a consequence of relaxing the fixed routing requirement, the possibility of deadlock is introduced. Fortunately, this possibility can be easily avoided in our quasi-adaptive scheme. Let us refer to packets that are taking their first routing step in a direction that differs from the direction that they would take in the fixed routing scheme as *contrary* packets. Once a contrary packet has left its source node it will be routed along the normal fixed routing path, thus we will no longer consider it to be contrary. The non-cyclical nature of the fixed routing scheme that is used for non-contrary packets guarantees that as long as at least one buffer slot in each packet buffer unit on every node is either free, or occupied with a *non-contrary* packet, all non-contrary packets will eventually progress to their destinations, leaving only contrary packets in the network. Given this restriction on the allocation of buffer slots, the only way that deadlock can occur is if contrary packets by themselves can form a routing cycle. However, this cannot occur. In order to form a cycle at least two packets must take their first routing step in each of the dimensions (or routing *directions*) represented in the cycle. In any cycle at least two packets will take their first routing step in the same direction that they would have under fixed routing and, hence, will not be contrary. Thus, the existence of routing cycles formed entirely by contrary packets is precluded and deadlock cannot occur.

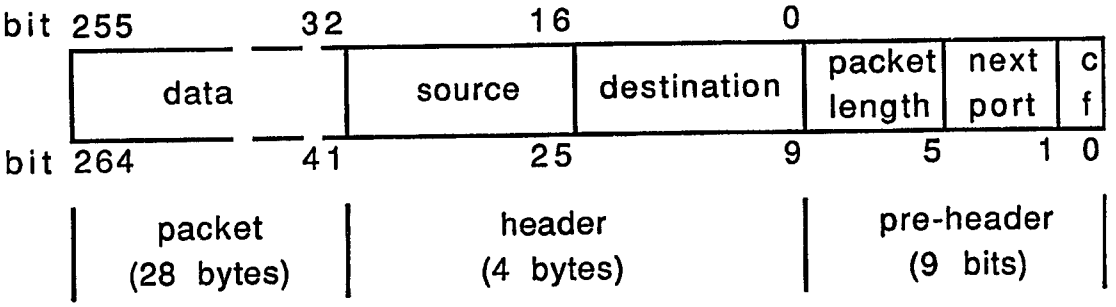


Figure 5.3. Message Packet Format.

5.4 Implementation of Functional Units

In this section we discuss further implementation details of the eight major functional units described in Sec. 5.1. The specific actions of each of these units are described and the message transport times through the critical path is discussed.

A central function of most of the functional units is the manipulation of message packets. The format of the message packets is illustrated in Fig. 5.3. Each message packet requires 4 bytes of header information that specifies the node numbers of both its source and destination. The destination node number is obtained from the message header block by the communication processor on the source node when the beginning of the message is first introduced into the network. The source value is known implicitly. The remaining nine bits provide information to the packet routing hardware. These bits are known as the packet preheader.

The first preheader bit indicates whether or not the packet is a contrary packet. Since packets may be contrary for at most one routing step, this bit is reset as it leaves its first packet buffer unit. The next 4 bits encode a selector to indicate the output port desired at the next node the packet will visit. As will be seen in the following sections, this field decreases the allocation time for routing resources. The final 4 bits indicate the length of the packet data and header (in 16 bit words), this allows the final packet of the message to be shorter than its earlier packets.

5.4.1 Buffer RAM

The buffer RAM is intended to provide temporary space efficient packet storage for all of the packet buffer units. The space within the RAM is evenly divided among the packet buffer units. The RAM is dual ported with output ports, from which data is only read, sharing simultaneous access with input ports, to which data is only written. The decision to use a RAM to hold the majority of the buffered packets instead of storing entire packets in the packet buffer unit was made for two primary reasons.

1. RAM cells used in a memory are typically much smaller than those used in registers [FMM87,HC85,mbo87].

2. Having a separate RAM allows more flexibility in distributing the implementation of the communication processor across multiple chips. In fact, high speed off-the-shelf static RAM chips could be used.

Based on our simulation studies, a reasonable RAM size for a degree 10 cube would be 4096 bytes. This would provide storage for 13 packets for each of the 10 packet buffer units. Dual ported CMOS static RAM cells that allow data to be written on only one of the RAM ports can be implemented with seven transistors [WE85]. For 4096 bytes of static RAM the storage cells alone would require about 230,000 transistors. This amount of information cannot be stored in registers within the packet buffer units with current technologies. However, storage for this amount of information could be provided by a pair of off-the-shelf $2K \times 8$ dual port static RAMs such as CY7C132-35C [IC 87].

Though it is part of the packet switching section of the communication chip, the buffer RAM does not lie directly on the critical timing path through the chip. This is because the first 25 bits of information that arrive at a packet buffer unit do not pass through the buffer RAM. The 9 bit packet preheader and the first 16 bits of the packet header use storage space within the packet buffer unit (described below) to bypass the buffer RAM. As long as the output side of the packet buffer unit can acquire the next 16 bits of the packet from the buffer RAM in less time than it takes it to send 25 bits

of information to the next node, the buffer RAM remains off of the critical timing path for the head of the packet. It is further required that each additional 16 bit word of the packet be accessed within 16 bit transmit times (which is the amount of information delivered by a previous buffer RAM access) to keep the buffer RAM out of the critical timing path for the transmission of the remainder of the packet. Assuming an order 10 cube and 35 ns memory access time we should be able to access memory at least every 350 ns. Thus, the buffer RAM will remain out of the critical path for all link bandwidths below 45 MHz.

5.4.2 Packet Buffer Units

A block diagram showing the data path components of a packet buffer unit is given in Fig. 5.4. The major component of a packet buffer unit is the shiftable content addressable memory (CAM). This memory is 29 bits wide: 1 bit to indicate if the packet is a contrary packet, 4 bits to select the next port, 4 bits to indicate the packet length (in 16 bit words), the first 16 bits of the packet header (the destination field), and 4 bits to specify the buffer RAM address at which the remaining bytes of the packet are stored. For this implementation we will assume the height of the CAM, that is, the number of packets that it and its corresponding space in the buffer RAM can hold, to be 13. In our simulations have we have evaluated CAM sizes that range from $\frac{2}{3}n$ to $2n$. The results were reasonable throughout the entire range. However, performance does increase slightly with increases in the CAM size. These results are explained further in Sec. 5.7.

The bit-wide value that results from the logical-and of all of the contrary flag bits is available to the controller at the input side of the packet buffer unit. This information allows the input controller to signal the receiving side of the flow control unit to refuse an incoming packet that would fill the CAM with contrary packets. The contents of the *next port* field of the CAM are used to associatively address the memory. It is useful to think of the CAM as having head and tail rows. Data in all rows of the memory may be shifted down so that all free memory rows accumulate at the tail end. When accessing memory

for either reads or writes, if multiple rows match the next port field, the row closest to the head is used. This allows multiple packets from the same message to move through the memory in a manner such that their order is preserved, while still allowing packets from different messages (and on different routes) to be accessed randomly. The ability to access packets randomly allows us to interleave the packets from different messages on the output link in a fair manner.

Each packet buffer unit contains two small, cooperating, controllers. One handles the loading of arriving packets into both the CAM and the buffer RAM. The other controller handles the forwarding of packets from the CAM and buffer RAM through the crossbar switch to the next packet buffer unit or the node input port. With the exception of the crossbar arbitration request register, all of the registers that the input controller affects are located above the CAM in the illustration given in Fig. 5.4. All of the registers that the output controller affects are located below the CAM in the illustration.

The operation of the input controller is given below.

1. Start: when free (CAM) row register is greater than 0, assert intention to use the CAM on the next cycle. Proceed.

2. Latch the RAM address offset into the RAM address register from the free row entry in the CAM. Zero the low order bits of the RAM address register. The high order bits of the RAM address register are wired to the buffer RAM offset for each specific packet buffer unit. The mid-order bits are latched into a static register from the CAM. The low order bits are implemented as a counter that is incremented after each write.

3. Release flow control block on input. Wait for start bits (10) to arrive on input line before proceeding.

4. Check the logical-and of the first arriving bit and the contrary flag line. If true, assert the flow control block and return to step 1. If not true, decrement the free row register and proceed.

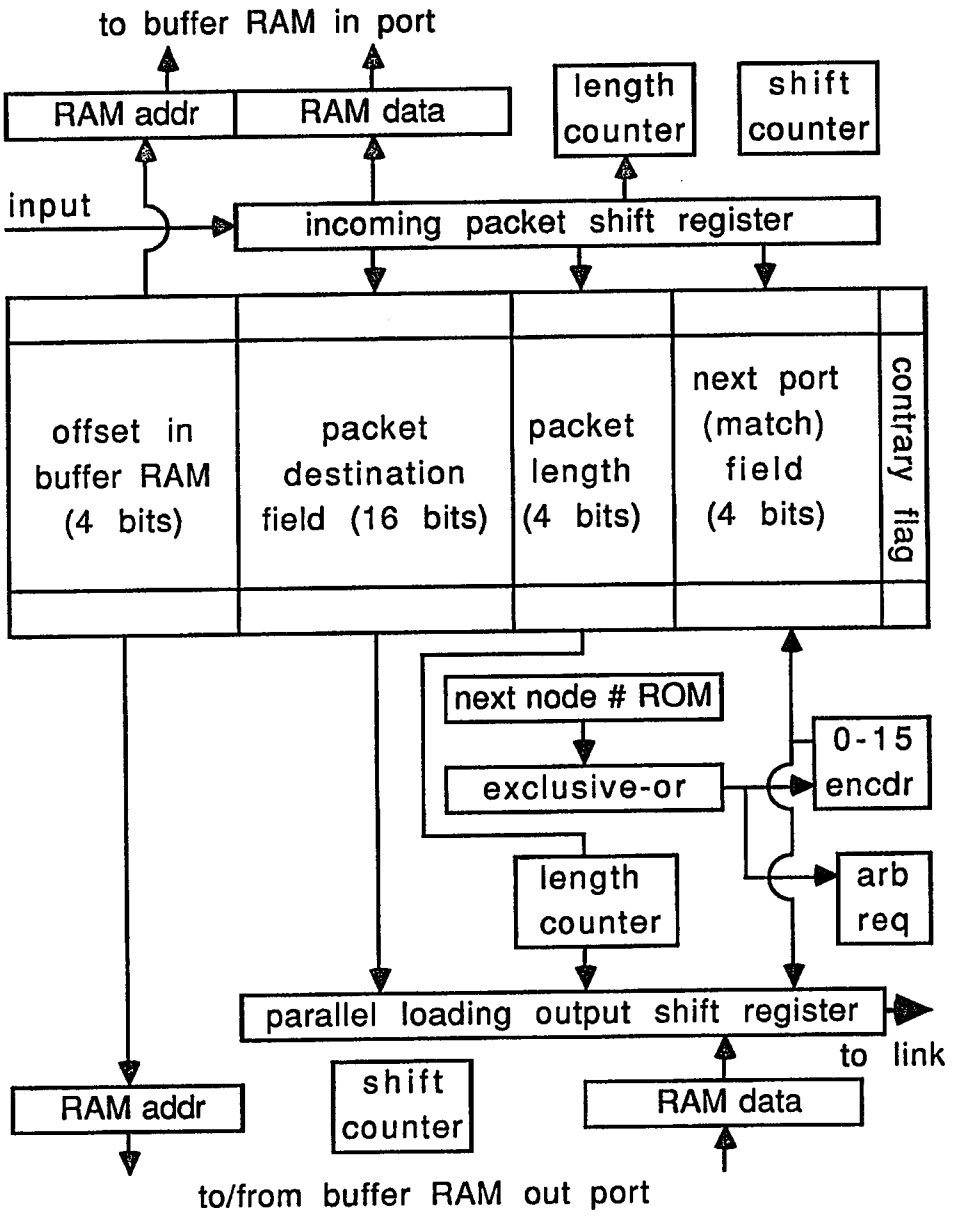


Figure 5.4. Packet Buffer Unit.

5. On the arrival of bit 8, latch the packet length field of the arriving packet into the length counter.

6. During the arrival of bit 23, assert intention to use the CAM on the next cycle.

7. On the arrival of bit 24, latch the first 24 bits of the incoming packet into the

first free CAM row, and set the bit that corresponds to the desired *next port* in the crossbar arbitration request register (the desired bit is determined by a 4 bit 0-to-15 decoder which is not illustrated).

8. After the arrival of 16 more bits, latch them into the RAM data register and check the length counter. If zero, assert the flow control block, write a non-valid address to the RAM address register, and return to step 1. If not zero, decrement the length counter and proceed.

9. While more bits are arriving, arbitrate for the buffer RAM input bus and write the buffer RAM data.

10. Increment the RAM address register, go to step 8.

Other parts of the input side of the packet buffer unit that are not shown include the free row register and the control unit. The input controller is given priority on all accesses to the CAM. Thus, on all accesses to the CAM, the output controller will enter a one cycle wait if the input controller has asserted its intention to use the CAM. The operation of the output controller is given below.

1. Start: when any of the crossbar arbitrations are successful, read the corresponding data from the CAM into the output shift register, RAM address register and length counter. The contrary bit is reset in the output shift register, since packets flowing out of a packet buffer unit are no longer contrary. The next port value is updated for the next routing step by setting it to a value determined from the next node number and the packet destination field. Proceed. The crossbar arbitration grant lines, which are not shown, feed into the 0-to-15 4 bit encoder. The result of this encoding is used to select the CAM entry.

2. Begin shifting, check if this packet is to be broadcast and not all of the broadcast copies have been sent. This is indicated by the source field of the packet being equal to the destination field and the next port field being less than the maximum

port number. If this is the case, increment the next port field and write it back to the CAM and set the appropriate bit in the crossbar arbitration register. If this is not the case, set a flag register to indicate that the CAM entry is to be freed as the last step of forwarding this packet. If the CAM entry were freed immediately, it could be reused and new packet data could be written to the corresponding buffer RAM locations before all of the current data is read.

3. Increment the length register to account for packet header bytes as explained for the input controller.

4. While bits are shifting out, check the length register and the input side RAM address register. If the length is not zero, the next action will depend on the result of comparing the RAM address registers on both the input and output sides. This check will prevent the output controller from trying to read RAM data before the input controller has written it. If the addresses are equal, the output side must wait until the input side finishes its current write. If the addresses are not equal, arbitrate for and read the first 16 bits of packet data from the buffer RAM into the RAM data register. When successful, increment the RAM address register and decrement the length register. If the length register is zero, go to step 6.

5. Wait until the low water mark is reached in the output shift register. Then, latch data from RAM data register into shift register. Go to step 4.

6. Check the two following independent conditions. If another packet with the same next port field is not in the CAM, reset the crossbar arbitration register. If the flag indicating that this CAM entry is to be freed is set, free the entry and reset the flag. Return to step 1. A CAM entry is freed by shifting all of the entries above it one step closer to the head. Simultaneously, the mid-order bits of the current RAM address register and a special next port value that indicates that this is a free entry are written to the tail of the CAM.

Parts of the output side of the packet buffer unit that are not shown include the control unit and logical-and gates for RAM address comparison and broadcast checking. A CMOS implementation of the CAM with 13 entries requires about 4000 transistors. The control units for both the input and output sides of the packet buffer unit can be constructed from PLAs. These PLAs and the remaining registers and logic require about 2500 additional transistors. Thus, the total device count for each packet buffer unit is approximately 6500 and about 65,000 devices would be required for an implementation of a 10-cube.

The critical path time is 31 cycles for the head of a packet. The majority of this time is consumed shifting the first 25 bits (plus two discarded start bits) into the incoming packet shift register. Without contention for resources, there is an additional cycle for loading the CAM, 2 cycles for arbitrating for the output link (see Sec. 5.4.3), and 1 cycle for unloading the CAM. Shifting of the output register can commence on the 32nd cycle. The critical path times for the remaining words of a packet are typically less since writes to the buffer RAM can commence after the arrival of each additional 16 bits.

5.4.3 Crossbar Switch

The routing scheme that we have proposed requires partial (upper triangular) crossbar connectivity between the input and output ports. That is, if we assume that: 1) our hypercube is connected in the standard manner such that all output ports O_i (where $1 \geq i \geq n$) on each node are connected to an input port I_i on some other node whose binary representation differs from the first node only in the ith bit position and 2) the node output port and node input port on each node are assigned the number 0 (i.e., $O_0 and I_0$ refer to the node output port and node input port, respectively), our routing scheme will always route a packet from input port I_i through an output port O_j such that $i < j$. The required crossbar connections are illustrated in Fig. 5.5. The inputs labelled 1 through n correspond to the output of the packet buffer units with the same numbers, and input 0 corresponds to the node output port. Similarly, the outputs labelled 1 through n

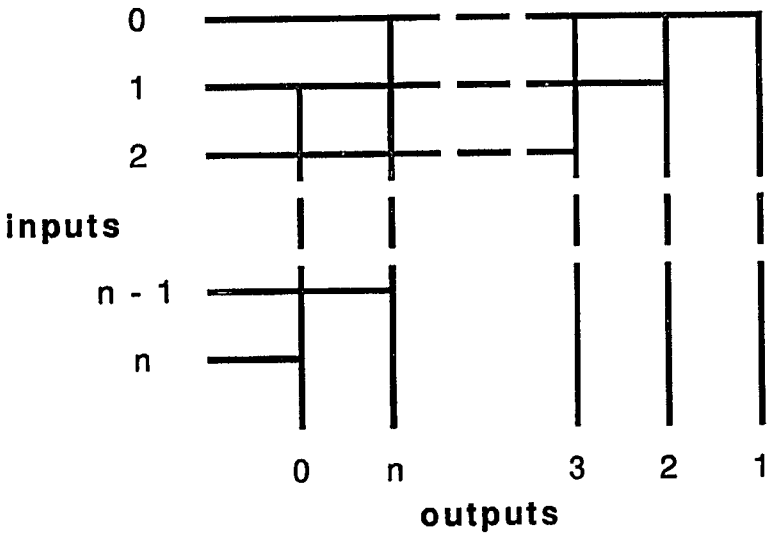


Figure 5.5. Connectivity of Input and Output Ports.

correspond to the input of the downstream packet buffer units with the same numbers, and output 0 corresponds to the node input port. A second similar crossbar is required to provide a path in the opposite direction for the flow control signals.

The units of the input side of the crossbar must arbitrate for the use of the outputs. This is accomplished by using a 1-of-$(n+1)$ arbiter for each of the $n+1$ crossbar outputs. Several designs for such arbiters exist (see [MHW87] for a brief overview or [PFL75] for more details). Generally, tree structured arbiters provide the most reasonable compromise between speed and fairness. A tree structured 1-of-8 arbiter, constructed from seven 1-of-2 arbiter modules is shown in Fig. 5.6. Each 1-of-2 arbiter module has two request lines and two corresponding grant lines. The grant line at the root of the arbitration tree is not asserted until the output flow control status is not blocked. As described in [PFL75], a 1-of-2 arbiter module may be constructed from 12 gates. The request signal experiences two gate delays for each arbiter module through which it passes, while the corresponding grant signal will experience one gate delay through each module. The total delay for a 1-of-M arbiter is $3\Delta \log_2 M$, where Δ denotes the nominal gate delay. For a cube of degree ten crossbar arbitration will require 12 gate delays. The critical path time stated in Sec. 5.4.2 allowed two cycles for arbitration. Thus, for clock periods in excess of

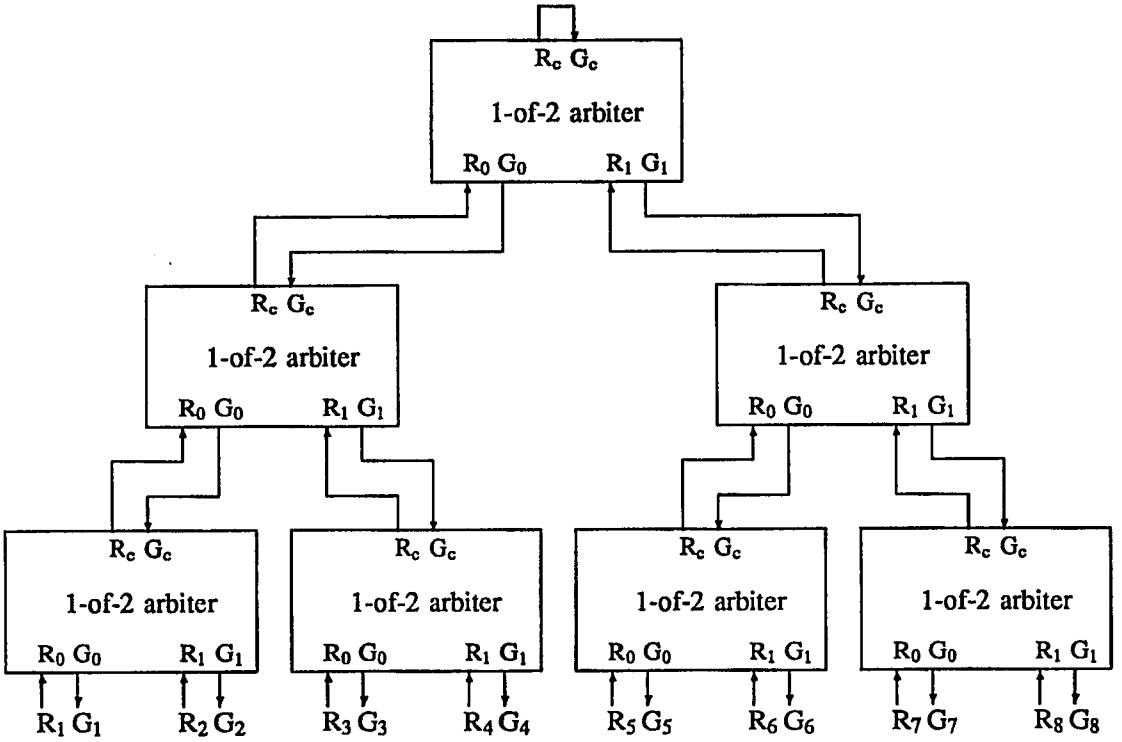


Figure 5.6. A 1-of-8 Arbiter Constructed from a Tree of 1-of-2 Arbiters.

6 gate delays the two cycles allowed for arbitration will be sufficient.

5.4.4 Node Output Port

The node output port introduces packets generated on the local node to the network through the crossbar switch. Its function is similar to that of the output side of the packet buffer units. An illustration of the node output buffer is provided in Fig. 5.7. All of the components illustrated are replicated for each of the n buffer slots. The main communication processor control unit is responsible for initiating the actions of the node output port. When a buffer slot for which a waiting message is available, the control unit fetches the 16 bit destination field of the packet from node memory. It then creates the 9 bit packet preheader (contrary flag, packet length, and next port number). This 25 bits of information is written to the prefetch buffer, 25 is written to the load register of the corresponding counter and the prefetch data available flag is set. The controller for

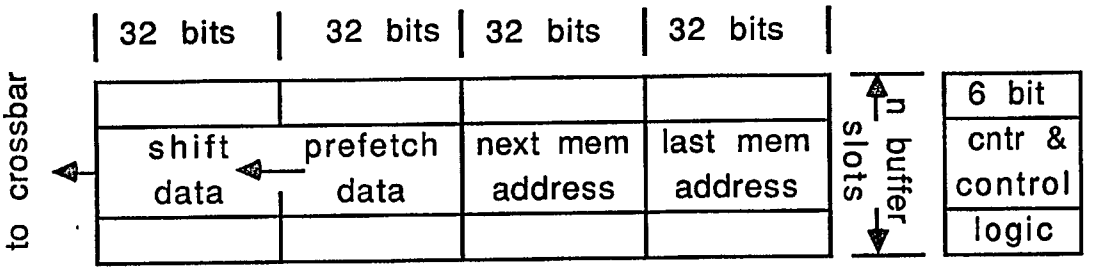


Figure 5.7. Node Output Port.

the buffer slot will notice the prefetch flag and begin its actions. Meanwhile, the main controller writes the node memory addresses of both the next and last words to fetch into their respective buffers.

The actions of the buffer slot controller are given below.

1. Start: When prefetch data is available, load the shift register with the data from the prefetch buffer, load the counter with the value in the counter load register, reset the prefetch refill request flag, and begin arbitration for the output link. Proceed.

2. When the output link is granted and data flow is enabled, start the shift register. Decrement the counter on each shift.

3. If the counter load register is not equal to 25 (i.e., if this is not the beginning of a packet), skip this step. After the 9 preheader bits have been sent, check the flow control signal. If the buffer slot controller is attempting to send a contrary packet which the downstream packet buffer unit is refusing, the flow control block that the downstream unit will have raised will be recognized during this bit. The details explaining why the flow control block will be recognized at this time are given in Sec. 5.4.8. If this is the case, return to step 1. Otherwise, proceed.

4. When the counter reaches 23, assert the prefetch refill request line. In response, the main control unit will reset the prefetch data available line and begin the operation of refilling the prefetch buffer if there is more message data to be sent.

5. When the counter reaches 0, return to step 1.

The main control unit loads subsequent data from the node location indicated by the next word field of the buffer slot into prefetch buffer 32 bits at a time. For each such load, the load register of the counter is set to 32 and the value of next word is incremented by four. An exception to this may occur for the last load of the last packet of a message which may be only 16 bits in length. In this case, the load register of the counter is set to 16. When the next word field is greater than the last word field the message has been completely loaded and the main control unit may begin to load the first packet of the next message from the send queue corresponding to the free buffer slot.

Broadcast packets are handled in the same manner as single destination messages by the buffer slot controllers. The only changes are in the manner in which the main controller loads packet data into the buffer slots. The main controller will fetch message data from main memory only once. This data is then written, as space becomes available, to the prefetch buffer of all the buffer slots to which the message is being broadcast. The next word and last word data that guides the fetching of message data rom node memory, needs to be recorded in only one location. Registers internal to the main control unit keep track of which buffer slots have and have not received the currently fetched word of packet data.

5.4.5 Node Input Port

The design of the node input port is illustrated in Fig. 5.8. The major component of the node input port is the FIFO. The 16 bit-wide FIFO receives data from the network via the crossbar switch. Data in the FIFO is removed from its bottom entry by the main control unit and written to the node memory. Writes to node memory are done 32 bits at a time whenever possible. Access to the FIFO counter is also shared by both the input and output sides of the unit. This counter provides the *FIFO address* to which the input shift register will write its next two bytes of packet data. After every two byte write, the counter is incremented by the node input port controller. A status flag that is read

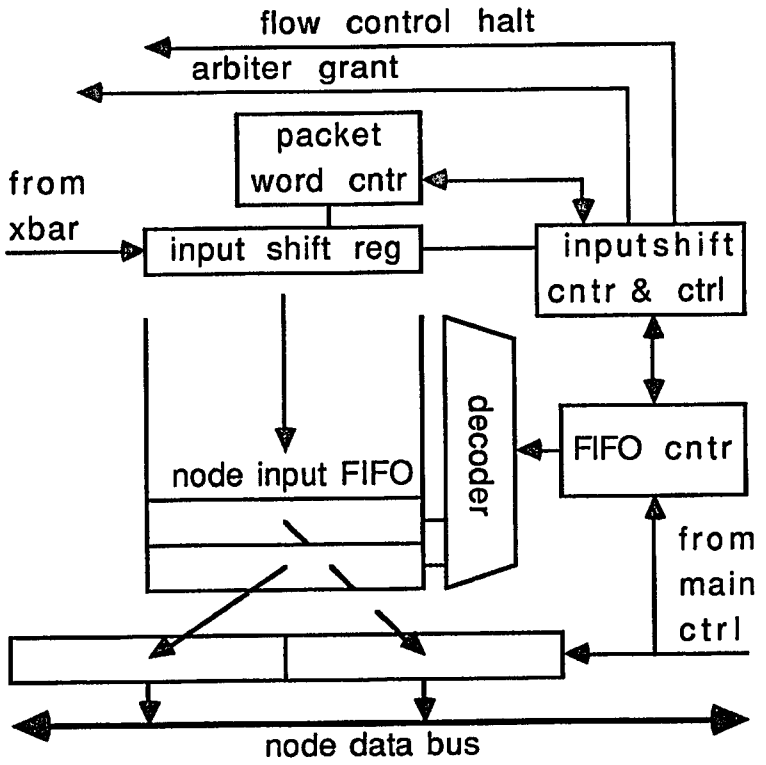


Figure 5.8. Node Input Port.

by the main controller is set whenever the counter is non-zero—thus, indicating that data is available in the FIFO. Upon reading each two bytes of data from the FIFO, the main controller will shift the FIFO and decrement the counter.

Whenever space is available in the FIFO, the input port controller will grant arbitration requests and enable the data flow signal. The specific actions of the input port controller are described below.

1. Start: Load the input shift counter with 26 and the packet word counter with 0.

2. When data flow is enabled, assert the grant line for the arbiter. The flow control line is wired to the FIFO counter, whenever the counter is below its maximum value, input data flow will be enabled.

3. After the start bits (10) arrive, begin to shift data into the input shift register, decrement the shift counter on each shift. Start bits will not arrive when data flow

is disabled.

4. On the arrival of the 8th bit (shift count equals 18), check the packet word counter. If it is zero, latch the 6th through 8th bits into the packet word counter.

5. On the arrival of the last bit of the packet word (shift count equals 0), write the packet to the FIFO, increment the FIFO counter and check the packet word counter. If it is zero, return to the start state. If it is not zero, decrement it, load the input shift counter with 18 and go to step 3.

5.4.6 Arrival Tracking Unit

The design of the arrival tracking unit is illustrated in Fig. 5.9. This unit serves two purposes. It watches the node address bus to ensure that the node CPU does not attempt to access message data before it has arrived. Additionally, it serves as an on-chip cache for the message arrival table. The chief component of the arrival tracking unit is a CAM with entries that specify the node memory addresses at which to store the next byte and the last byte of arriving messages. The main control unit consults and updates this memory when transferring arriving packets from the node input port to node memory.

Each active entry in the CAM corresponds to a message that is currently arriving at its destination node. Entries are uniquely identified by the message source and the direction of the first routing step; hence, this information is used to associatively address entries in the CAM. An additional one bit associative field is used to identify *expected* entries. There are two types of non-active entries: free entries, and entries for expected but not yet arriving messages. Expected entries are allowed only for requests for *ANY* message type. This is required because the associative field does not distinguish message types. When space in the CAM is needed for an arriving message, the first choice is to use a free entry. However, if no free entries exist, an entry for an expected message will be used if one is available. In such a case, the existing expected entry will not be saved because a valid copy of its information will exist in the arrival table located in

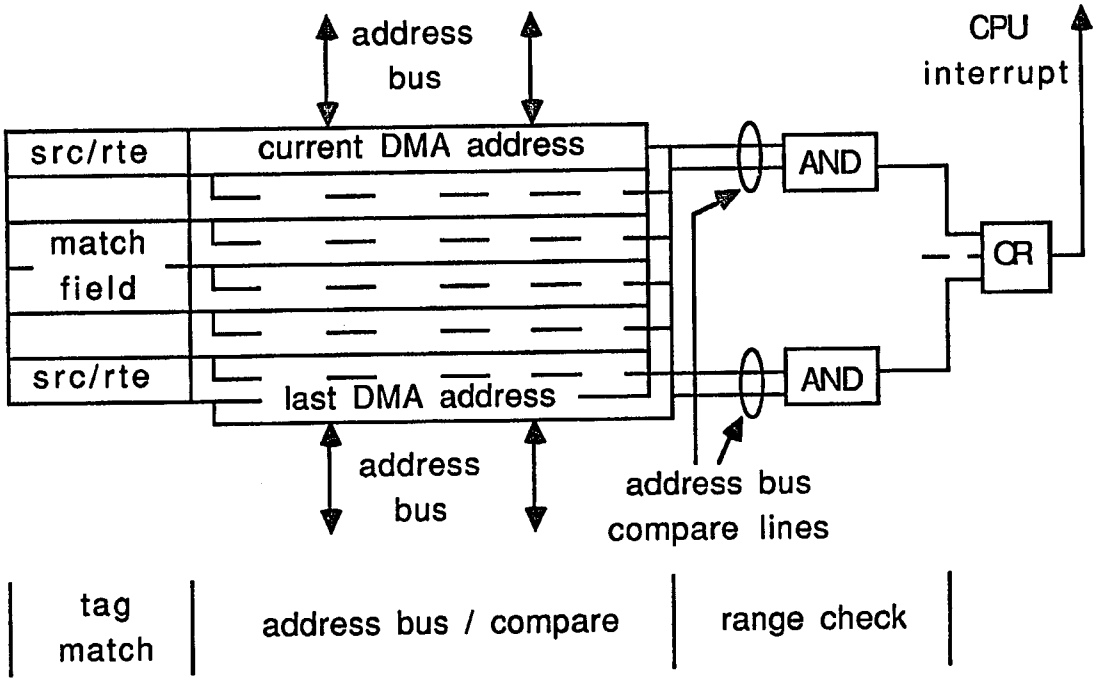


Figure 5.9. Arrival Tracking Unit.

node memory. When a message for an expected entry begins to arrive, the state of the entry is changed from expected to arriving. Arriving entries are not removed from the CAM until the arrival of their message has completed. Space may also be requested for an expected entry. In this case, the request will be granted only if a free entry exists.

A magnitude comparator exists for each node memory address in the CAM. If a memory request is made for a location that lies between the next byte and last byte values of any active entry in the CAM, a CPU interrupt will be generated.

5.4.7 Control Unit

The actions of the main control unit fall into two domains, those in response to communication instruction execution requests and those in response to status indications from any of the three functional units over which it may exercise control. The main control unit is responsible for the control of the arrival tracking unit, the input side of the node output port and the output side of the node input port. Though each of these

activities have already been discussed in detail, they are summarized below.

As a result of communication instruction execution requests, the control unit will manipulate communication queues and, possibly, initiate the actions of the node output port. In response to requests from the node output port the main control unit may fetch message data from node memory, manipulate the send queue and/or update the status or data in the node output port. Requests from the node input port may require the writing of message data to node memory, updating the status of the input port, manipulation of several communication queues and/or updating of the arrival tracking unit. Additionally, the main control unit may need to generate an interrupt for the node CPU. The only requests from the arrival tracking unit will be for the generation of an interrupt sequence for the node CPU, either for anomalies that it detects or on behalf of the arrival tracking unit.

A controller of such complexity may be implemented in several different forms ranging from PLAs to a microcode engine. In any case, it is reasonable to assume that such a controller could be implemented on a custom chip with enough room left to also accommodate a few of the smaller remaining functional units.

5.4.8 Flow Control Unit

The flow control unit consists of two separate sections: the receiving section and the sending section. These sections are located at opposite ends of the single wire links between nodes. Flow control information is passed on the same wire as the message data. For this purpose, a two bit synchronization pattern (a one followed by a zero) is interspersed with the data that travels on each link. This pattern is generated by the units that submit data to the crossbar—that is, the output side of the packet buffer units and the node output port. The pattern is inserted after the nine bit preheader and after every 16 bits for the remainder of the packet.

The simple circuit that comprises the flow control unit is shown in Fig. 5.10. The capacitor C1 represents the total capacitance of the interconnect between the two chips.

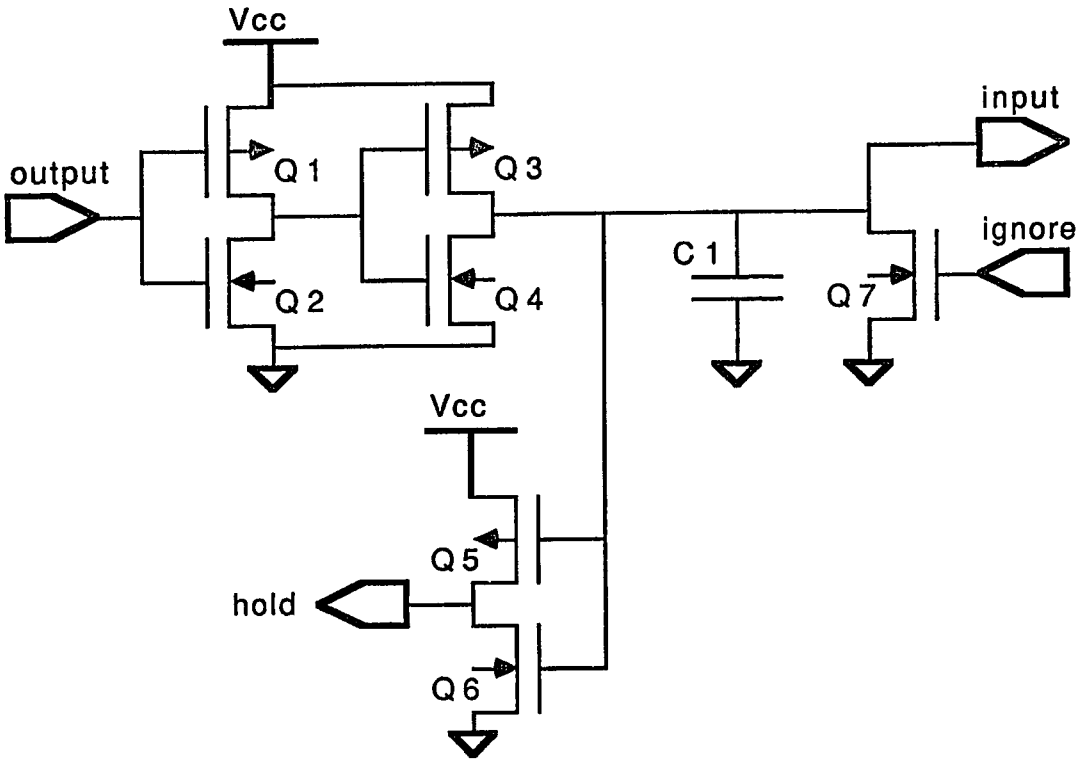


Figure 5.10. Flow Control Circuit.

This value represents the capacitance of one meter of backplane *wire* and the input and output capacitance of the chips at each end of it, as calculated from data in [Mot83a] and [Mot83b]. The single transistor Q7 functions as the receiving section of the flow control unit. It is turned on when the receiving unit wishes to block the reception of further information. As long as Q7 is off reception is enabled.

The output driver on the sending chip is modelled by Q1 through Q4. The flow control detection circuit consists of the inverter formed by Q5 and Q6. When the output of this inverter (labelled hold) is high during the one bit of the one-zero synchronization pattern it is an indication that the receiver is blocking the reception of further data. In this case, transmission of the one bit is repeated by the sending unit until the blocking condition is removed.

The sizes of transistors Q1 through Q7 are given in Table 5.1. The results of a spice simulation on the flow control circuit are given in Fig. 5.11. The spice simulation shows

transistor	width	length
Q1	100	5
Q2	40	5
Q3	250	5
Q4	100	5
Q5	25	5
Q6	20	5
Q7	250	5

Table 5.1. Transistor Sizes (in microns) for Flow Control Unit.

that the assertion of a flow control block signal can be detected by the sending side of the flow control unit within 15ns of it being raised by the receiving side of the flow control

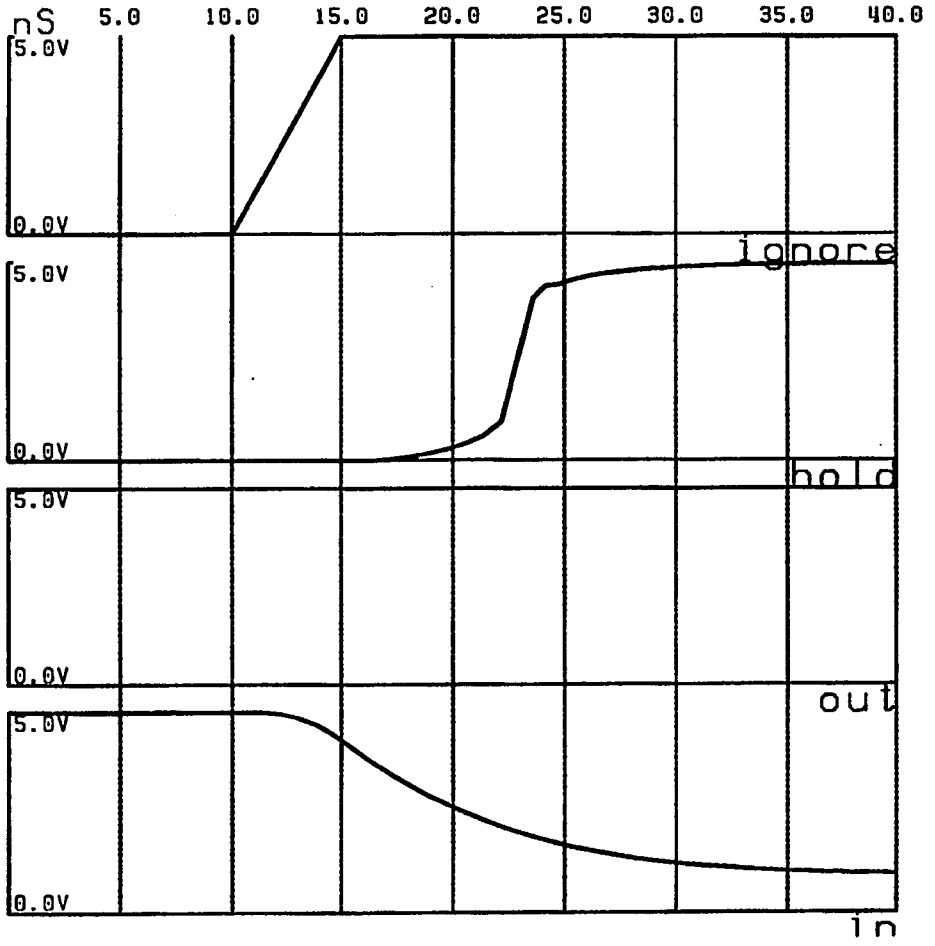


Figure 5.11. Spice Simulation of Flow Control Circuit.

5.5 Summary of Functional Unit Implementations

The design for the communication processor that was offered in the preceding sections can be easily partitioned for implementation across several chips. One possibility is to partition the functional units into the following groups: the control unit, the input and output ports, and the arrival tracking unit; the packet buffer units, the crossbar switch, and the flow control units; and the buffer RAM. Such a partitioning allows the use of off-the-shelf RAMs while yielding reasonable pin and transistor counts on each chip.

5.6 Buffer and Packet Sizes

Our quasi-adaptive packet based message transport scheme dedicates a fixed number of packet buffers to each packet buffer unit in the communication processor. Since the total amount of packet storage within a communication processor will be limitied, attempts at tuning performance can be made by varying the packet size and the number of buffers while holding the product of the two constant. The simulation results reported in the following section assume packet lengths of 32 bytes with 13 packet buffers per packet buffer unit. In an effort to investigate the tradeoff between the number of buffers and packet size, we compare the results from the following section with two additional sets of simulation runs. The first of these sets differs by doubling the number of packets that may be handled by each packet buffer unit to 26. The second set of simulations keeps the number of packets that may be handled by each packet buffer unit the same at 13 and, instead, doubles the size of the packets to 64 bytes. The results of this comparison are included in the following section.

5.7 Simulation Results

We explain the simulation metrics and parameters, and present and discuss the results for wormhole routing, our original fixed packet routing and our improved quasi-adaptive

version of packet routing in this section. The primary performance metric presented is the average elapsed time from the moment a message is queued for sending on its source node until it reaches its destination. We have also compared the times for messages of specific lengths and distances. These results correlate well with the average values that are presented. All results are given for two different average message lengths across a variety of traffic loads. As a further check, we have also compared both the maximum times taken by any message and the total simulation times. Neither of these checks indicate anomalies in any of the simulation cases.

For all cases, message destinations are chosen uniformly. The message lengths are given by an exponential distribution with a mean of 512 bytes for the first set of results and with a mean of 2048 bytes for the second set. The intergeneration time for messages at each node is given by a normal distribution. This data was gathered at nine different rates of message generation. For the shorter messages the mean of the intergeneration time ranges from 1024 ticks to 9216 ticks, with two increases of 256 ticks, followed by three increases of 512 ticks and, finally, three increases of 2048 ticks. The variance is always equal to half of the mean. For the longer messages the mean of the intergeneration time ranges from 4096 ticks to 36846 ticks, with increases of 1024 ticks, 2048 ticks and 8192 ticks. The link transfer rate is 2 ticks per byte and arbitration for shared resources is assumed to take 4 ticks. These values are derived from the implementation which was presented in the preceding sections. For all routing schemes the same pseudo-random sequence of numbers are generated for the message destination, length and intergeneration time. This ensures that the kth message will be of length l and will travel from source s to destination d starting at time t for all of the routing schemes evaluated. In all cases, simulations are performed for hypercubes of degree 6.

The intergeneration time for messages is expressed as an ideal link utilization value. This value is derived by calculating the total amount of link time required to handle the transfer of all of the messages generated during the simulation. For example, a message of length M that travels L hops will contribute $2M \cdot L$ link ticks to the link time total.

This time is then divided by the product of the total number of links in the hypercube and the time from the start of the simulation until the last message is generated. This is not a true utilization value in the sense that we only integrate the total available link capacity up until the time of the last message generation. However, this does meet our primary objective providing a utilization value that is the same for all transport schemes with the same simulation parameters. This is guaranteed because the time of generation for the last message is always the same; whereas, the total time varies somewhat for each different routing mechanism.

The simulation results are given in Figs. 5.12 and 5.13. The figures for each of the two different message lengths are given in two parts with differing vertical scales. By examining the link utilization axis it can be seen that the two parts of Fig. 5.12 overlap by two data points. Similarly, with Fig. 5.13. The mean elapsed times for both the arrival of the first packet (or 32 data bytes) and the last byte are shown. This allows us to see the effect that the packet interleaving has on both latency and bandwidth. The *first* times (e.g., the line labelled *first wormhole*) indicate message latency. The difference between corresponding *first* and *last* times (e.g., *last wormhole* less *first wormhole*) provide an indication of the message bandwidth.

For the smaller messages our initial fixed routing packet scheme leads to decreases in message latency times that range from 35% of the wormhole latency times for the heaviest link utilization values to 87% of the wormhole time for the lightest utilization value. The latency times remain below 50% of the wormhole times for the four heaviest utilization values. On the other hand, bandwidth for the fixed packet scheme has also decreased from about 37% of that of the wormhole scheme for the heavier loads to about 81% for the lighter loads. Even with the moderate decreases in bandwidth incurred by the packet routed scheme, the arrival time for entire messages is quicker for the three heaviest traffic loads.

Blockage at the source nodes limits improvements in latency times for the large messages at higher link utilizations. For the lower link utilizations the improvements in

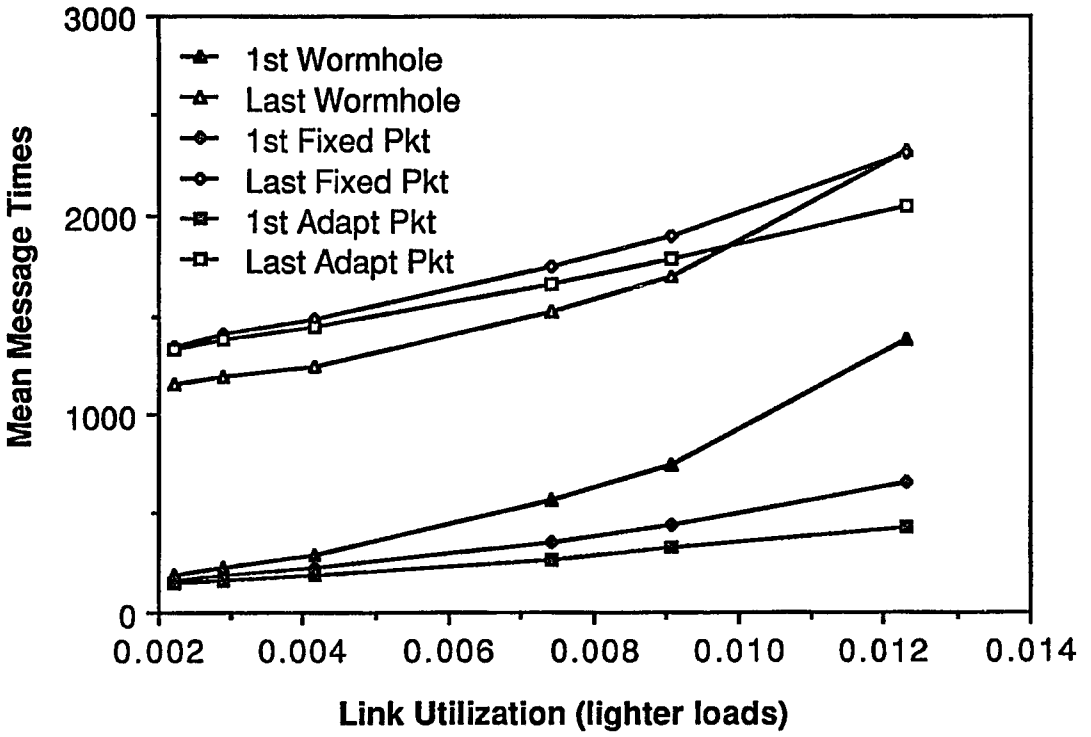


Figure 5.12. Mean Message Times, Length = exp(512) (Part 1-of-2).

latency time are greater than they were for the shorter messages. This occurs because the benefits of interleaving packets from different messages have a much greater impact as message size increases. For the longer messages our initial fixed routing packet scheme leads to decreases in latency times that ranged from 47% of the wormhole times to 69% of the wormhole times. Bandwidth ranges from 39% to 72% of the wormhole times.

The change to the quasi-adaptive packet scheme leads to significant improvements. For smaller messages the latency times decrease to less than 50% of the times for wormhole routing for all but the 3 lightest loads. The latency times range from 10% to 77% of those for wormhole routing. Bandwidth for our quasi-adaptive packet scheme ranges from about 28% of that for wormhole at the heaviest traffic load to about 81% at the lightest load. For the four heaviest traffic loads the average time for the messages to completely arrive is less with the quasi-adaptive packet scheme than it is with wormhole routing.

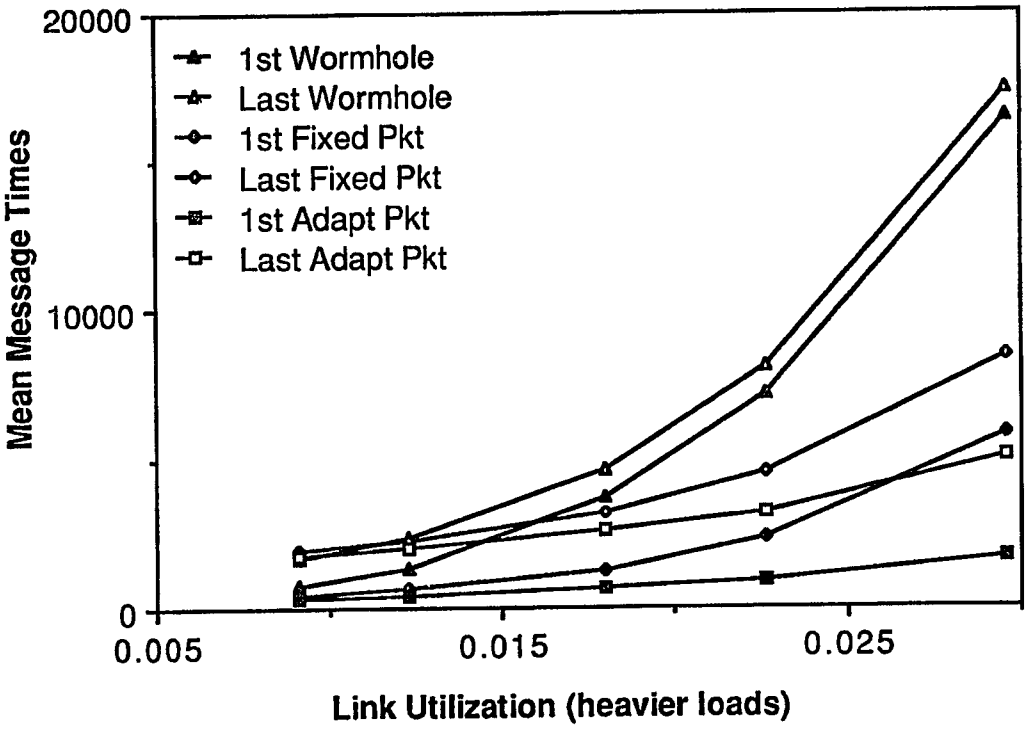


Figure 5.12. Mean Message Times, Length = exp(512) (Part 2-of-2).

The results are even better for the large messages. Message latency ranges between 14% and 31% of that for wormhole routing across the entire range of loads. Bandwidth for the adaptive packet scheme ranges from about 31% of that of wormhole at the heaviest traffic loads to about 72% at the lightest.

The decreases in bandwidth will likely have an insignificant effect on most programs. Consider, for example, that the NCUBE hypercube with on-chip floating point hardware consumes message data at a rate of 0.188 Mbytes per second when performing the double precision vector operation: $\vec{X} = a\vec{X} + \vec{Y}$. Peak message bandwidth on the same system is 0.77 Mbytes per second. Bandwidth would have to decrease by more than a factor of four in any system with a similar calculation to bandwidth ratio before performance in the above operation would begin to be affected. Even then, our packet based scheme still has the advantage of being able to start processing message data much sooner. It is also the case that the difference in bandwidth between wormhole routing and our packet scheme is the least significant in those cases where our improvement in latency times

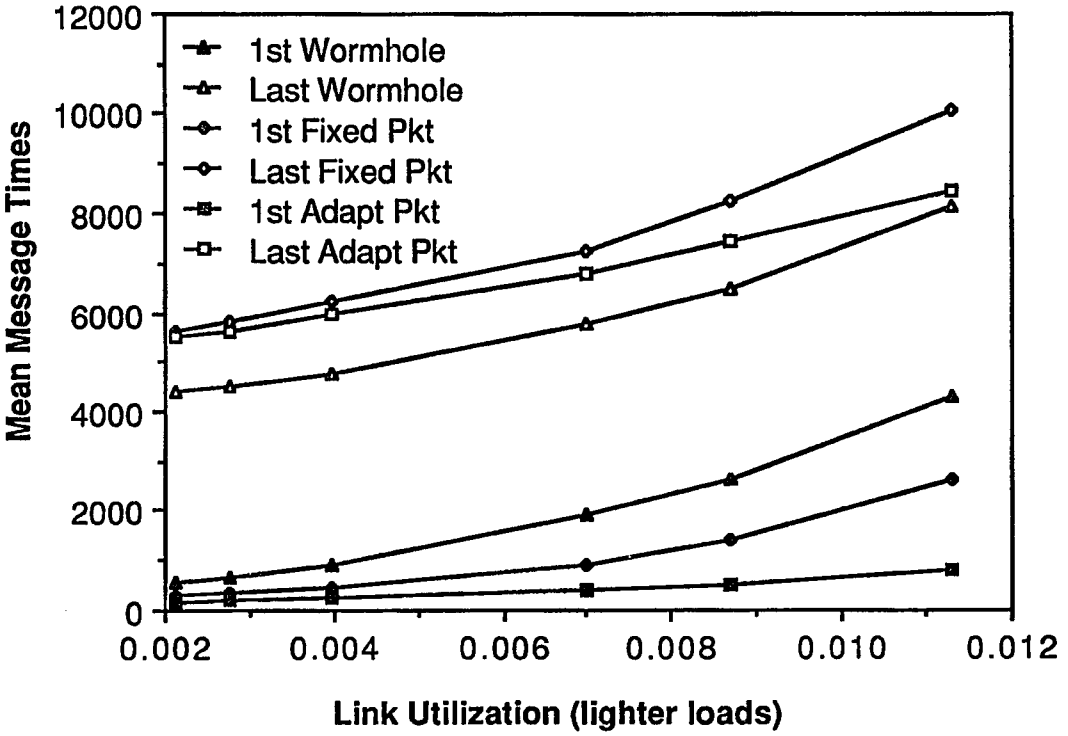


Figure 5.13. Mean Message Times, Length = exp(2048) (Part 1-of-2).

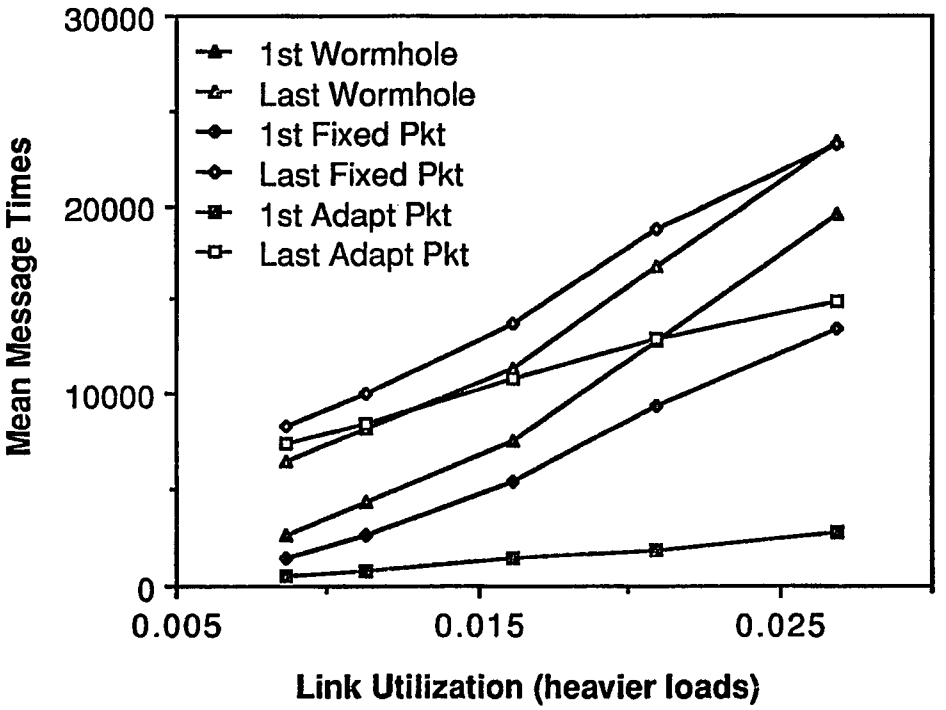


Figure 5.13. Mean Message Times, Length = exp(2048) (Part 2-of-2).

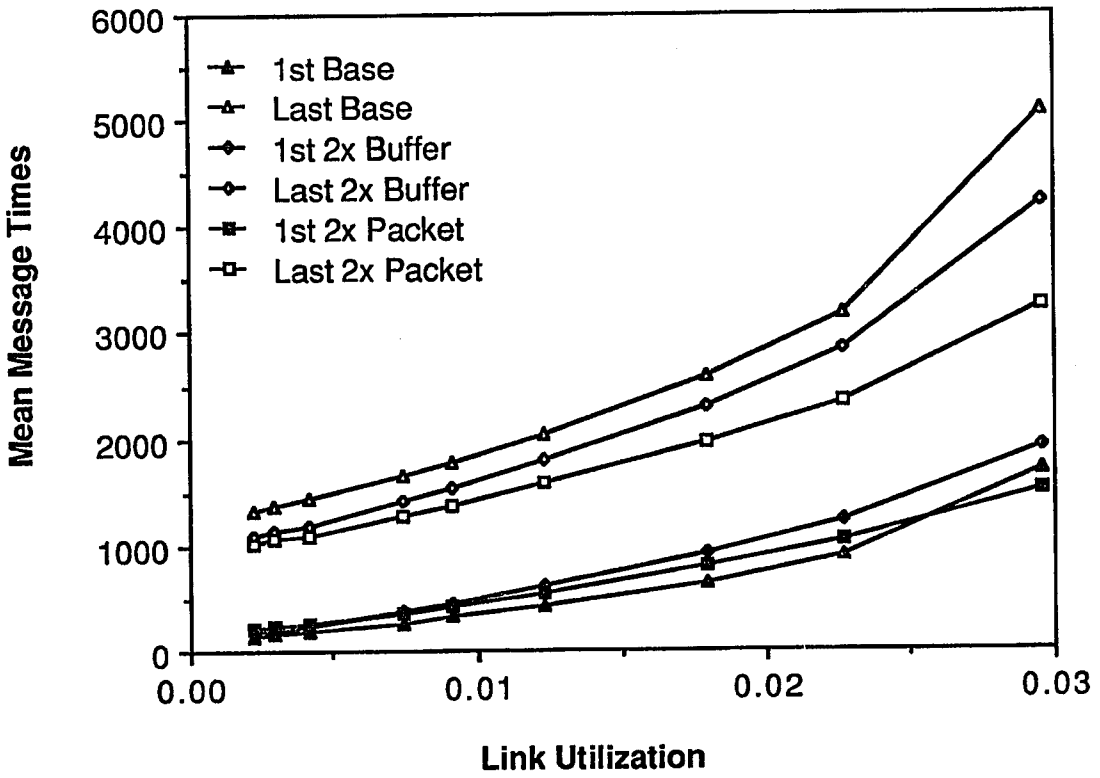


Figure 5.14. Message Times, Buffer/Packet Tradeoffs, Length = exp(512).

are also least significant. Considering all of the above, it seems reasonable to expect that most programs would be able to reap the benefits of reduced message latency times without incurring any costs from the effectively reduced message bandwidth by using our quasi-adaptive packet based routing scheme.

The tradeoff between increasing packet size and the number of packets is shown in Figs. 5.14 and 5.15. From these figures we can see that it is preferable to increase the length of packets rather than the number of packet buffers. This trend is consistent for the arrival of the last packet and should remain so until packet length begins to approach the average message length. However, if we look closely at the arrival times of the first packets we see that for the shorter messages increases the additional time required transmit the larger packet does not lead to lower message times until the traffic load gets very high. The tradeoffs in packet and buffer sizes are summarized by the follows.

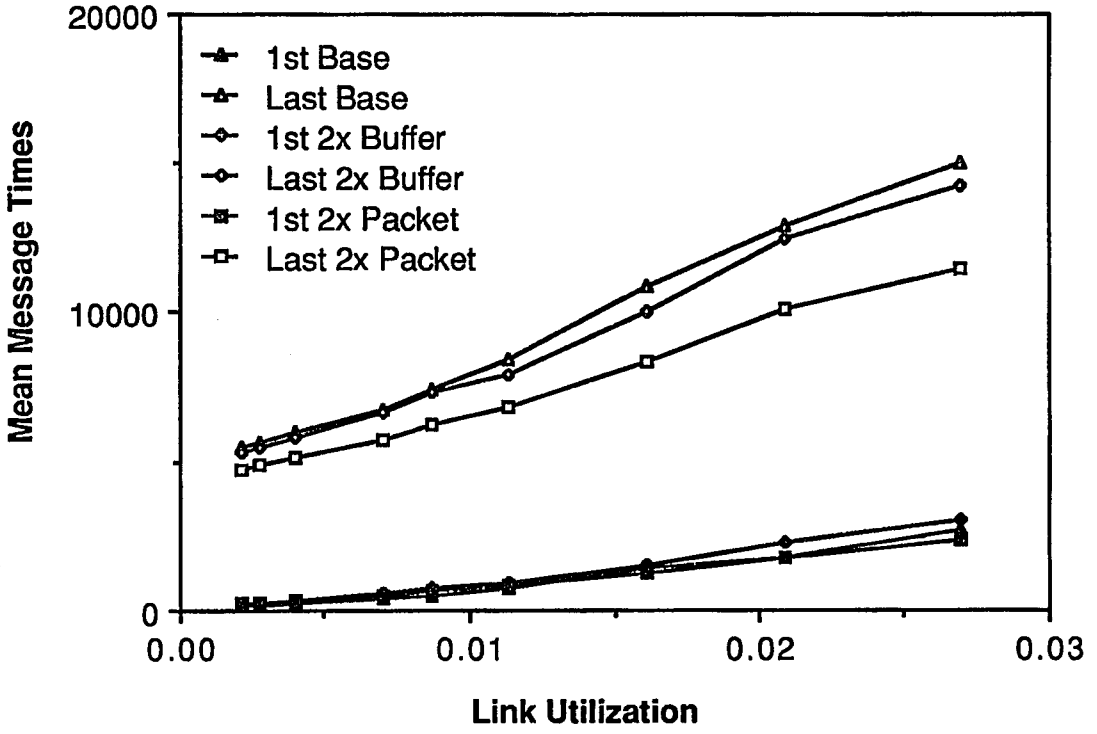


Figure 5.15. Message Times, Buffer/Packet Tradeoffs, Length = exp(2048).

Adding buffers provides the greatest performance gain until an adequate supply of buffers is available. An adequate number of buffers appears to be from about 1 to $1\frac{1}{2}$ times the order of the cube. Once an adequate supply of buffers is available the arrival times for the total packet can be improved by increasing the packet size until the point where packet size approaches the average message size. However, increasing packet sizes may also lead to increases in the delivery time for an individual packet. Therefore, in systems where the latency of the first packet is important (i.e., systems where message use is to be overlapped with message arrival), packet sizes should be limited to a small fraction of average message size. In the systems that we simulated, packet sizes of about 10% of the average message size worked best.

5.8 Chapter Highlights

In this chapter we have developed a new communication architecture: its functionality has been explained and its implementation has been described. We concluded by providing a set of performance simulation results for random communication loads. Our results indicate that the performance potential for our scheme exceeds that of wormhole routing which is among the most preferred of the existing schemes.

CHAPTER 6

CONCLUSION

A large potential exists for increasing the communication performance of hypercube multiprocessors. The development of this potential will allow an increasing variety of algorithms to be efficiently executed on these systems. In this dissertation we have introduced a classification scheme for parallel programs and their communications. The classification scheme established a framework to facilitate discussion of communication system issues. This framework was then used to identify classes of programs with similar communication characteristics. Further discussion of communication issues was divided into two broad groups: NN and BC communications and RA communications. Support for communications from both of these groups was combined into a new communication architecture called quasi-adaptive packet routing that we introduced in Chapter 5.

Within the class of programs that use NN and BC communications, we identified a large subclass of communications that we called C-deterministic. Programs that have C-deterministic communications contain an inherent knowledge of their communication patterns. This knowledge allows us to remove much of the generality that is built into typical communication protocols (e.g., NCUBE's Vertex, Intel's IHOS, and to a lesser extent Caltech's CrOS). By streamlining the protocols and providing more asynchronous and efficient semantics for the communication routines that interface to user programs we were able to realize significant performance improvements. These improvements surpassed those reported by other groups that have addressed similar issues [BFFO87,BF87b]. For

C-deterministic programs, the improvement in communication support does not require any architectural changes.

The above improvements may also be realized for the class of NN and BC communications. However, achieving the improvements for this class requires additional effort because the NCUBE architecture cannot handle messages of unknown size without using the cumbersome three-way handshake protocol that was described in Sec. 3.1.1 (similar situations exist for the other first generation architectures). Information about subsequent messages may be embedded in earlier messages, providing a *sender driven* form of C-determinism. Alternatively, the minor hardware changes described in Sec. 3.3 could be implemented. Either of the above changes would enable the largest class of existing hypercube programs (those that use BC and NN communications) to execute with significantly increased communication performance. We consider this to be one of the two key contributions of this thesis. Our experiments on two representative programs show that the communication performance was improved by a factor of about 3 on the program that employed NN communications and about 2 on the program with BC communications. Total program execution time decreased by a factor of about 1.3 as a result of the improved communications. This suggests that programs designed for high performance communication systems will be able to take greater advantage of the increase in effective CPU power that is available as a consequence of the reduced demands of the more powerful communication system.

Our study of NN and BC communications included measuring the times for all of their constituent operations. As a result of the insights obtained from these experiments, we were able to optimize several of these operations and embody the improvements in our communications module. The optimization of some of the remaining operations require architectural changes. Simulation results that predict the effect of the architectural changes indicate additional, substantial improvements. We also showed that most of the optimizations identified for NN and BC communications are also applicable to RA communications.

Our study of RA communications began by considering the requirements of application programs. This provided the basis for the set of communication operations and semantics that we introduced. We then considered many different strategies for transporting messages among non-neighboring nodes. Several key design choices were simulated. As a result of our study, we chose two transport strategies for further consideration: wormhole routing, which is an existing scheme, and the quasi-adaptive packet routing scheme that we developed in the latter sections of this dissertation. The limitations of wormhole routing were then analyzed and our quasi-adaptive transport scheme, which was designed to avoid the chief limitations of wormhole routing, was introduced.

We designed, as a proof of principle, our quasi-adaptive communication system. The discussion of this system began with an overview of the architecture. This was followed by a discussion of the data structures that are maintained by the communication processor and an explanation of the message transport semantics. The issues of deadlock and bottlenecks at the message source were discussed. A more detailed description of the functional units that constitute the communication processor was then given. Finally, we offered simulation results that compared the performance of various versions of our design and the wormhole routing scheme. These results demonstrated the potential for significant improvements in the execution time of RA programs (results for NN and BC programs were given earlier) that would result from the use of our proposed communication architecture. We consider the potential for significant RA program performance improvements to be the second key contribution of this thesis.

6.1 Future Work

An important direction for future work is to investigate the complete requirements for the efficient support of a shared virtual memory system. The existence of a high performance communication system is certainly necessary, but not sufficient. Providing a shared virtual memory programming environment may prove to be the crucial first step in integrating large scale distributed-memory multiprocessors into the mainstream of

general purpose computing.

BIBLIOGRAPHY

[ABG85] Mauricio Arango, Hussein Badr, and David Gelernter. Staged circuit switching. *IEEE Transactions on Computers*, C-34:174–180, February 1985.

[AC87] Steven Anderson and Marina C. Chen. Parallel branch-and-bound algorithms on the hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 309–317, SIAM, Philadelphia, 1987.

[AM88] Tarek S. Abdel-Rahman and Trevor N. Mudge. Parallel best first branch and bound algorithms on hypercube multiprocessors. In Geoffrey C. Fox, editor, *Proceedings Third Conference on Hypercube Concurrent Computers and Applications*, ACM, 1988.

[Ame87a] *Ametek Series 2010.* Sales Brochure, Ametek Computer Research Division, 606 E.Huntington Drive, Monrovia, CA 91016, 1987.

[Ame87b] *Ametek Series 2010.* News Release, Ametek Computer Research Division, 606 E. Huntington Drive, Monrovia, CA 91016, 1987.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System*, pages 200–211. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[BF87a] C. Baillie and J. Flower. *CrOS III and Cubix on the NCUBE.* Caltech Concurrent Computation Project Technical Report C3P-432, California Institute of Technology, May 1987.

[BF87b] Clive Baillie and Jon Flower. *CrOS III on Ncube – the Limits.* Caltech Concurrent Computation Project Technical Report C3P-492, California Institute of Technology, 1987.

[BFFO87] Clive Baillie, Ed Felton, Jon Flower, and Steve Otto. *CrOS III+ on Ncube – CrOS III Plus a Library of Super-fast Functions.* Caltech Concurrent Computation Project Technical Report C3P-434, California Institute of Technology, 1987.

[BFHP87] Donna Bergmark, Joan M. Francioni, Brenda K. Helminen, and David A. Poplawski. On the performance of the FPS T-series hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, SIAM, Philadelphia, 1987.

[BI85] Sandeep N. Bhatt and Ilse C. F. Ipsen. *How to Embed Trees in Hypercubes.* Research Report YALEU/DCS/RR-443, Department of Computer Science, Yale University, December 1985.

[Bra86] J. E. Brandenburg. Description and analysis of concurrent symbolic programs with dynamic load balancing. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986*, SIAM, Philadelphia, 1986.

[BS86] Joseph E. Brandenburg and David S. Scott. *Embeddings of Communication Trees and Grids into Hypercubes.* Technical Report 1, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006, 1986.

[CMP87a] E. Chow, H. Madan, and J. Peterson. *Hyperswitch Network for the Hypercube Concurrent Computer.* Concurrent Processor Systems Group, Technical Report, NASA Jet Propulsion Laboratory, 1987.

[CMP*87b] E. Chow, H. Madan, J. Peterson, D. Grunwald, and D. Reed. *Hyperswitch Network for the Hypercube Computer.* Caltech Concurrent Computation Project C3P-484, California Institute of Technology, 1987.

[CMP*88] E. Chow, H. Madan, J. Peterson, D. Grunwald, and D. Reed. Hyperswitch network for the hypercube computer. In *15th Annual International Symposium on Computer Architecture*, pages 90–99, May 1988.

[CT87] Tony F. Chan and Ray S. Tuminaro. Implementation of multigrid algorithms on hypercubes. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 730–737, SIAM, Philadelphia, 1987.

[Cub] *Cubelib–Software Library.* Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006.

[Dal86a] William J. Dally. *On the Performance of k-ary n-cube Interconnection Networks.* Department of Computer Science Technical Report 5228:TR:86, California Institute of Technology, 1986.

[Dal86b] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* PhD thesis, Department of Computer Science, California Institute of Technology, 1986.

[DP78] A. Despain and D. Patterson. X-tree: a tree structured multiprocessor computer architecture. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, pages 144–151, April 1978.

[DS86] William J. Dally and Charles L. Seitz. The Torus Routing Chip. *Distributed Computing*, 1:187–196, 1986.

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

[FJL*88] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors Volume I: General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[FK86a] Geoffrey C. Fox and Adam Kolawa. Implementation of the high performance crystalline operating system on the Intel iPSC hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986*, pages 269–271, SIAM, Society for Industrial and Applied Mathematics, 1400 Architects Bldg., 117 South 17th Street, Philadelphia, PA 19103, 1986.

[FK86b] Geoffrey C. Fox and Steve W. Kolawa. Concurrent computation and the theory of complex systems. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986*, pages 244–268, SIAM, Society for Industrial and Applied Mathematics, 1400 Architects Bldg., 117 South 17th Street, Philadelphia, PA 19103, 1986.

[Fly66] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21, September 1966.

[FMM87] Michael J. Flynn, Chad L. Mitchell, and Johannes M. Mulder. And now a case for more complex instruction sets. *Computer*, 71–83, September 1987.

[FMO*87] E. Felten, R. Morison, S. Otto, K. Barish, R. Fatland, and F. Ho. Chess on a hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 327–332, SIAM, Philadelphia, 1987.

[FO88] Edward W. Felten and Steve W. Otto. Chess on a hypercube. In Geoffrey C. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, ACM, 1988.

[Fox85] Geoffrey Fox. *The Performance of the CalTech Hypercube in Scientific Calculations, a Preliminary Analysis.* Technical Report CALT-68-1298, Hm161, California Institute of Technology, 1985.

[Fuj83] R. M. Fujimoto. *VLSI Communication Components for Multicomputer Networks.* PhD thesis, Computer Sciences Division, University of California, Berkeley, 1983.

[Gel81] David Gelernter. A dag-based algorithm for prevention of store-and-forward deadlock in packet networks. *IEEE Transactions on Computers*, C-30:709–715, October 1981.

[GMB88] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing (to appear)*, July 1988.

[GR88] Dirk Grunwald and Daniel Reed. Multiprocessor computer networks: measurements and prognostications. In Geoffrey C. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, ACM, 1988.

[Gun81] K. D. Gunther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communications*, COM-29:512–524, April 1981.

[Har69] Frank Harary. *Graph Theory*. Addison-Wesley, Reading, 1969.

[HC85] M. Horowitz and P. Chow. The MIPS-X microprocessor. In *Proceedings of Wescon*, Stanford University, 1985.

[Hea87] Michael T. Heath. *Hypercube Multiprocessors 1987*. SIAM, Philadelphia, 1987.

[Hil85] W. Daniel Hillis. *The Connection Machine*. The MIT Press, Cambridge, Massachusetts, 1985.

[HJ86] Ching-Tien Ho and Lennart Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 640–648, IEEE, 1986.

[HMS*86] John P. Hayes, Trevor N. Mudge, Quentin F. Stout, Steve Colley, and John Palmer. A microprocessor-based hypercube supercomputer. *IEEE MICRO*, 6–17, October 1986.

[HS86] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 1170–1183, December 1986.

[Hyp] Hypernet system 14/n data sheet. Ametek Computer Research Division, 606 E. Huntington Drive, Monrovia, CA 91016.

[IC 87] *IC Master Volume II*. Hearst Business Communications, Inc., 645 Stewart Avenue, Garden City, NY 11530, 1987.

[iPS85] *iPSC User's Guide*. Intel Scientific Computers, Santa Clara, California, order number 175455-001 edition, 1985.

[Joe79] A. E. Joel. Circuit switching: unique architecture and applications. *IEEE Computer*, 10–22, June 1979.

[KJ87] A. E. Kayaalp and R. Jain. Parallel implementation of an algorithm for three-dimensional reconstruction of integrated circuit pattern topography using the scanning electron microscope stereo technique on the NCUBE. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 438–444, SIAM, Philadelphia, 1987.

[KK79] P. Kermani and L. Kleinrock. Virtual cut-through: a new computer communication switching technique. In *Computer Networks, Volume 3*, pages 267–286, North-Holland, Amsterdam, 1979.

[Lan82] C. R. Lang. *The Extension of Object-Oriented Languages to a Homogeneous Concurrent Architecture.* Department of Computer Science, Technical Report, 5014:TR:82, California Institute of Technology, 1982.

[LYL87] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data prefecthing in shared memory multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 28–31, 1987.

[MA87] Trevor N. Mudge and Tarek S. Abdel-Rahman. Vision algorithms for hypercube machines. *Journal of Parallel and Distributed Computing*, 4(2):79–94, 1987.

[MBA87] Trevor N. Mudge, Gregory D. Buzzard, and Tarek S. Abdel-Rahman. A high performance operating system for the ncube. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 90–99, SIAM, Philadelphia, 1987.

[mbo87] Anant Agarwal et. al. On-chip instruction caches for high performance processors. In *Proceedings of Advanced Research in VLSI*, Stanford University, March 1987.

[MC685] *MC68020 32-Bit Micrporocessor User's Manual.* Motorola Incorporated, second edition edition, 1985.

[MGN79] G. M. Masson, G. C. Gingher, and S. Nakamura. A sampler of circuit switched networks. *IEEE Computer*, 32–48, June 1979.

[MHW87] Trevor N. Mudge, John P. Hayes, and Donald C. Winsor. Multiple bus architectures. *Computer*, 42–48, June 1987.

[Mot83a] *Motorola High-Speed CMOS Integrated Circuits.* Motorola Incorporated, 1983.

[Mot83b] *Motorola Schottky TTL Data.* Motorola Incorporated, 1983.

[MP85] Mohammad Malkawi and Janak Patel. Compiler directed memory management policy for numerical programs. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 97–106, December 1985.

[MS80] P. M. Merlin and P. J. Schweitzer. Deadlock avoidance in store-and-forward networks–I: store-and-forward deadlock. *IEEE Transactions on Communications*, COM-28:345–354, March 1980.

[MWP*87] William R. Martin, Tzu-Chiang Wan, Doug Poland, Trevor N. Mudge, and Tarek S. Abdel-Rahman. Monte carlo photon transport on the NCUBE. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 454–463, SIAM, Philadelphia, 1987.

[NCU85] *NCUBE Handbook.* NCUBE Corporation, 1815 N.W. 169th Place, Suite 2030, Beaverton, OR 97006, 1985.

[Nug88] Steve Nugent. The iPSC/2 direct-connect communications technology. In Geoffrey C. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, ACM, 1988.

[OM87] O. A. Olukotun and T. N. Mudge. A preliminary investigation into parallel routing on a hypercube computer. In *Proceedings of the Design Automation Conference*, pages 814–820, June 1987.

[PFL75] R. C. Pearce, J. A. Field, and W. D. Little. Asynchronous arbiter module. *IEEE Transactions on Computers*, 931–932, September 1975.

[PLG87] Chrisila C. Pettey, Michael R. Leuze, and John J. Grefenstette. Genetic algorithms on a hypercube multiprocessor. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 333–341, SIAM, Philadelphia, 1987.

[PR87] David A. Poplawski and David O. Rich. *Code Paging on Hypercubes.* Computer Science Technical Report CS-TR 87-1, Michigan Technological University, January 1987.

[PTLP85] J. Peterson, J. Tuazon, M Liberman, and D. Pniel. The Mark III hypercube-ensemble concurrent computer. In *Proceedings of the International Conference on Parallel Processing*, pages 71–73, 1985.

[Qui87] Michael J. Quinn. Implementing best-first branch-and-bound algorithms on hypercube multicomputers. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 318–326, SIAM, Philadelphia, 1987.

[RF87] Daniel A. Reed and Richard M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing*, pages 138–144. The MIT Press, 1987.

[RG87] Daniel A. Reed and Dirk C. Grunwald. The performance of multicomputer interconnection networks. *Computer*, 63–73, June 1987.

[SB77] Herbert Sullivan and Theodore R. Bashkow. A large scale, homogeneous, fully distributed parallel machine, I. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 105–117, 1977.

[SBK77] Herbert Sullivan, Theodore R. Bashkow, and David Klappholz. A large scale, homogeneous, fully distributed parallel machine, II. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 118–124, 1977.

[SC81] Charles H. Sauer and K. Mani Chandy. *Computer Systems Performance Modeling*, pages 194–212. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Sei85] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28:22–33, January 1985.

[SPCC] Jon S. Squire and Sandra M. Palais. Programming and design considerations of a highly parallel computer. In *AFIPS Conference Proceedings*, pages 395–400, 1963, SJCC.

[Spe81] Alfred Z. Spector. *Multiprocessing Architectures for Local Computer Networks*. PhD thesis, Stanford University, August 1981.

[SW87] Quentin F. Stout and Bruce Wagar. Passing messages in link-bound hypercubes. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 251–257, SIAM, Philadelphia, 1987.

[Swa86] P. N. Swarztrauber. *Multiprocessor FFTs*. preprint, National Center for Atmospheric Research, Boulder, Colorado, November 1986.

[SWar] Quentin F. Stout and Bruce A. Wagar. Intensive hypercube communication I: prearranged communication in link-bound machines. *Journal of Parallel and Distributed Computing*, to appear.

[SZ87] Steven R. Seidel and Lynn R. Ziegler. Sorting on hypercubes. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 285–291, SIAM, Philadelphia, 1987.

[Tou80] S. Toueg. Deadlock-and livelock-free packet switching networks. In *Proceedings of the 12th ACM Symposium on the Theory of Computing*, pages 94–99, 1980.

[TPPL85] J. Tuazon, J. Peterson, M Pniel, and D. Liberman. Caltech/JPL Mark II hypercube concurrent processor. In *Proceedings of the International Conference on Parallel Processing*, pages 666–671, 1985.

[TU79] S. Toueg and J. D. Ullman. Deadlock-free packet switching networks. In *Proceedings of the 11th ACM Symposium on the Theory of Computing*, pages 89–98, 1979.

[Val82] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal of Computing*, 350–361, May 1982.

[Wag87] Bruce Wagar. Hyperquicksort: a fast sorting algorithm for hypercubes. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 292–299, SIAM, Philadelphia, 1987.

[Wal87] Stephen R. Walton. Performance of the one-dimensional Fast Fourier Transform on the hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 530–535, SIAM, Philadelphia, 1987.

[Wal88] M. Mitchell Waldorp. Hypercube breaks a programming barrier. *Science*, 240:286, April 15 1988.

[WE85] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, 1985.

[WLY85] Benjamin W. Wah, Guo-jie Li, and Chee Fen Yu. Multiprocessing of combinatorial search problems. *IEEE Computer*, 93–108, June 1985.