This is to certify that the

dissertation entitled

INTERPROCESSOR COMMUNICATION

IN DISTRIBUTED MEMORY MULTIPROCESSORS

presented by

Youran Lan

has been accepted towards fulfillment
of the requirements for

Ph. D. degree in Computer Science

Lionel M. Ni
Major professor

Date July 26, 1988

INTERPROCESSOR COMMUNICATION
IN
DISTRIBUTED MEMORY MULTIPROCESSORS

By

Youran Lan

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1988

ABSTRACT

INTERPROCESSOR COMMUNICATION IN DISTRIBUTED MEMORY MULTIPROCESSORS

By

Youran Lan

Distributed memory multiprocessors (DMMPs) have gained much attention recently due to their unique architectural characteristics to establish a massively parallel processing environment. In such systems, the interprocessor communication mechanism has been identified as the major source of system bottleneck. Thus, an efficient interprocessor communication mechanism is the key to the future success of DMMPs. This motivates the study of a fast, versatile, and fault-tolerant interprocessor communication mechanism for DMMPs.

Both *store-and-forward* and *virtual cut-through* communication techniques are discussed. Formal models are developed to facilitate the performance comparison of these two techniques. Three types of interprocessor communication patterns are demanded from application point of view, which are *unicast* (one-to-one), *multicast* (one-to-many), and *broadcast* (one-to-all). A graph theoretical model, namely the optimal multicast tree, is proposed to characterize these communication patterns and to define the performance evaluation criteria, *time* and *traffic*.

The multicast communication, in particular, is highly demanded, but not directly supported by any existing DMMP. A distributed multicast algorithm based on a heuristic greedy method is proposed. In addition to guaranteeing a shortest path for message delivery between the source and each destination, the total traffic created is very close to

the optimal solution. More importantly, the algorithm can be efficiently implemented in hardware using the virtual cut-through technique. The architecture of the hardware router is presented. A prototype router design for a 3-cube has been fabricated by MOSIS using the 3 micron CMOS technology.

Enhancement to the proposed algorithm and its hardware implementation is studied, which allows the communication mechanism to be able to handle interprocessor communication in a faulty hypercube in which each fault-free node has at most one faulty neighboring node.

In summary, centered around the interprocessor communication issue, this dissertation focuses on modeling, algorithm development, and hardware implementation of a versatile and efficient communication mechanism. The proposed communication mechanism is novel in the sense that it is the first communication hardware for DMMPs which directly supports all three types of communications, and the first one which has fault-tolerant capability. It can be readily applied to future generations of DMMPs to significantly increase overall system performance.

To my parents:

Zhen-lu Lan and *Zhu-yu Zhou*

ACKNOWLEDGEMENTS

I would like to thank Professor Lionel M. Ni, my thesis advisor, for his invaluable inspiration and guidance throughout my graduate study. Without the knowledge and time he gave me, this work would have been impossible. I would like to thank Professor Abdol-Hossein Esfahanian, my thesis co-advisor, for directing me to pursue rigorous research approach and for his continual encouragement.

I would like to thank Professors Edwin Kashy, George Stockman, and Anil Jain, for their encouragement and many excellent comments during the course of my dissertation research.

I would like to acknowledge all the faculty members and students who gave me help and assistance during my studying at Michigan State University. Too numerous to list all names, in particular, I would like to express my appreciation to Bruce McMillen, Chung-Ta King, Xiao-la Lin, and Ning Liao, for their valuable help and insightful comments; to Chong-wei Xu, Eric Wu, and Taieb Znati for helpful discussions; and to M. Driscoll, T. Chen, W. Chou, J. Miller, and P. Prins for their excellent work on the detailed layout of the prototype design of the router chip.

I am very grateful to my parents and parents-in-law for their years of concern and support; to my wife Ben-lu for her constant encouragement and direct assistance in the preparing of this manuscript; and to my wonderful daughter Lana for being so understanding.

This work was supported in part by the DARPA ACMP project and in part by the State of Michigan RE/ED project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A *distributed-memory multiprocessor* (DMMP) is a computer system consisting of many processors in which each processor is physically associated with its own local memory and all processors work independently and communicate through an interconnection network connecting the processors. The hypercube multiprocessor is the best known example of DMMPs, and has attracted a great deal of attention in the past few years. Many hypercube multiprocessors, such as the Ncube, iPSC, Ametek, and FPS T-series are commercially available. Massively parallel DMMPs possess the potential of achieving computation power beyond the megaflops range [HMSC86, Wile87]. Interprocessor communication is the fundamental means allowing processors to communicate in such systems. However, it is known that the interprocessor communication mechanism is the major bottleneck of DMMPs, especially in the first generation systems. Centered around the interprocessor communication problem, this dissertation research focuses on modeling, algorithm development, and hardware implementation of a versatile and efficient interprocessor communication mechanism. The proposed interprocessor communication mechanism not only significantly speeds up communication, but also has fault-tolerant capabilities. It can be readily applied to future generations of DMMPs to significantly increase the overall system performance.

1.1 DEMAND OF MASSIVELY PARALLEL MULTIPROCESSORS

The requirements of powerful computer systems which can handle computationally intensive problems are increasingly in demand in many areas. Weather forecasting, simulations, modeling of complex physical phenomena, advanced data base management, artificial intelligence, image processing and pattern recognition, mechanical engineering analysis, and medical diagnosis are some examples among many other large-scale scientific and engineering applications [HwBr84, FoOt84, BaPa86].

To cope with these challenging computational requirements, many advanced computer architectures have been developed. Great efforts have been made toward the following two trends in order to increase computational power by orders of magnitude. One trend is to build vector machines with one or a few very powerful central processors, depending mainly on very fast circuit technology. For example, the Cyber 205 and Cray-X/MP depend heavily on function unit pipelining and interleaved memory modules to gain high preformance. The computational power of a single processor, however, is limited by both physical and architectural bounds. To achieve new performance improvement in such systems has become more and more difficult. Thus, a natural trend is to construct systems consisting of multiple processors, which has been shown in recent years to be the most straightforward and cost-effective approach to achieve high performance. With continuous technological advances in integrated circuits, more powerful and compact, but less expensive, microprocessor, memory and communication chips are available. This makes the construction of massively parallel multiprocessor systems feasible. In such systems, by adding more processors and using a better interconnection network, the performance can be greatly improved to the level far beyond the performance of existing supercomputers.

1.2 DISTRIBUTED MEMORY MULTIPROCESSORS (DMMPs)

Existing multiple processor computer systems can be characterized into two structural classes: SIMD (*single instruction stream / multiple data stream*) architecture and MIMD (*multiple instruction stream / multiple data stream*) architecture. All these architectures are centered around the concept of parallelism [HwBr84, Patt85].

An SIMD computer consists of an array of processors, a network to interconnect these processors, and a central control unit to synchronize the operation of the processors. In an SIMD computer, all processors are executing the same instruction at any given time, but multiple sets of operands are fetched and operated on in multiple processors in a synchronous fashion. All the processors are controlled by the central controller.

Typical examples of SIMD machines include the Illiac IV, which has a central control unit and 64 mesh connected processing elements, each of which has direct connections to four other processing elements [BBKK68]; MPP, a system which contains 16K bit-serial processing elements connected in a two dimensional mesh structure [Batc80]; and the Connection Machine, consisting of 64K bit-serial processors with the structure of cube-connected toroidal lattices [Hill85, SASL85]. Because of the synchronous feature, these machines are good for solving very structured problems, such as matrix operations and image processing, which deal with mainly array data types. For example, the primary purpose of the MPP is to process satellite imagery.

An MIMD computer, or a *multiprocessor system*, consists of many processors communicating through an interconnection mechanism [HwBr84]. In an MIMD computer, all processors work independently in an asynchronous fashion. They execute different instruction streams on different sets of operands. Usually, these processors are homogeneous and the communication delay between processors is relatively small but non-negligible. Because of the multiple instruction stream feature, MIMD computers provide a more general computation model. They are more flexible and versatile than

SIMD computers.

Depending on the structure of the memories in the system, multiprocessor systems can be classified as *centralized-memory* multiprocessors and *distributed-memory* multiprocessors [NiKP87]. In a centralized-memory multiprocessor, all memory modules are equally accessible to all processors. A processor/memory interconnection network, therefore, is needed to allow all processors in the system to access the memory modules. There are several hardware bottlenecks in such a system, which include the number of processors, the memory bandwidth, and the bandwidth of interconnection networks. For example, many commercial products of centralized-memory systems including Sequent's Balance 8000 and 21000, Encore's Multimax, and CRAY-X/MP [DoDu85, HwBr84, Olso85] can have no more than 30 processors.

In a distributed-memory multiprocessor, however, each memory module is physically associated with a processor. No memory is globally accessible. An interprocessor communication network is needed to allow the processors to communicate with each other. Because each memory module is attached to a corresponding individual processor, the performance of an algorithm depends on how well the application problem is partitioned and mapped into the processors. A multiprocessor system may have a mixed memory structure to provide a local memory to each processor and global memory modules shared by all processors.

From the viewpoint of processes, there are two basic process synchronization and communication models. One is the *shared memory* model in which the system has a global memory accessible by all processors. The processes communicate through shared variables. In such a system, the access time to a unit of data is the same for all processors. A hardware device or a software protocol is required in such systems for arbitrating the access to the memory among the processes which share the memory. The shared memory may cause a software bottleneck.

The other process synchronization and communication model is *message-passing* in which processors communicate by explicit message-passing through the interprocessor communication network. In a message-passing system, the performance of an algorithm depends on how well the application problem is partitioned and mapped into the processors, and how efficient the communication mechanism in the system is. Centralized-memory multiprocessors usually adopt the shared memory model, whereas distributed-memory multiprocessors usually prefer the message-passing model [NiKP87].

To construct a massively parallel system consisting of hundreds or even thousands of processors, the distributed-memory structure is a promising approach. In order to reduce the software bottleneck in such a complex system, message-passing is naturally a better choice as the process synchronization and communication mechanism. Therefore, in this study, we consider *message-passing distributed-memory multiprocessors*. From here on, we use distributed-memory multiprocessor (DMMP) with the implication of message-passing model.

1.3 INTERPROCESSOR COMMUNICATION

As mentioned in the previous section, a DMMP system does not have the software bottleneck as in the case of a shared memory system. However, DMMPs do have an interprocessor communication problem.

There are three basic communication types: *unicast* (one-to-one), *broadcast* (one-to-all), and *multicast* (one-to-many). In the first generation DMMPs, only unicast is directly supported. Message routing for broadcast is done by subroutine calls at the source and each intermediate processor. Multicast communication is usually implemented by issuing multiple unicast.

The interprocessor communication mechanism and the computational power of each individual processor in a DMMP system are two of the major factors which affect

the performance of the system. In order to fully explore the computational power of a DMMP, computation and communication must be balanced, that is, a proper communication/computation ratio (about 1) should be achieved [ShFi87].

However, the communication/computation ratios in the first generation DMMPs are very high. Table 1.1 is a comparison of the communication time (to transfer a 64 bit data over a link) to the computation time (to preform a multiplication on two double precision floating-point numbers) for three first generation DMMPs [Duni87].

Table 1.1 Comparison of communication and computation of three DMMPs

Item	AMETEK S-14	Intel iPSC	NCUBE
8-byte transfer time (μs)	640	1120	470
8-byte multiply time (μs)	33.9	43.0	14.7
Comm./Comp.	19	26	32

As can be seen from the table, the communication/computation ratios are all greater than 10. This indicates the communication mechanism provided in the first generation machines does not match the speed of powerful processors and thus becomes the major bottleneck of the system preformance.

Therefore, to provide an efficient interprocessor communication is the key to the successful exploitation of parallelism in DMMPs. With the advent of VLSI technology, processors are becoming faster and more powerful. This causes interprocessor communication become more and more important and will dominate computing cost in both hardware and software [HLSM82, Fox83].

In order to improve the overall system performance, the communication overhead must be significantly reduced. As will be discussed further shortly, hardware implementation of efficient communication algorithms is a necessity for the success of future generation DMMPs. Therefore, we need to develop algorithms which can efficiently handle

all three types of communication, and to consider hardware implementation of the algorithms.

Interprocessor communication in faulty DMMPs is another important issue. Since we are studying highly parallel systems, consisting of up to one thousand or even more processors, the reliability problem becomes a natural concern. A good interconnection topology, such as the hypercube, may provide inherently fault-tolerant capability by having multiple routing paths between each pair of processors. However, when a single processor fails, the routing methods for a fault-free hypercube can no longer be applied. We need to provide a communication mechanism which has certain fault-tolerant capability. Fault-tolerance is important especially when a large job is to be carried out, which requires computation time close to or longer than the average non-fault run time of the system.

1.4 MOTIVATION AND PROBLEM STATEMENT

As indicated in [ReFu87], the emergence of multiprocessor systems in the past years has posed several important and challenging problems in (1) network topology selection, (2) communication hardware design, (3) operating system design, (4) fault tolerance considerations, and (5) algorithm design. In recent years, hypercube topology has stood out as a dominating interconnection topology for message-passing distributed memory multiprocessors. Some experts believe that hypercube multiprocessors are the most promising highly parallel multiprocessor systems [Myer86, Seit85]. However, the first generation hypercubes do not perform as well as people once expected, mainly because of the software implementation of communication algorithms. In order to fully explore the computational power of DMMPs, all the above problems have to be solved. Design of efficient parallel algorithms has been a major research issue in the application domain in the past few years. Development of operating systems better suitable for DMMPs is another important issue in the system software domain [MuBA87, FoKo86].

Some research has been done to develop communication hardware implementing unicast for second generation hypercube multiprocessors [DaSe86, iPSC88]. However, hardware design for multicast communication has received little attention. No fault-tolerant consideration has been made on the existing communication hardware designs.

Motivated by the above observations, our major concern is the communication hardware design and related fault tolerance issues. The objective of this research is to study a fast, versatile, and fault-tolerant message-passing mechanism for DMMPs, in particular, for the hypercube multiprocessors. The major goal is to develop a hardware router which can efficiently handle all three types of interprocessor communications including the multicast communication which is highly demanded but not directly supported by any existing DMMP. The router works not only in fault-free hypercubes, but also in faulty hypercubes in which each fault-free node has at most one faulty neighboring node. The router can be readily applied to future generation DMMPs in order to significantly speedup interprocessor communication and to greatly improve the overall system performance. Our ultimate goal is to make future DMMPs true MIMD supercomputers.

1.5 THESIS ORGANIZATION

This introduction has discussed the requirement of massively parallel computing systems, especially the DMMPs, and the problems existing in current DMMPs. Also, the motivation of this dissertation research has been addressed.

The next chapter gives a brief review of multiprocessor systems and shows why we select distributed-memory multiprocessors. Some background about interconnection topologies, in particular, the hypercube topology, is given, followed by a comparison of existing hypercube multiprocessors.

Chapter 3 addresses some fundamental issues in interprocessor communication which is the key to the performance of such systems.

In Chapter 4, following the presentation of some graph theoretical notation and terminology, the multicast communication is formally modeled as an *Optimal Multicast Tree* (OMT), a graph theoretical problem. The OMT is a model for all three types of communication in a broad sense. Following the OMT model, algorithms for unicast and broadcast are studied first. Then the multicast problem in some simple cases is examined. Based on the observation of the complexity of the problem and previous research on some closely related areas, the OMT problem is conjectured to be an NP-hard problem. After presenting optimal solutions to some special cases, in Chapter 5, a heuristic multicast algorithm for the hypercube environment is presented. The proposed multicast algorithm is distributed in the sense that the overall routing is not calculated solely by the source. Instead, each involved node decides its own routing. Simulation results of the algorithm with comparison to alternative multi-destination routing methods are presented which attest to the efficiency of the proposed algorithm.

From the system designers' point of view, simply proposing a polynomial time complexity algorithm is still not of much interest. Software approaches are well known to be too slow. Thus, besides the theoretical study and performance analysis of the proposed algorithm, the hardware design of a dedicated router, which efficiently implements the algorithm, is presented in Chapter 6.

The problem of routing in faulty hypercubes is studied in Chapter 7. No hardware routing devices developed so far have fault-tolerant capability. Chapter 7 first presents a model for faulty hypercube multiprocessors, and then proposes fault-tolerant routing algorithms and demonstrates how the hardware design presented in Chapter 6 can be modified to work properly in faulty situations.

The last chapter summarizes the work of this dissertation research and presents suggestions for related future research.

CHAPTER 2

DISTRIBUTED MEMORY MULTIPROCESSORS

In order to maximize the performance of a multiprocessor system, the components of the system should be balanced so that there is no bottleneck in the system. As the VLSI technology advances rapidly, the computing power of each individual processor becomes faster and more powerful. This situation puts great pressure on system designers to provide an efficient communication mechanism which can match the speed of the processors. Data communication is the key to the successful exploitation of parallelism, while the interconnection network plays a fundamental role in determining communication efficiency. This chapter discusses why message-passing distributed memory multiprocessor systems have become popular in recent years. Various interconnection topologies are investigated. In particular, the topological properties of hypercube is studied in detail. A brief comparison of some commercially available hypercube multiprocessors is given.

2.1 DMMPs UNDER CONSIDERATION

In this study, a *message-passing distributed-memory multiprocessor* (DMMP) is considered. Figure 2.1 shows the generic structure of such a system.

There are N processors in the system, with N ranging from hundreds to thousands. The processors are interconnected by certain interconnection topology. The processors are also connected to a host processor or I/O processors for communicating with external devices. All processors work independently in an asynchronous fashion and

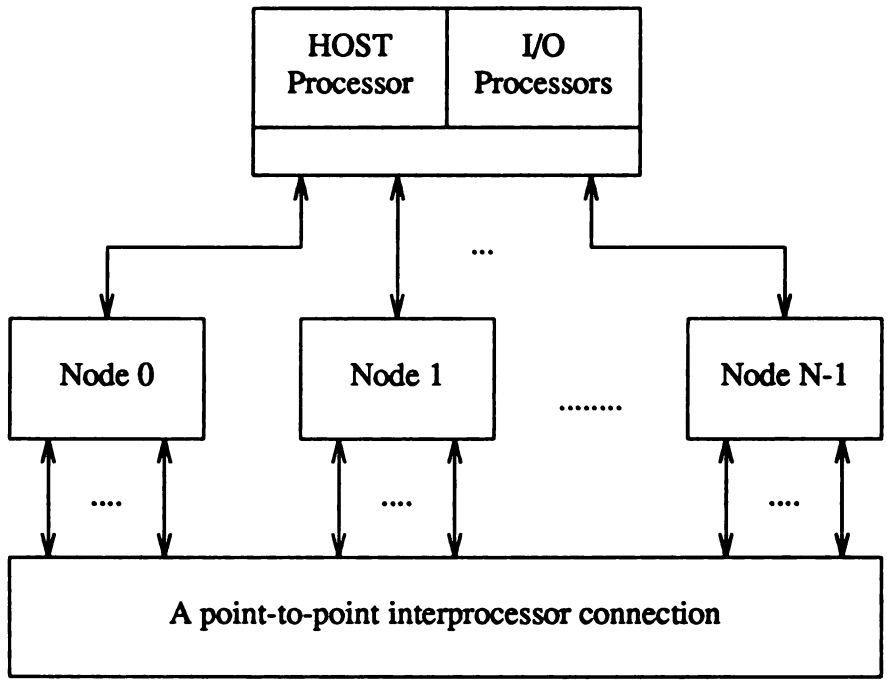


Figure 2.1 The generic structure of the DMMP under consideration

communicate by message-passing through the interconnection network.

Figure 2.2 shows a generic structure of a node in the system. Each node has its own processor and local memory. The processor/memory pair is associated with a router to handle interprocessor communication task. Each router has n incoming links and n outgoing links connected to its n neighboring nodes and another pair of links connected to the local processor. Through the input/output links, the router communicates with the routers of other nodes.

Basically, the processor/memory pair performs computation, while the router handles interprocessor communication. The local processor is involved in the communication task only when the node is a source or a destination.

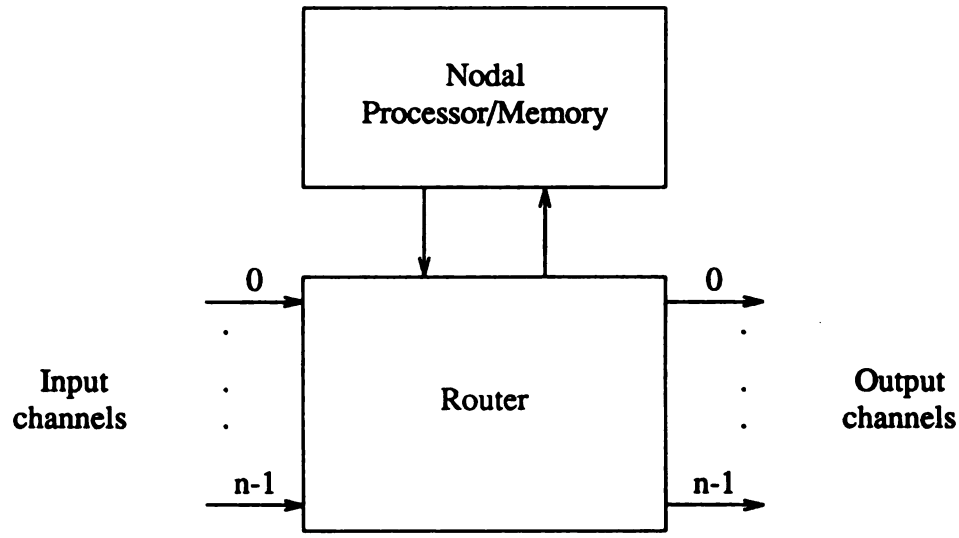


Figure 2.2 The generic structure of individual nodes in the DMMP

2.2 INTERCONNECTION TOPOLOGIES FOR DMMPs

Many interconnection topologies have been proposed for highly parallel multiprocessor systems [Feng81, HLSM82]. Some of them have been adopted in real machines. Various interconnection topologies used in commercial systems can be roughly divided into three categories: *shared bus*, *multistage network*, and *point-to-point* interconnection network.

Compared with other interconnection topologies, the bus structure is least complex and the easiest to construct with low overall system cost for hardware. Also, it is not difficult to modify system configuration by adding or removing some nodes. However, simultaneous communication between multiple pairs of processors is impossible in a shared bus system. Data transfer rate of the bus limits the overall system capacity. Thus, the number of processors in the system is small. Furthermore, the failure of the bus will cause the failure of the entire system. Since the bus-based architecture is not good for

massively parallel DMMPs, it is mainly used in shared memory multiprocessors such as Encore's Multimax and Sequent's Balance.

A multistage network consists of many stages of interconnected switches. Only the first stage and the last stage are connected to nodes. Switches in all intermediate stages are connected to other switches. A typical multistage interconnection network in a multiprocessor system with $N=k^n$ processors consists of n stages with N/k $k\times k$ switch boxes at each stage. A notable feature of the multistage network is that the distance between any pair of nodes is the same. The most popular multistage network is the *Omega (shuffle-exchange)* network and its variations. A generalized $b^n\times b^n$ shuffle-exchange network consists of n stages of $b\times b$ crossbar switches linked by perfect shuffle interconnections. Usually $b=2$, and $n=\log_2 N$. An omega network with $N=8$ and $b=2$ is shown in Figure 2.3.

A typical example of DMMP system using multistage switching network is the BBN Butterfly machine [BBN87]. Its interconnection topology resembles that of a Fast Fourier Transform Butterfly. A 16 node BBN Butterfly interconnection network is shown in Figure 2.4. The interconnection network is constructed by two stages ($\log_4 16=2$) of switching units. Each stage consists of 4 (16/4=4) 4×4 switches.

The major disadvantage of the multistage network is that it is very difficult to scale up the system to contain a very large number of processors, since the cost and the complexity of the switching hardware will grow up rapidly. Thus, multistage networks are good for medium size systems.

In order to reduce the software bottleneck, message-passing seems to be a reasonable choice for process synchronization and communication mechanism. For a distributed-memory multiprocessor system with up to thousands of processors, point-to-point interconnection networks provide a promising interconnection structure.

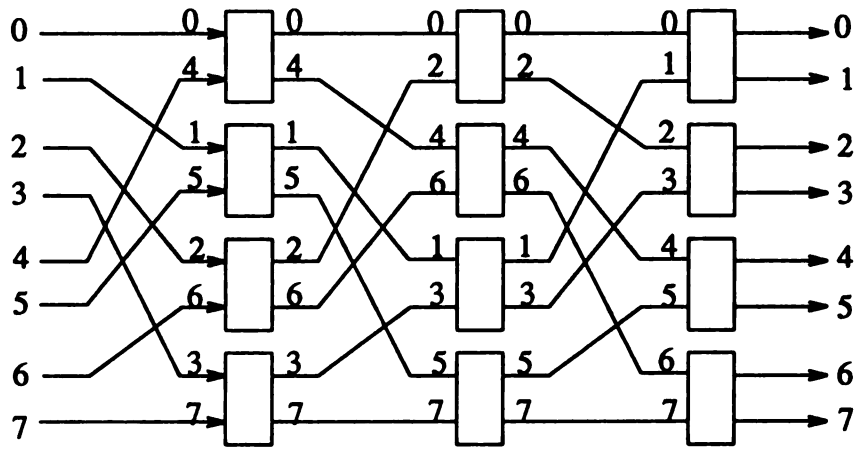


Figure 2.3 An 8×8 Omega network

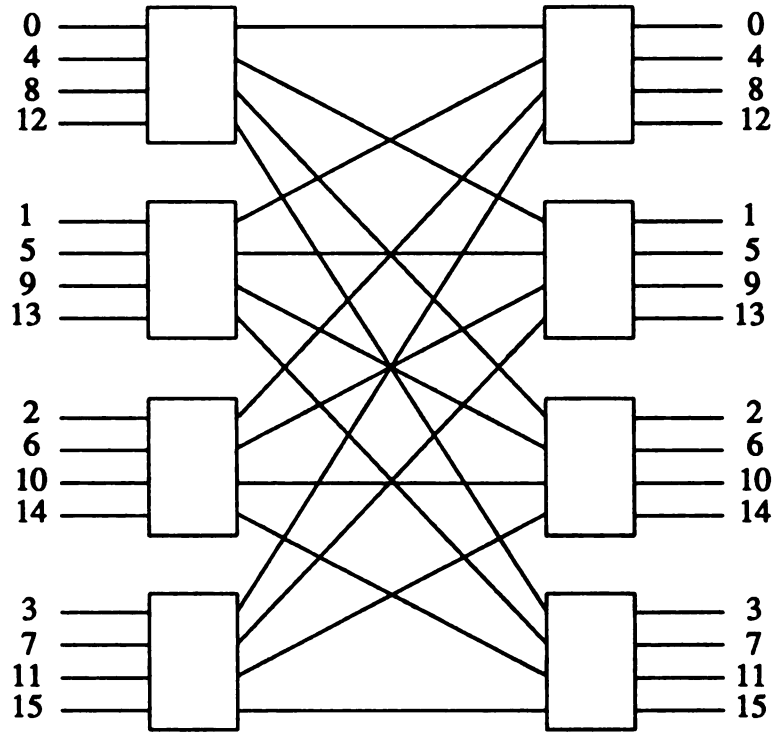


Figure 2.4 A 16 node BBN Butterfly interconnection network

The following issues have to be considered when designing or selecting a point-to-point interconnection topology:

1. *Diameter* of the network. This is defined to be the maximum distance between all pairs of nodes in the system. (The distance between two nodes is the length in number of edges of a shortest path between two nodes. See formal definition in Section 4.1.) A network with a smaller diameter has smaller worst case delay caused by the message handling at intermediate nodes. The average communication distance also depends mainly on the network diameter.
2. *Degree* of the network. This is defined to be the number of connection links per node. A large degree of network can provide a number of alternative paths between pairs of nodes. However, a large degree also implies more expensive hardware cost.
3. *Regularity*. A regular network is usually easy to construct and expand, and may provide an easy routing mechanism.
4. *Routing*. The ease of message routing is essential to the system performance and is greatly affected by the network topology.
5. *Expansion*. This is measured as the ease of adding new nodes to the network. A network should be incrementally expandable. That is, the size of the network can be increased by easily adding more nodes to it.
6. *Robustness*. A reliable operation of the interconnection network is very important to the overall system performance. When the number of processors and communication links increases, the probability that some components fail increases proportionally. Thus, it is desirable to design a network which can keep full connection capacity with graceful degradation when existing certain faults.

Some of the above issues are related; however, some may conflict. Thus, a trade-off must be made between them for each individual design.

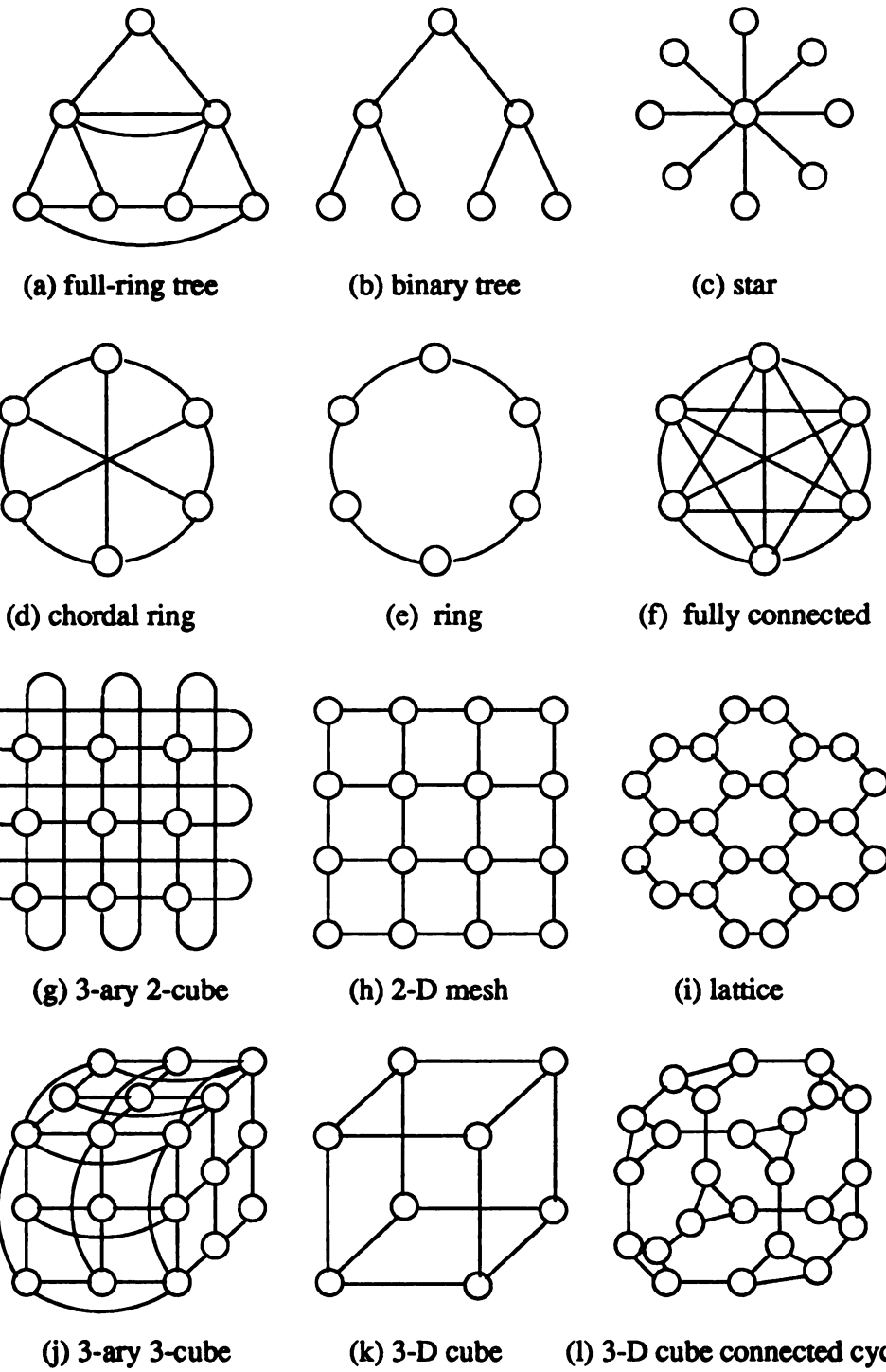


Figure 2.5 Examples of point-to-point interconnection networks

Various point-to-point topologies have been studied. Figure 2.5 shows some popular point-to-point topologies. Typical examples are ring, tree, star, mesh (2-D and higher dimension), lattice, completely connected, hypercube, etc.

Among those topologies, the hypercube has attracted most attention in the recent years. The topological properties of hypercube will be discussed in detail in Section 2.3. Another topology, the binary tree (and its augmented variations) has also been extensively studied. Tree structure is especially good for AI application. Tree topology has a small degree, thus, low hardware cost. However, it has poor robustness and may have a congestion problem toward the root. With some addition of links, the augmented binary trees can somewhat alleviate the above problems. A typical example is the *full ring binary tree* [DePa78], as shown in Figure 2.5(a), which is a regular binary tree with all nodes in the same level connected as a ring.

A rather general topology is the k-ary n-cube (torus) consisting of k^n nodes. It is a topology with n dimensions, and k nodes at each dimension. The k nodes in a dimension are connected as a ring. Figure 2.5(g) is a 3-ary 2-cube, and Figure 2.5(j) is a 3-ary 3-cube where some nodes and edges are not shown. The topology of a k-ary n-cube is equivalent to a corresponding multistage network whose switches are replaced by processor/memory pairs [GKLS83]. Binary cube is a special case of k-ary n-cube (k=2). A 2-D mesh can also be considered as a k-ary 2-cube.

Two more cube-related topologies are the *cube-connected cycle* (CCC) [PrVu81] and *hypernet* [HwGh87]. Figure 2.5(l) is an example of a cube-connected cycle. The major motivation of CCC is to keep the node degree constant, so that it is easier to expand a CCC than a hypercube. Hypernet is a class of hierarchical networks for modular construction of very large size parallel systems by providing structured building blocks for nodes and links.

While numerous interconnection topologies have been proposed, the binary cube is still the most popular one.

2.3 TOPOLOGICAL PROPERTIES OF HYPERCUBE

Hypercube structure has been the subject of many research projects and has been studied from different perspectives. As a result, many topological properties of hypercube have been discovered [Fold77, BaPa86, BrSc86, HMSC86, KrVC86, SaSc85a, SuBa77, TPPL85].

The properties of an n dimensional hypercube graph (Q_n) can be briefly summarized as follows:

(1) A Q_n has $N=2^n$ nodes, with addresses from 0 to 2^n-1 in binary form $(b_{n-1}b_{n-2}\cdots b_0)$, and $n2^{n-1}$ edges.

(2) There is an *edge* (or *link*) between two nodes if and only if the binary addresses of the two nodes differ at exactly one bit position. If the bit position is position i, then the edge is said to be at i-th *dimension*, or be the i-th dimensional edge. Thus, a Q_n is *n-regular* (each node has exactly n edges connecting to n neighboring nodes).

(3) A Q_n can be recursively constructed by combining two Q_{n-1}'s. Let $(b_{n-2}\cdots b_0)$ be an address in Q_{n-1}, then there is a link between two corresponding nodes $(0b_{n-2}\cdots b_0)$ and $(1b_{n-2}\cdots b_0)$.

(4) A Q_n can be split into two Q_{n-1}'s in n different ways. Namely, all nodes having value 1 at bit position i and all edges incident to these nodes are put into one subcube; all other nodes and their incident edges are put into another subcube, for $0\le i\le n-1$. Moreover, it can be split into 2^k Q_{n-k}'s $(1\le k\le n-1)$.

(5) The *distance* between any two nodes is the number of bit positions by which the two binary addresses differ. That is, the *Hamming* distance of the two binary numbers.

(6) The diameter of a Q_n (the maximum distance between all pairs of nodes in a Q_n) is n.

(7) There are $d!$ distinct paths of length d between two nodes of distance d in a Q_n.

(8) A set of n *node–disjoint* paths between any two nodes in a Q_n can be constructed. If the distance between the two nodes is d and all paths been selected are as short as possible, then d of the n paths are of length d, and the remaining $n-d$ paths are of length $d+2$. The construction of the set of paths is not unique if $d>2$.

(9) Q_n is a bipartite graph. Thus, there are no cycles of odd lengths in it.

(10) A ring of length $p=2q$ can be mapped into Q_n when $2 \le q \le 2^{n-1}$.

(11) A binary tree of height $n-2$ (i.e. $n-1$ levels) can be mapped into Q_n (a single node is a 1 level tree of height 0).

(12) A d-dimensional mesh $(m_1 \times m_2 \times \cdots \times m_d)$ can be mapped into Q_n if $\sum_{i=1}^{d} \log_2 \lceil m_i \rceil \le n$.

One of the most important advantages of hypercube topology is that it is a superset of many other topologies, such as ring, two or higher dimensional mesh, tree, etc. As we know, different application problems are best executed in systems with different architectures. Many numerical algorithms can be naturally decomposed such that the communication pattern required by the job tasks matches the hypercube topology. Some popular topologies, such as mesh, tree, FFT(Fast Fourier Transform), are used in a great many scientific applications. For example, matrix operations (matrix-vector multiplication, convolution, etc.) are suited to be solved in mesh structured multiprocessors, while search algorithms, linear recurrences are best executed in tree structured multiprocessors. The FFT, which is one of the most common used computational algorithms in almost all areas of scientific computation [BaPa86], can be perfectly mapped into hypercube topology. Because of the wide range of embeddability of hypercube topology, problems of those structures can be partitioned and mapped into a hypercube very easily and be solved efficiently.

The diameter of a hypercube is relatively small, compared with other topologies such as tree and 2-D mesh. It grows logarithmically with the number of nodes. This

makes it possible to construct a system consisting of thousands of processors.

Each node in a hypercube is topologically identical. There are no corner-versus-edge, or root-versus-leaf nodes, as are found in regular grids and trees. Because of the symmetry, there is no special congestion point as would happen in the tree topology. This symmetry also makes message routing in the hypercube relatively easy, which will be discussed in detail later. The symmetry is also a useful feature for dynamic reconfiguration of the system. Hypercube topology provides multiple paths between any pair of nodes in the network. This makes the hypercube topology inherently fault-tolerant.

All these properties make hypercube multiprocessors very versatile and suitable for a wide range of computational applications [FoOt84, BaPa86]. As an impressive example, in an 10-dimensional Ncube/ten hypercube, for some specially selected problems with well tuned algorithms, a speedup of 1009 to 1020 over uniprocessor has been achieved [GuMB88].

2.4 REVIEW OF HYPERCUBE MULTIPROCESSORS

Since the first hypercube multiprocessor, the COSMIC Cube, was demonstrated in Caltech [Seit85], many other hypercube multiprocessors have been made commercially available. Examples of hypercube multiprocessors include Intel's iPSC (up to 128 processors), Ncube's hypercube (up to 1024 processors), Ametek's S/14 (up to 256 processors), FPS's T series (up to 2^{14} processors), and JPL's MARK III [Amet86, GrRe86, HMSC86, SASL85, PTLP85]. In the rest of this section, we briefly outline a typical DMMP, the NCUBE. Then we give a comparison of the above mentioned systems.

2.4.1 Characteristics of a typical hypercube - the NCUBE

As an example of DMMPs, we briefly review the characteristics of the NCUBE hypercube multiprocessor [HMSC86].

System architecture: Modularity and scalability are two architectural features of DMMPs. An NCUBE hypercube multiprocessor can be configured to have various size by combining different number of processor boards and I/O boards to form a system. A maximum size NCUBE/ten has 1024 nodes, which consists of 16 processor boards and 8 I/O boards. A processor board contains 64 nodes. The I/O boards include host boards, graphics boards and open systems boards that can be configured for custom design. An I/O board contains 16 I/O processors, each having connections to 8 nodes. Thus, an I/O board has connections to a 128-node subcube. At least one of the I/O boards must be a host board.

Host: An NCUBE has one or more (up to eight) host boards. A host board has an Intel 80286 to run the Axis operating system. The board has 4Mbyte memory shared by the host processor and other processors on the board. It supports peripherals such as terminals, disk drives, tape drives, and network controllers, etc. The host(s) provide the primary user interface and perform functions such as editing, debugging, program and data downloading, performance monitoring, file maintenance, etc.

Node: A unique feature of the NCUBE is its compactness. Each node requires only one processor chip and six memory chips. The processor chip is custom designed, which contains a vax-like 32-bit processor, built-in floating-point support, memory controller support, and eleven bidirectional communication channels. Ten of them are connected to 10 neighboring nodes, the other one is connected to a I/O processor. Each node has 128K or 512Kbyte memory. All eleven channels can be active simultaneously. At 10 MHz clock, a node executes non-arithmetic instructions at 2MIPS or single precision floating point operations at 0.5MFLOPS.

Communication: Communication between nodes is done by means of asynchronous DMA operations over the full duplex bidirectional links between neighboring nodes. For a 10MHz clock, the data transfer rate is about 1Mbyte/second on each direction (700Kbyte/second for a 7MHz clock). Because of the DMA, the processor needs

only to initiate a send or a receive operation for each communication request. Communication between the hypercube nodes and external devices are handled by the I/O processors through the eleventh channel. Each I/O processor has a 128K (or 512K) RAM which occupies a fixed slot of memory space in the host's 4Mbyte memory. To perform an input operation, the data are first sent to the host's memory, then are transferred to target nodes through DMA channels. The output operations are performed in a similar way.

System software: The host processor (80286) in each host board runs Axis operating system, a variant of Unix, which is compatible with both Unix System V and 4.3 BSD. Since an NCUBE may have more than one host board, each having its own file system. A potential data inconsistency and conflict may exist. Axis provides the capability to manage the multiple file systems as one unified, distributed file system. It also supports hypercube partitioning, so that a user can allocate a subcube of a proper size for his particular application. Both FORTRAN-77 and C are supported by the Axis. Each node runs a small operating system called Vertex, whose main function is to support message-passing between neighboring nodes.

A system of 1024 nodes may have the speed of 2000 MIPS and 500FLOPS on single precision or 300MFLOPS on double precision operations.

2.4.2 Comparison of hypercube multiprocessors

The characteristics of several hypercube multiprocessors are compared in Table 2.1 [ShFi87, ReFu87, Duni87, PTLP85]. Both iPSC and Ametek S/14 use standard microprocessors. While FPT T series uses Inmos Transputer [GuHS86]. NCUBE developed its own custom designed VLSI chips. All systems have a host processor which is used for program downloading/uploading and data transfer between the nodes and external devices. In the table, rows 5 to 8 are the memory size at each node, the number of communication channels connected to each node, and the data transmission rate of each channel (in bit/second), respectively. Lines 11 and 12 are the performance

measurement of the entire system having maximum number of nodes for non-arithmetic operations and double precision floating point operations, respectively.

Table 2.1 Hypercube system characteristics

No.	Item	Ametek S/14	FPS T-Series	Intel iPSC	Mark-III	NCUBE
1	# of PE's	16-256	8-4096	32-128	32-1024	64-1024
2	(clock) Processor	8 MHz 80286	20 MHz Transputer	8 MHz 80286	16 MHz 68020	7 or 10 MHz Custom
3	(clock) Floating Point	8 MHz 80287	included	6 MHz 80287	16 MHz 68881	included
4	I/O Processor	80186	included	80186	68020	DMA controller
5	Memory (Byte)	1M	1M	0.5-4.5M	4M	128-512K
6	Channels/Node	8	4×4	7	8	11
7	Bandwidth(bps)	3M	20M	10M	13.5M	10M
8	Host	VAX	MicroVax	286/310	system 19	80286
9	Host OS	Unix/Ultrix	VMS	Xenix	CrOS	Axis
10	Node OS	Xos/Mars	Occam	iPSC/OS	Mercury	Vertex
11	MIPS	200	30000	100	2000	2000
12	MFLOPS	12	65000	8	2000	300

CHAPTER 3

ISSUES ON INTERPROCESSOR COMMUNICATION

In a distributed memory message-passing multiprocessor system, processors do not have shared memory. Message passing is the only means for interprocessor communication. If a node wants to send a message to a neighboring node, the message delivery is relatively simple. However, if a node wants to send a message to a distant node, the message has to traverse through some intermediate nodes. To send a message from a node to a number of other nodes, the situation becomes more complicated. The major problem in interprocessor communication is message routing, that is, to determine which path(s) should be used to deliver a message from the source node to some destination node(s). In this chapter, we first introduce a model for analyzing the message passing time between two nodes. Then, we discuss some fundamental issues in interprocessor communication, which are the keys to the performance of DMMPs.

3.1 A MODEL FOR INTERPROCESSOR COMMUNICATION

Let us consider passing a message from a source node u_s to a destination node u_d with distance $d(u_s,u_d)=h$ in a DMMP. Some benchmarking has been done for several hypercubes [ShFi87, GrRe86]. For communication between neighboring nodes (h=1), as depicted in Figure 3.1, it is assumed that the time (t) needed to pass a message of size S bytes, can be expressed as:

Figure 3.1 Communication between two neighboring nodes

$$t=t_l+t_cS \tag{3.1}$$

where t_l is the communication *latency*, that is, the overhead time caused by processor u_s to initiate the communication and by u_d to terminate the communication; t_c is the time needed to transmit one byte of data. After measuring the communication time for messages with different lengths, the parameters t_l and t_c can be determined by a least-squares fit. Table 3.1 shows the results for four first generation hypercube multiprocessors [GrRe86, ShFi87].

Table 3.1 One hop communication comparison

	Mark-III	Intel iPSC	Ametek S/14	Ncube/ten
Latency: t_l (μs)	95	1700	550	384
t_c (μs/Byte)	0.563	2.83	9.53	2.6

For a more general communication, that is, communication between any nodes ($h \geq 1$), single or multiple destinations, we propose the following model. Figure 3.2 depicts a path selected for sending a message from u_s to u_d, which could be a one destination communication, or a path from the source to one of several destinations in a multiple-destination communication.

As discussed in Section 2.1, we assumed that each node is associated with a router to handle communication tasks. Let $\tau_s=\tau_{ss}+\tau_{sd}$ be the node-to-node communication

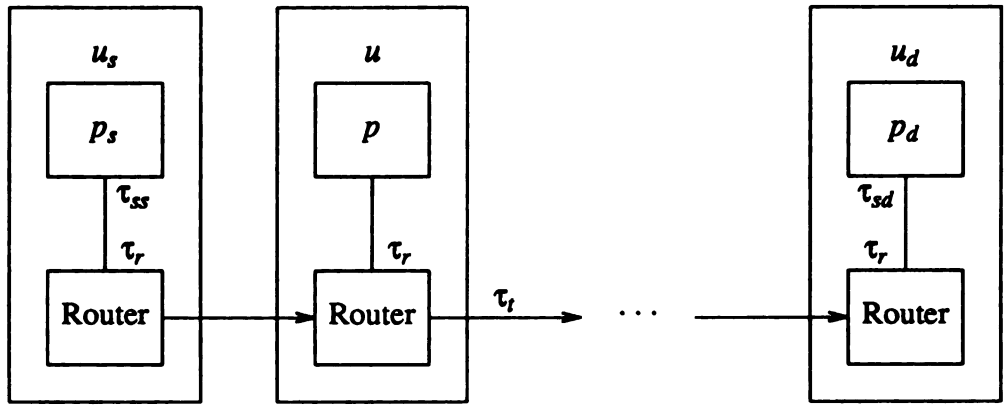


Figure 3.2 One-to-one message passing model (h hops)

start-terminate time, where τ_{ss} is the start-up time spent at the source node, that is, the time interval from the moment when the source processor (p_s) issues a *send* command until the moment when the router begins to receive the message; and τ_{sd} is the time spent at the destination node to terminate the communication, that is, the time interval from the moment when the router begins to send out the message until the moment when the destination nodal processor (p_d) receives the entire message. Let τ_d be the delay time from the moment when a sending router starts transmitting the message till the receiving router (at a neighboring node) starts to make routing decision. In order to simplify the discussion, the delay time τ_d at the source node (the message transfer time from the source processor to its router) is treated the same as that between two routers. Parameter τ_d is dependent on the size of the message, the bandwidth of the link, and the message forwarding scheme, which will be discussed in detail in Section 3.5. Finally, τ_p is the processing time required to make a routing decision at a router. As shown in Figure 3.2, for an h-hop message passing, h+1 routers are involved. Thus, the time (t) required to send a message to a node at h-hop away can be expressed in Eq. (3.2).

$$t = \tau_s + (h+1)\tau_p + (h+1)\tau_d. \tag{3.2}$$

In the remaining sections, we will discuss different communication mechanisms that affect the communication time.

3.2 THREE TYPES OF INTERPROCESSOR COMMUNICATIONS

At the system level, depending on the number of destinations, interprocessor communications can be classified into three types: unicast, broadcast and multicast.

Unicast (One-to-one) communication is the sending of a message from a source node to one destination node. It is directly supported by all DMMPs. Unicast in a hypercube multiprocessor can be easily implemented based on the Hamming code. If the distance between the source and the destination is d (i.e., their binary addresses differ at exactly d bit positions), a total number of $d!$ distinct (shortest) paths, all having $d-1$ intermediate nodes, can be constructed between the two nodes by changing, one at a time, the values at the d bit positions in different orders. Using any one of the $d!$ paths, the message will traverse the same number of links.

Broadcast (one-to-all) is a type of information exchange in which a source node wishes to send a message to all other nodes in the system as quickly as possible. A frequently used approach can be described as follows. In the first step, the source sends out n copies of the message to all its n neighboring nodes. Subsequently, these nodes duplicate and send out the message to those neighboring nodes which have not yet received the message. The process continues until all nodes receive the message. Broadcast in hypercube has been studied in [SuBa77, SaSc85b, HoJo86, BrSc86]. It has been shown that broadcast in an n-cube can be done in n time steps even if each node sends a message to no more than one neighbor during each time step.

In *Multicast (one-to-many)* communication, a node wants to send the same message to k other nodes ($1<k<2^n-1$). To send messages to the right destinations at the right time is very important in many applications, since a node may have to await a

message from some other nodes before continuing its computation. Therefore, it is desirable that each individual destination receives the message in the fewest possible time steps. This implies that each destination should receive the message through a shortest path between the source and that destination. Moreover, the number of intermediate nodes required to deliver the source message to all destinations should be as few as possible. This is because that number is a direct measure of traffic created by the multicast communication in the network. Multicast communication has been studied in [LaEN88a, LaEN88b].

3.3 CENTRALIZED VS. DISTRIBUTED ROUTING

An essential issue in implementing a routing scheme is whether the source node should determine all the intermediate nodes (and related links) for message delivery or the source node and each *forward* node (a node involved in further message forwarding, which may or may not be a destination node, see a formal definition in Section 4.2) should determine only its neighboring node(s) which should be involved in the message delivery. The first method is referred to as *centralized* routing; whereas, the second method is known as *distributed* routing.

The main disadvantage of centralized routing is that the addresses of all intermediate nodes must be tagged with the message. This will create extra communication overhead. Especially in the case of multi-destination message routing, it is unrealistic to specify a particular path for each of the destinations, because it will result in an undesirable amount of message overhead.

Distributed routing avoids this problem by carrying only the destination addresses in the message header. For a regular structured interconnection topology like hypercube, it is easy to make routing decisions based on the source and destination addresses only.

Another disadvantage of centralized routing is the fault-tolerant consideration. In the case when some nodes or links fail, by centralized routing, a source node must have global fault information in order to find a feasible path. By distributed routing, however, the routing decision is made based on local fault information only. Thus, it is more flexible and easier to implement. Therefore, in this study, we consider distributed routing only. Routing in faulty hypercubes will be discussed in detail in Chapter 7.

3.4 PACKET VS. CIRCUIT SWITCHING

When a message delivery between non-neighboring nodes occurs, the message has to be forwarded through some intermediate nodes. Two transport mechanisms are currently used: *circuit switching* and *packet switching*.

Circuit switching is similar to the communication mechanism in traditional telephone networks. Using circuit switching, a physical communication path between the source and the destination has to be established at first. Then, the source can send out the message to the destination.

Packet switching is widely used in computer networks. By packet switching, however, no physical path is established before the starting of a communication. A message is decomposed into packets and then the communication is carried out in the form of packets. The source determines its output link(s), sends out the message to the neighboring node(s). Then, each of the nodes which receives the message will decide its output link(s) for further forwarding, and so on.

The major advantage of circuit switching is that routing overhead is paid only once at the circuit set-up time. If each intermediate node can provide physical electrical connection, then no buffers are needed. After the physical path is established, the messages can traverse through the path with very little delay. However, once a physical path is established, none of the links along the path can be used (or shared) by another message delivery. If the messages are short, and the communication is bursty, then the

bandwidth of the communication path will be wasted.

The Packet switching mechanism makes efficient use of the bandwidth of the communication links since it requests for one link at a time and releases the link immediately after it is used. However, it is required to make routing decisions for each message packet. Also, a buffer is needed at each node to temporarily store the message. The efficiency of the communication depends on the strategy for making the routing decisions at the source and intermediate nodes as well as the size of the packet. If the packet size is small and the message size is relatively large, then several packets may be required to send one message. The routing overhead will be greater and message reassembly may be required at the destination node depending on whether deterministic routing or adaptive routing is used.

Packet switching mechanism can be further divided into store-and-forward and virtual cut-through methods, which are discussed in the next section.

3.5 STORE-AND-FORWARD VS. VIRTUAL CUT-THROUGH FORWARDING

There are two switching methodologies to handle the switching tasks at intermediate nodes: *store-and-forward* [Tane81] and *virtual cut-through* [KeKl79]. In all first generation DMMPs, the store-and-forward approach is used, in which a node receives the entire message competely, and then further forwards the message. By virtual cut-through approach, however, a node decides message forwarding link(s) right after the destination addresses in the message header are received. The node then immediately sends out the message to the selected link(s). The data part of the incoming message is buffered only when the selected output link is unavailable. Obviously, the virtual cut-through approach provides faster forwarding service. Thus, virtual cut-through approach is preferred to the store-and-forward approach.

In this study, we use a *relay* approach for message forwarding, which is very similar to the virtual cut-through approach used in computer networks [KeKl79]. The

difference between our relay approach and the virtual cut-through approach is that the latter requires the entire message be received completely if the selected link is blocked. In relay method, however, as soon as the link is available, the message will be sent out.

Let us take a close examination of the h-hop message-passing time expressed in Eq. (3.2) under the assumption of store-and-forward approach and relay approach, respectively.

In store-and-forward approach, Eq. (3.2) can be written as:

$$t = \tau_s + (h+1)\tau_p + (h+1)\tau_t S \tag{3.3}$$

where

τ_s: communication start-terminate time;

τ_p: the processing time required to make routing decision at each router;

τ_t: data transmission rate (*byte /second*) of the communication link;

S: the size of a message (or a packet) in *byte*.

Note that, since the entire message has to be received and stored at a receiving router, and then the router can start to make routing decision, that means a router has to "wait" for $\tau_t S$ time, which corresponds to the delay time τ_d in Eq. (3.2). This results in the last term, $(h+1)\tau_t S$ in Eq. (3.3). In order to ease our analysis, we made an approximation in the above model that to transfer data from nodal processor to the router in the source node takes the same time as that between routers at two neighboring nodes. Comparing Eq. (3.3) with Eq. (3.1), it can be easily seen that the latency $t_l = \tau_s + 2\tau_p$, and $t_c = 2\tau_t$ is the time charged to each byte of data been transferred.

If we assume the message size S is a fixed value, as in the packet switching case, then the time t is a function of the distance h. Equation 3.3 can be rewritten as:

$$t = \tau_s + (h+1)(\tau_p + \tau_t S) \tag{3.4}$$

We observe that the term τ_s does not depend on the number of hops, h, in the communication. As a result, the term $\tau_p + \tau_t S$ can be used to approximate the value as a *time step*,

the time needed to transfer a message (of S byte) from a node to one of its neighboring node, which consists of two parts: the processing time τ_p and the waiting time $\tau_t S$.

Similarly, for the case of relay method, the time required is:

$$t = \tau_s + (h+1)\tau'_p + (h+1)\tau_t S' + \tau_t (S - S') \tag{3.5}$$

where

τ'_p: the processing time required at a router. If a hardware router is used, then $\tau'_p \ll \tau_p$ should hold; otherwise, $\tau'_p = \tau_p$.

S': the size in byte of the address part in the message.

Note that $\tau_t S'$ is the time each router has to "wait" to start processing, and $\tau_t(S-S')t$ counts the time needed to transfer the remaining portion of the message through the path. Similar to Eq. 3.4, we can rewrite Eq. 3.5 as follows:

$$t = (\tau_s + \tau_t (S - S')) + (h+1)(\tau'_p + \tau_t S') \tag{3.6}$$

As before, since the term $(\tau_s + \tau_t(S-S'))$ does not depend on the number of hops, we use $\tau'_p + \tau_t S'$ to approximate the value of a time step for the relay method, which also consists of two parts: τ'_p and $\tau_t S'$. Note that $\tau'_p \ll \tau_p$ and $S' \ll S$. Thus, $\tau'_p + \tau_t S' \ll \tau_p + \tau_t S$, which implies the hardware implementation of relay method is much faster than the software implementation of store-and-forward method.

In the above discussion, τ_t is determined by the bandwidth of communication links, τ_p or τ'_p is dependent upon the efficiency of the routing algorithm and on the architecture and technology of the router. In first generation hypercube multiprocessors, however, no dedicated router has been designed. In Intel iPSC, the function of the router is performed by the local processor and by the Ethernet communication coprocessor. In NCUBE hypercube, a DMA controller in each node handles interprocessor communication. However, it is not a dedicated hardware router. Nodal processor still needs to participate in interprocessor communication even in an intermediate node.

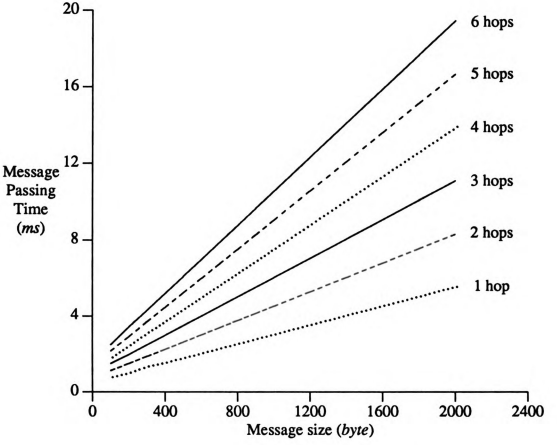


Figure 3.3 Message-passing time in a 64-node NCUBE

As an example of store-and-forward routing scheme, we measured the message-passing time for different message sizes and various distances in a 64-node NCUBE hypercube with a 7 MHz clock rate. The results are shown in Figure 3.3.

After using the least squares fit method, the parameters in Eq. (3.4) are determined as follows: τ_s=13.5 μsec, τ_p=232 μsec, τ_t=1.3 $\mu sec/byte$. The relatively small value of τ_s, compared with τ_p, indicates that the source and destination nodes can be treated the same way as intermediate nodes from the point of view of estimating communication overhead. In the case of one hop message-passing, using the model of Eq. (3.1), the two parameters are measured as: t_l=515μsec and t_c=2.5$\mu sec/byte$. Since the clock rate is 7 MHz, the theoretically estimated data transmission rate on the links is 1.4 $\mu sec/byte$. As can be seen, the measured value of 1.3$\mu sec/byte$ is very close to the estimated rate. Also, applying the model of Eq. (3.4) to the one hop case, they match reasonably well ($t_l \simeq \tau_s+2\tau_p$, and $t_c \simeq 2\tau_t$).

3.6 ADAPTIVE VS. NON-ADAPTIVE ROUTING

Based on the relationship between routing procedure and the traffic condition in the network, packet switching scheme can be classified as *adaptive* routing and *non-adaptive (deterministic)* routing.

With non-adaptive routing, the path from a source to each destination is determined *a priori*, that is, solely determined by the source and destination addresses. It is independent of the traffic variation in the network. In a DMMP system, the network usually provides multiple paths between the source and destination. However, only one of the paths is regularly used. The alternative paths are used only when the failures of some components make the "regular" path unfeasible and the routing algorithms have fault-tolerant ability.

By adaptive routing strategy, however, the path from a source to a destination is not fixed. It may vary from time to time, even the network is fault-free. The source and

each intermediate node along the path will select a seemly optimal output communication link based on its knowledge of global traffic condition at that particular moment.

Both strategies have their advantages and disadvantages. Theoretically, adaptive routing has the advantage of being able to evenly distribute the traffic load over the communication links, and consequently provide a better interprocessor communication service. However, as indicated in [ChBN81], the time required to update the global traffic information and the traffic overhead caused by updating the information makes it less attractive. Especially in a multiprocessor environment, since the communication links provide very fast data transmission rate, it is very difficult to accurately reflect the dynamic traffic condition in the network. Also, the time overhead needed to calculate the "optimal" output link is undesirable.

The deterministic strategy, on the other hand, is simple and easy to implement. In a multiprocessor system, if each node has a hardware router and the communication links have a high bandwidth, the deterministic routing is more attractive. In fact, all existing DMMPs adopt non-adaptive routing strategy. Therefore, only non-adaptive routing is considered in this study.

3.7 SOFTWARE VS. HARDWARE IMPLEMENTATION

In computer networks, communication functions are traditionally implemented by software approaches. This is also true in the first generation DMMPs. In this study, software implementation refers to the approaches where routing is made through subroutine calls by nodal processor (as in iPSC), or dedicated communication co-processor (as in Ametek), or I/O processor (as in NCUBE). By hardware implementation, we mean a hardware device dedicated to routing purpose is used at each node. The device is a hardware implementation of communication algorithms.

The store-and-forward method used in first generation DMMPs is already slow; the software implementation of the algorithms makes the situation even worse. The result is

a very poor communication/computation ratio as shown in Table 1.1. This becomes a severe weakness of the first generation DMMPs.

Through hardware implementation, a dedicated hardware routing device is attached to each nodal processor. The nodal processor no longer participates in the routing activities unless the node is a source or a destination. Furthermore, hardware routing device can implement a better message forwarding scheme to greatly reduce the time delay introduced at each intermediate node, thus, significantly reduce the overall communication time (two to three orders of magnitude smaller). For example, in the second generation Intel hypercube iPSC/2, by using a hardware routing device, only a few μsec is needed at each intermediate node to establish a physical path [iPSC88], and approximately 350 μsec is needed for a message transmit and acknowledgement receive. As mentioned before, the communication latency time is 1.7 *msec* in its first generation iPSC machines.

Hardware router for unicast has been reported in [DaSe86], in which the virtual cut-through packet switching method is used, and in [iPSC88], in which the circuit switching method is used. Hardware implementation for broadcast communication has also been studied [Seit87]. A hardware router which directly supports all three types of interprocessor communication is reported in [LaNE88].

CHAPTER 4

OPTIMAL MULTICAST TREE (OMT) — A MODEL FOR INTERPROCESSOR COMMUNICATION

We have discussed some fundamental issues about interprocessor communication in last chapter. In this chapter, some useful notation and definitions are introduced in Section 4.1. Then, in Section 4.2, a graph theoretical model – the Optimal Multicast Tree (OMT) is presented for the multicast communication problem. The OMT model is general enough to cover all three types of communications. Following the OMT model, in Section 4.3, the unicast and broadcast communication is studied. Section 4.4 investigates some insightful examples of multicast problem. The general multicast problem will be studied in Chapter 5.

4.1 GRAPH THEORETICAL NOTATION AND DEFINITIONS

We will closely follow the graph theoretical terminology and the notation of [Hara72]. Terms not defined here can be found in that book. Let $G(V,E)$ be a *graph*, with the *node set* $V(G)=V$ and *edge set* $E(G)=E$. We use *edge* and *link* interchangeably. When G is known from the context, the sets $V(G)$ and $E(G)$ will be referred to by V and E, respectively. If an edge $e=(u,v)\in E$, then nodes u and v are said to be *neighboring nodes* (or *neighbors*) and the edge e is said to be *incident* to these nodes. The *degree*, $deg_G(v)$, of a node $v\in V$ is equal to the number of edges in G which are incident to v.

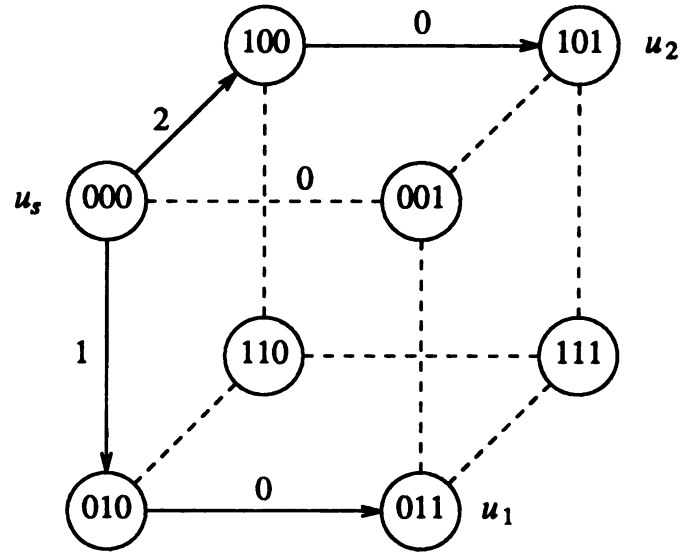
A path is an alternating sequence of nodes and edges, beginning and ending with nodes, in which all the nodes (and thus all the edges) are distinct. A path p from node u_0 to node u_d can be represented by an ordered sequence of nodes $(u_0,u_1,\cdots,u_j,\cdots,u_{d-1},u_d)$. Or, alternatively, by a sequence of edges following node u_0 [LeHa88]: $u_0|(l_0,l_1,\cdots,l_{d-1})$, where edge $l_j=(u_j,u_{j+1})$, $0\le j\le d-1$. The length of a path p is measured by the number of edges contained in the path. Therefore, the above path has length d. A path is *shortest* or *minimal*, if there are no shorter paths between the two given nodes.

A graph is said to be *connected* if every pair of its nodes are joined by a path. A *tree* is a connected graph which contains no *cycles*. A graph $H(V,E)$ is a *subgraph* of another graph $G(V,E)$, if $V(H)\subseteq V(G)$ and $E(H)\subseteq E(G)$. When $V(H)=V(G)$, H is called a *spanning* subgraph. A subgraph which is a tree is referred to as a *subtree*. The *distance*, $d_G(u, v)$, between a pair of nodes u and v in G, is equal to the length (in number of edges) of a shortest path joining u and v.

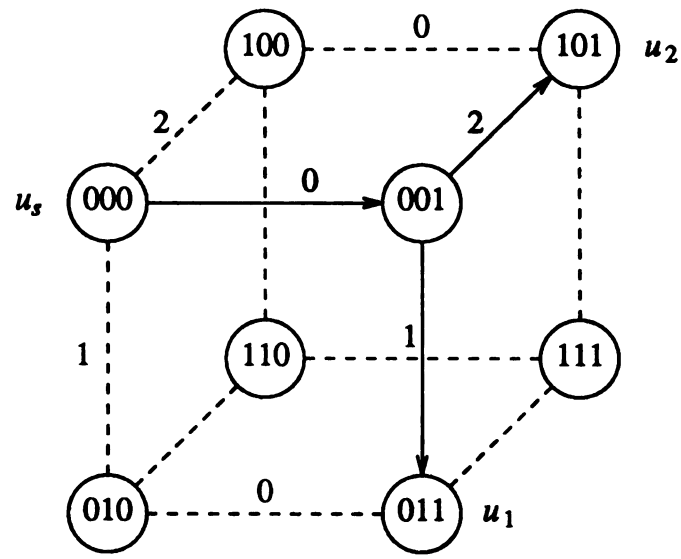
4.2 THE OPTIMAL MULTICAST TREE MODEL

Multicast communication is highly demanded in many application areas, for example, in simulation for electrical engineering, modeling for mechanical engineering, simulation for computer networks, etc. Consider gate level simulation in electrical engineering. The output of a gate is usually connected to the inputs of several other gates, which is a typical example of multicast. In first generation hypercube multiprocessors, multicast is basically done by issuing multiple one-to-one communication, which is not an efficient approach.

Let us examine a simple example of communication from a source node to two destination nodes in a 3-cube. As show in Figure 4.1, suppose source node 000 (u_s) wants to send a message to both 011 (u_1) and 101 (u_2). In Figure 4.1 (a), the source sends out two messages separately to two *intermediate* nodes 010 and 100. Then, the



(a) Two separate paths for two destinations



(b) Two paths share a common link

Figure 4.1 A multicast example in 3-cube

two nodes pass the message to u_1 and u_2, respectively. As we can see, each message traverses 2 links. The total number of links involved in the communication is four. In Figure (b), however, the source first sends a message to node 001 only, then the message is duplicated at that node and sent out to u_1 and u_2. If we assume that in both cases the time required for an intermediate node to forward a message is the same, then the two destinations will receive the message in the same time interval (two time steps). The total number of links involved is three in the latter case, however, it is four in the previous case.

The two generic parameters which can be used as measures of communication efficiency are *time* and *traffic*. They are formally defined as follows. Parameter *time* is quantified using *time steps*, where a time step is the actual time needed to send a unit of information (message) from a node to one of its neighboring nodes. This time is assumed to be constant for all pairs of neighboring nodes. Parameter *traffic* is quantified by the number of messages traversing over the communication links which are used to deliver the source message to its destination(s). A unit of traffic is measured as a message traverses over a link. Note that the number of time steps required to inform a destination is at least equal to the distance from the source to that destination. Also, for each destination, at least one distinct communication link will be used. That is, at least one unit of traffic will be created. Hence, the number of destinations gives a lower bound for the amount of traffic required to complete the communication. In general, the relationship between the two parameters are quite complicated. Usually, they are not totally independent.

In order to make the situation more clear, let us consider a multi-destination message passing example in a 2-D mesh environment, as shown in Figure 4.2. Assume node u_s (1) wants to send a message to u_1 (9), u_2 (10), u_3 (13), and u_4 (14). There are many ways to implement the multicast. Two possible configurations are depicted in Figure 4.2(b) and (c). In Figure 4.2(b), u_s first sends the message to node 6, node 6 then

forwards the message to node 11, at node 11, the message is replicated and sent to nodes 10 and 12, and so on. Eventually, u_1 receives the message after 6 steps, while u_2, u_3 and u_4 receive the message after 3, 4 and 5 steps, respectively. The traffic for this configuration is 7. While in Figure 4.2(c), the message is replicated at node 6. Destination u_1 receives the message in 4 steps in stead of 6 steps. The time steps for u_2, u_3 and u_4 to receive the message are the same as before. However, the total traffic increases from 7 to 8.

Clearly, it is desirable to develop a routing mechanism that completes communication while minimizing both time and traffic. However, as we can see, the two parameters may be in conflict. Minimizing one parameter may prevent minimizing the other. In multiprocessor environment, we want every processor to receive messages in the shortest possible time in order to reduce unnecessary "waiting". Thus, we consider the time step requirement a higher priority. Based on that, we then try to minimize the traffic.

A graph theoretical model for multicast communication in distributed memory multiprocessors is now formally defined as follows. Let $G(V, E)$ be a graph corresponding to the topology of interprocessor communication network of the distributed memory multiprocessor under consideration, and $D=\{u_0,u_1,\cdots,u_k\}$ be a subset of V. Node u_0 corresponds to the source node, and nodes $u_1,u_2,\cdots,u_k$ correspond to k destination nodes in a multicast. The multicast problem is the problem of finding a subtree $T(V, E)$ of $G(V, E)$, called the *Optimal Multicast Tree* (OMT), such that

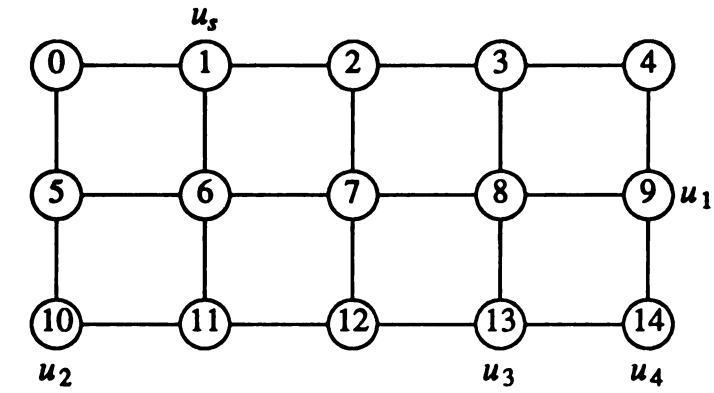
(a) $D \subseteq V(T)$,

(b) $d_T(u_0,u_i)=d_G(u_0,u_i)$, for $1 \le i \le k$, and

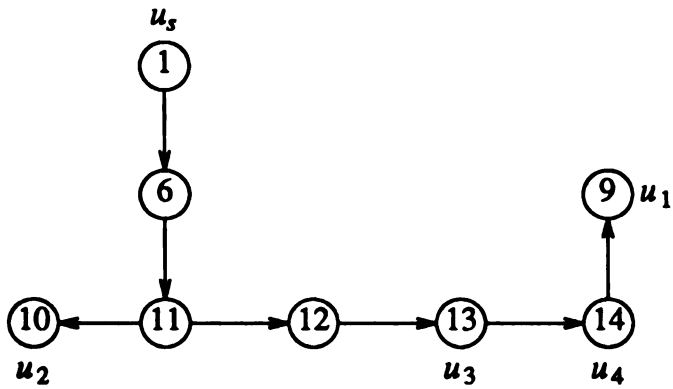
(c) $|E(T)|$ is as small as possible.

where $E(T)$ and $V(T)$ are the edge set and vertex set of tree $T(V,E)$, respectively.

A subtree of G which satisfies the conditions (a) and (b) above is referred to as a *multicast tree (MT)*. In a multicast tree, a *leaf* node is a destination node of degree one;

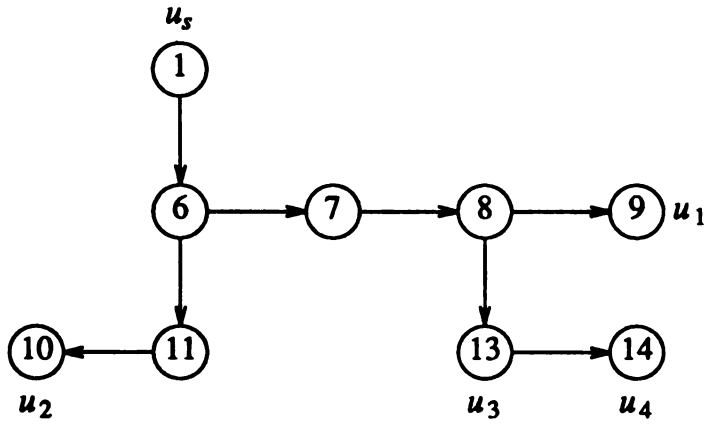


(a) A 2-D Mesh



(b)

Time: $u_1(6), u_2(3)$
$u_3(4), u_4(5)$
Total traffic: 7



(c)

Time: $u_1(4), u_2(3)$
$u_3(4), u_4(5)$
Total traffic: 8

Figure 4.2 A example of multicast in a 2-D mesh

all non-leaf nodes are referred to as *forward* nodes. Thus, an *intermediate* node refers to a non-destination forward node. In general, OMT may be not unique. Observing the model, we can see that condition (b) is to ensure the minimum time steps while condition (c) is to reduce the traffic with the implication that a message traverses any link no more than once.

Notice that, in a broad sense (when $1 \le k \le 2^n-1$) the *OMT* model actually refers to all three types of communication: unicast ($k=1$), broadcast ($k=2^n-1$), and multicast in a narrow sense ($1<k<2^n-1$). Thus, the model of the *OMT* problem is a general model for any interprocessor communication. In the following discussion, we usually use the name *multicast* in its narrow sense.

In the following discussion, we will concentrate our discussion on the *OMT* problem when $G(V,E)$ is the n-cube graph, since hypercube has become the major interconnection network for DMMPs in recent years. We first discuss unicast and broadcast in hypercube multiprocessors. Then, we will consider the third communication type — the multicast. A heuristic multicast algorithm which achieves condition (b) above and may only compromise condition (c) will be presented in Chapter 5.

4.3 UNICAST AND BROADCAST IN HYPERCUBE MULTIPROCESSORS

In the last section, we presented a general graph theoretical model for interprocessor communication. In this section, we first give a graph theoretical model for hypercube multiprocessor and introduce some notations for communication in hypercube environment. Then, we examine how an OMT can be constructed in the case of unicast and broadcast in hypercube multiprocessors.

An n-dimensional hypercube multiprocessor can be modeled as a graph $Q_n(V,E)$, with $|V|=N=2^n$, $|E|=n2^{n-1}$. Each node represents a processor (and its memory); each edge represents a communication link between a pair of processors. The 2^n nodes are distinctly addressed by n-bit binary numbers, $b_{(n-1)}b_{(n-2)}\cdots b_j\cdots b_0$. For each node

$u \in V(Q_n)$, let $a(u)$ and $||a(u)||$ denote the binary address of node u and the number of 1's in binary number $a(u)$, respectively. Also, let $\oplus$ denote the *bitwise exclusive or* (*XOR*) operation on binary numbers. Then, $e=(u,v) \in E(Q_n)$, if and only if $||a(u) \oplus a(v)||=1$. An edge connecting nodes u and v is said to be at *dimension* j or be the j-th dimensional edge if the addresses of u and v differ at bit position j only, i.e., $a(u) \oplus a(v)=2^j$. It is implied that $deg(u)=n$ for every node $u \in V(Q_n)$. That is, each node has links at n dimensions ranging from 0 (lowest dimension) to $n-1$ (highest dimension). Also, $d_{Q_n}(u,v)=||a(u) \oplus a(v)||$, for every pair of nodes $u,v \in V(Q_n)$.

In this study, we use $a(u_i)$ to represent the address of node u_i, in general. However, in some cases, we need to compare the addresses of two nodes. Thus, $a(u_i)$ and $a(u_j)$ are called the *actual addresses* of node u_i and u_j, respectively, while the *relative address* of node u_i with respect to node u_j, is defined as $r_j(u_i)=a(u_j) \oplus a(u_i)$.

In the following discussion, algorithms are to be executed at each node which receives a message. The node *originating* a message is called the *source* node and represented by u_s. Notation u_0 is used to represent any local node which is currently executing the algorithm. Thus, u_0 could mean the source node or any other node which received a message and is executing the algorithm. From the algorithm point of view, each such node is treated as a "source" node. When we have to distinguish between the real source node which *originates* the message and the local node which is currently *executing* the communication algorithm, we use u_s and u_0, respectively. Otherwise, we just use u_0 in our discussion.

For a binary number $B=b_{n-1} \cdots b_1 b_0$, we define function W as follows:

$$W_i(B) = \begin{cases} j & \text{if } j \text{ is the } i\text{-}th \text{ Right most bit position having value 1} \\ -1 & \text{if } b_j = 0 \ \forall \ 0 \le j \le n-1; \text{ or } B \text{ has less than } i \text{ bit positions having value 1} \end{cases}$$

where $0 \le i, j \le n-1$.

For example, $W_1(01001)=0$; $W_2(01001)=3$; $W_3(01001)=-1$; $W_1(00000)=-1$.

Using the above notation, we can now give the algorithms for unicast and broadcast communication. In Section 2.3, we summarized some attractive topological properties of hypercube. Due to those special properties, one-to-one and broadcast communication in hypercube can be done in a quite straightforward way.

Sullivan and Bashkow [SuBa77] presented an algorithm for message-passing from one node to another arbitrary node in the system, which now becomes a standard unicast algorithm for fault-free hypercube. The idea is that each receiving node first checks if the local address matches the destination address. If it does, then the message is sent to the processor. Otherwise, the algorithm finds the right most bit position in which the relative address of destination with respect to local address has value 1 and sends the message to that dimension.

```
Algorithm UNICAST.1:
begin
  (* find the relative destination address with respect to local address *)
  r0(ud) := a(u0)⊕a(ud);
  if (r0(ud)=0) then
      send message to local processor
  else
      send message to dimension W1(r0(ud));
end.
```

Figure 4.3 The "standard" unicast algorithm

Algorithm UNICAST.1 in Figure 4.3 shows the idea. The destination address is carried in the message header together with the data part (see Section 6.2 for a detail description of the message format). Note that the algorithm is first executed at the source node ($u_0=u_s$). As a result, the message is forwarded to a neighboring node determined by the algorithm. Then the algorithm is executed at that node again, and so on. It

is obvious that the path found by algorithm UNICAST.1 goes through dimension $W_i(a(u_s)\oplus a(u_d))$ at step $i+1$ ($0\le i\le d-1$), and eventually guides the message to the destination u_d in d steps. Let $r_s(u_d)=a(u_s)\oplus a(u_d)$. Then the path constructed by the algorithm can be represented by $a(u_s)\,|(W_1(r_s(u_d)),W_2(r_s(u_d)),\cdots W_{d-1}(r_s(u_d)))$.

A broadcast algorithm has also been presented in [SuBa77] and followed by other researchers [BrSc86, HoJo86, SaSc85b, Kats88]. Sullivan and Bashkow's algorithm works by sending a *weight* along with the data part of the message. The weight indicates how the receiving node should continue sending the message. Eventually, each node in the system will receive the broadcast message exactly once and in no later than n time steps. The algorithm is listed in Figure 4.4 as BROADCAST.1.

```
Algorithm BROADCAST.1:
  (* The links of each node is assigned dimension 1 to n *)
  (* the algorithm is executed at every node *)
begin
  if u0=us then weight:=n+1;
  (* otherwise u0 receives a weight from another node *)
  for i:=1 to weight-1 do
      begin
          new.weight :=i;
          send message to dimension i with new.weight
      end;
end.
```

Figure 4.4 A broadcast algorithm using a *weight*

The broadcast algorithms presented in [BrSc86, HoJo86] and [SaSc86] do not use explicit *weight*. Instead, the relative address of local node with respect to the source node is used to direct the broadcasting. The basic idea can be described as follows [BrSc86]. The neighbors of any node u in an n-cube are of the form $a(u)\oplus 2^i$ for $i=0,1,\cdots,n-1$. To implement a broadcast, each node sends broadcast message only to

those neighbors $a(u) \oplus 2^j$ such that $2^j > a(u)$. The algorithm is shown in Figure 4.5 as BROADCAST.2.

Algorithm BROADCAST.2:

```
begin
   (* Input: actual local address a(u_0) and source address a(u_s) *)
   If a(u_0)=a(u_s) then
      Send message to all neighbors
   else
   begin
      Send the message to local processor;
      (* Calculate relative address of local node with respect to to the source *)
      r(u_0)≡b_(n-1)b_(n-2) ··· b_j ··· b_0=a(u_0)⊕a(u_s);
      Find the largest l such that b_l=1;
      Send out message at the j-th dimension for all l<j≤n-1
         (* to node(s) a(u_0)⊕2^j *)
   end;
end.
```

Figure 4.5 A broadcast algorithm without using *weight*

Algorithm BROADCAST.2 is simple, and easy to implement. If the message has to carry the source address for other purpose, then the algorithm does not require any extra message overhead. This algorithm is actually implemented in our hardware router design to be presented in Chapter 6. However, this algorithm has a severe disadvantage in that it works well only in fault-free hypercube.

Katseff introduces a version of BROADCAST.1, which replaces the integer *weight* by a Boolean array *travel* [Kats88]. The algorithm BROADCAST.3 shown Figure 4.6 is modified from the broadcast algorithm (*Algorithm 5*) presented in [Kats88]. Algorithm BROADCAST.3 is more consistent with our definition of dimensions than algorithm BROADCAST.1. In algorithm BROADCAST.3, *Control** is a control vector received at the current node. If bit position *i* in *Control* * has value 1, then the received message is sent

Algorithm BROADCAST.3:

```
1. begin
     (* if u0=us, the router get a Control* vector with all bits set to 1 *)
     (* otherwise, u0 receives a Control* vector from a parent node *)
2.    for j:=0 to n−1 do
3.      begin
4.        send message to local processor;
5.        if Control*[j]=1 and dimension j is fault-free then
6.          begin
              (* form Control_j vector *)
7.            for b:=0 to n−1 do
8.              if Control*[b]=1 and b>j
                  then Control_j[b]:=1
9.              else Control_j[b]:=0;
10.           send message with Control_j to dimension j
11.         end;
12.     end;
13. end.
```

Figure 4.6 A broadcast algorithm using a *Control* vector

to dimension *i*. $Control_i$ is the control vector to be sent to dimension *i* from current node. Different dimensions will receive different new *Control* vectors calculated based on the the dimension *id* and the content in the received *Control** vector. At the source node, the vector *Control** is initialized to all 1's. The algorithm is to be executed in the *router* or *I/O processor* of every node.

Comparing algorithm BROADCAST.2 with algorithm BROADCAST.3, we can see that the two algorithms are essentially the same. However, BROADCAST.3 is more flexible and versatile than BROADCAST.2. As will be discussed later in Chapter 7, since the *Control* vectors are calculated at each node, algorithm BROADCAST.3 can be modified to route messages in a hypercube with certain faulty nodes. However, BROADCAST.3 is not as simple as BROADCAST.2. Also it requires each message to carry a distinct *Control* vector.

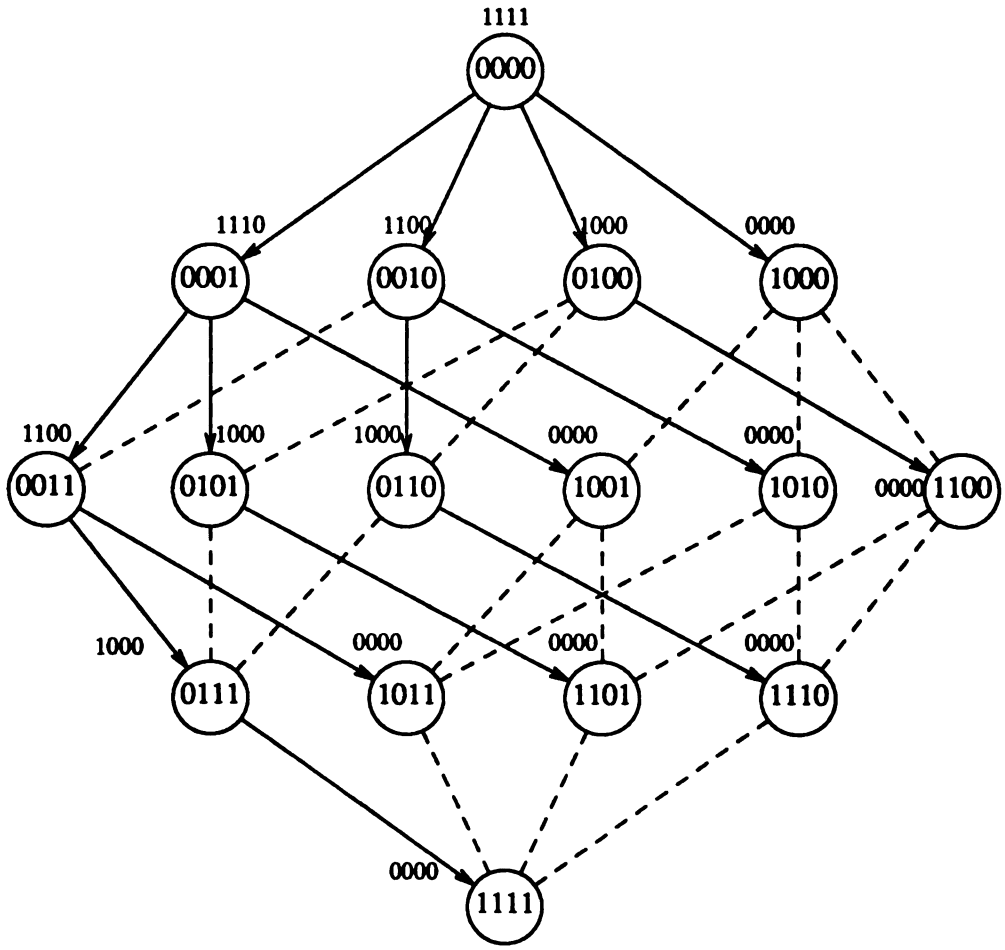


Figure 4.7 A broadcast tree generated by BROADCAST.3

The broadcast tree resulting from executing either algorithm BROADCAST.2 or BROADCAST.3, in a 4-cube, is shown in Figure 4.7, where a circle represents a node with its address inside the circle. The binary numbers outside a circle is the binary *Control* vector received by that node. The source address is assumed to be 0000. The arrows form the message routing pathes, while the dashed lines are the links not involved in this particular broadcasting.

4.4. MULTICAST IN HYPERCUBE ENVIRONMENT

We have discussed unicast and broadcast in hypercube environment. In this section, we take a look at multicast communication. We first investigate special characteristics of multicast and examine some simple examples. Then we will go to general case in next chapter.

Let $Q_n(V,E)$ be the topology of n-cube as defined above, and a node u_0 be the source node in a multicast. The node set V of graph $Q_n(V, E)$ can be partitioned into $n+1$ disjoint subsets (*n+1 levels*), $V_0,V_1,\cdots,V_n$, with respect to node u_0, where $V_i=\{v\in V \mid d_{Q_n}(u_0,v)=i\}$.

The *level* of a node u, denoted by $\lambda(u)$, is equal to $d_{Q_n}(u_0,u)$. Of course, $\lambda(u_0)=0$. From here on, we assume the source node has address zero, i.e., $a(u_0)=0$. If, however, $a(u_0)\neq 0$, then we can relabel the nodes by *XOR*ing all node addresses with the source address. Thus, if $u\in V_j$, then $||a(u)||=d_{Q_n}(u_0,u)=\lambda(u)=j$ (note that $a(u_0)=0$). Also, $D=\{u_1,u_2,\cdots,u_k\}$ will denote the set of destination nodes in a multicast. And set $M=\{u_0\}+D$ will be called a *multicast set*. Furthermore, for simplicity of discussion, we assume that $\lambda(u_1)\leq\lambda(u_2)\leq\cdots\leq\lambda(u_k)$.

Definition: A node u is an *ancestor* of a node v (or v is a *descendant* of u) if and only if u is contained in a shortest path joining u_0 and v. Thus, $\lambda(u)\leq\lambda(v)$ and $\lambda(v)-\lambda(u)=d_{Q_n}(u,v)$.

Definition: Let $X \subseteq V-\{u_0\}$ be a non-empty set. Then, the *common ancestors* of X, the set $CA(X)$, is defined as follows:

$$CA(X)=\{u \in V \mid u \text{ is an ancestor of every node } v \in X\}.$$

Note that $CA(X)$ is not empty, as for any X, $u_0 \in CA(X)$. The *nearest common ancestor* of X, denoted by $NCA(X)$, is defined as

$$NCA(X)=\{u \in V \mid u \in CA(X), \text{ and for every } v \in CA(X), \lambda(u) \geq \lambda(v)\}.$$

Lemma 1: Let $X \subseteq V-\{u_0\}$ and $|X|=2$. Then $NCA(X)$ contains only one element. That is, the nearest common ancestor of any two nodes is unique. Furthermore, the address of this node can be calculated by bitwise *AND*ing the binary addresses of the two nodes.

Proof: Let $X=\{u_1,u_2\}$, and "&" denote the bitwise AND operation on two or more binary numbers. For simplicity, denote $\lambda(u_i)$ by λ_i. Furthermore, let u_x be the node whose binary address $a(u_x)$ is given by

$$a(u_x)=a(u_1)\&a(u_2) \tag{4.1}$$

This implies that

(*A*). $a(u_x)$ has value 1 at exactly λ_x bit positions and value 0 at the remaining $n-\lambda_x$ bit positions, and

(*B*). $a(u_1)$ and $a(u_2)$ have value 1 at the same λ_x bit positions as $a(u_x)$. Furthermore, at each of the $n-\lambda_x$ bit positions, at least one of the two binary addresses $a(u_1)$ and $a(u_2)$ has value 0.

We first prove $u_x \in NCA(X)$. It is obvious from Eq. (4.1) that $u_x \in CA(X)$. In order to prove the *nearest* requirement, suppose there exists another node $u_y \in CA(X)$ with $\lambda_y > \lambda_x$. With no loss of generality, assume the only difference between $a(u_y)$ and $a(u_x)$ is that $a(u_y)$ has value 1 at one more bit position, say b_h, among those $n-\lambda_x$ bit positions. That is, $\lambda_y=\lambda_x+1$. By condition (*B*), $a(u_y)$ differs from at least one of the two addresses $a(u_1)$ and $a(u_2)$, say $a(u_1)$, at one more bit position than $a(u_x)$ does. We then have $d_{Q_n}(u_y,u_1)=\lambda_1-\lambda_x+1$. However, $\lambda(u_1)-\lambda(u_y)=\lambda_1-(\lambda_x+1)=\lambda_1-\lambda_x-1$. Thus,

$\lambda(u_1)-\lambda(u_y)\neq d_{Q_n}(u_y,u_1)$, u_y is not an ancestor of u_1. Therefore, $u_y \notin CA(X)$, λ_x is maximum. We conclude $u_x \in NCA(X)$.

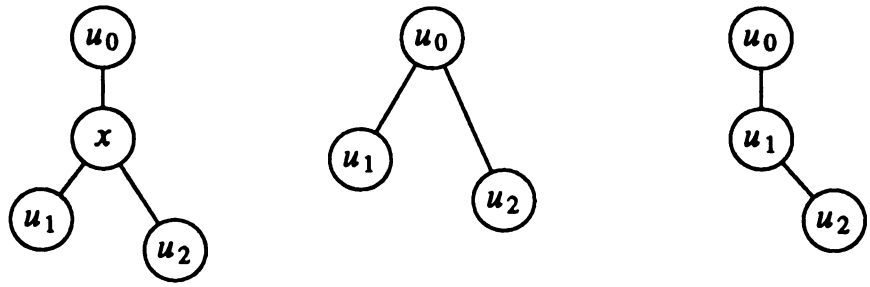
Now let us prove the uniqueness, that is, $NCA(X)=\{u_x\}$. Suppose there is another node $u_z \neq u_x$, such that $u_z \in NCA(X)$. Then, we must have $\lambda_z=\lambda_x$, which implies $||a(u_z)||=||a(u_x)||$. With no loss of generality, suppose $a(u_z)$ has a 0 at one of the λ_x bit positions, say b_g, and a 1 at one of the $n-\lambda_x$ bit positions, say b_h. Again, from condition (B), $a(u_z)$ differs from at least one of the two addresses $a(u_1)$ and $a(u_2)$, say $a(u_1)$, in *two* more bit positions than $a(u_x)$ does, i.e., $d_{Q_n}(u_z,u_1)=\lambda_1-\lambda_x+2$. However, $\lambda(u_1)-\lambda(u_z)=\lambda(u_1)-\lambda(u_x)=\lambda_1-\lambda_x$. Thus, $\lambda(u_1)-\lambda(u_z)\neq d_{Q_n}(u_z,u_1)$, u_z is not an ancestor of u_1, and $u_z \notin NCA(X)$. Therefore, $NCA(X)=\{u_x\}$. ■

The argument presented in Lemma 1 can be easily generalized to the case of $|X|>2$. Thus we have the following theorem.

Theorem 4.1: Let $X \subseteq V-\{u_0\}$ and $|X|=k$, $(2 \leq k \leq 2^n-1)$. Then $NCA(X)$ contains only one element. That is, the nearest common ancestor of any k nodes is unique. Furthermore, the address of this node can be calculated by bitwise *AND*ing the binary addresses of all the k nodes.

Let us start to investigate the multicast for a small number of destinations, and then we will get into general case. We first consider the situation that a source u_0 wants to send a message to only two other destinations, u_1 and u_2. We first investigate all possible topological patterns in the solution space of OMT.

All the possible patterns of OMTs for $|M|=3$ is illustrated in Figure 4.8. In the figure, circles denote nodes and lines denote shortest pathes joining pairs of nodes. Observe that if T2.1 is the desired subtree, then node $x=NCA(u_1,u_2)$. Similarly, if T2.2 is the desired subtree, we have $u_0=NCA(u_1,u_2)$. Finally, when T2.3 is the solution, $u_1=NCA(u_1,u_2)$. In fact, we may refer to T2.1 as the *general pattern* of the OMT when $|M|=3$. This is because when $x=u_0$, we obtain subtree T2.2, and when $x=u_1$, we obtain subtree T2.3.



General Pattern T 2.1 | $x=u_0$ T 2.2 | $x=u_1$ T 2.3

Figure 4.8 One-to-two multicast trees in hypercube

The above observations lead us to an algorithm for generating an OMT when $M=\{u_0,u_1,u_2\}$. The algorithm is listed in Figure 4.9 as algorithm One_to_Two_Multicast.

Algorithm One_to_Two_Multicast
input: $a(u_0)$, $a(u_1)$, $a(u_2)$;
$a(x) := a(u_1)$ & $a(u_2)$;
form a path between u_0 and x;
form a path between x and u_1;
form a path between x and u_2.

Figure 4.9 Multicast algorithm for two destination case

In the One_to_Two_Multicast algorithm, to form a path between node u and node v implies that any shortest path between u and v is acceptable. If $u=v$, then no path has to be established. The validity of the above algorithm can be seen as follows. Denote the multicast tree found by the algorithm by $T(V,E)$. Let $UB=||a(u_1)||+||a(u_2)||$ (which is a upper bound of the traffic). In all three subtrees, $|E(T)|=UB-d(u_0,x)$. By Lemma

1, x is the unique NCA of u_1 and u_2, which implies $d(u_0,x)=\lambda(x)$ is maximum. Thus, $|E(T)|$ is minimum.

We now consider multicast with three destination nodes, i.e., $M=\{u_0,u_1,u_2,u_3\}$. Again, we investigate all possible subtree patterns, find a general pattern, and then give an optimal algorithm. The nine possible subtree patterns are shown in Figure 4.10. Under the assumption that $\lambda(u_1)\leq\lambda(u_2)\leq\lambda(u_3)$, in Figure 4.10, a pattern having nodes labeled u_1, u_2 and u_3 represents a unique tree configuration. Each of the other patterns actually represents three *non-isomorphic* subtrees obtained by permuting i,j, and k, namely, $(i,j,k)\in\{(1,2,3),(1,3,2),(2,3,1)\}$. Observing the nine patterns, we can see that T3.1 is a general pattern. When $y=x_{12}=x_{23}=x_{13}$, T3.1 becomes T3.2, and when $x_{ij}\in\{u_1,u_2\}$, T3.1 becomes T3.3. When $y=u_0$ and $y=u_1$, the tree patterns in first row correspond to patterns in second and third rows, respectively. Clearly, node y is $NCA(u_1,u_2,u_3)$. Also, node x_{ij} must satisfy

$$\lambda(x_{ij})=\max\{\lambda(NCA(u_1,u_2)),\lambda(NCA(u_2,u_3)),\lambda(NCA(u_1,u_3))\}.$$

Since there is only one general pattern, we can again write a simple algorithm to solve the problem. which is listed in Figure 4.11.

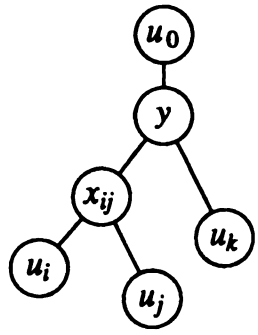
Denote the subtree found by the above algorithm by $T(V,E)$, and let

$$UB=||a(u_1)||+||a(u_2)||+||a(u_3)||.$$

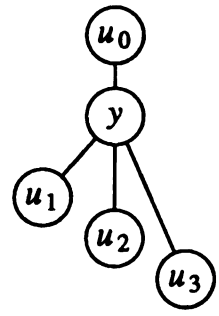
It is not difficult to see that the following equation holds for every subtree pattern in Figure 4.10.

$$\begin{aligned}|E(T)|&=UB-2d(u_0,y)-d(y,x_{ij})=UB-2||a(y)||-(||a(x_{ij})||-||a(y)||)\\&=UB-||a(y)||-||a(x_{ij})||.\end{aligned}$$

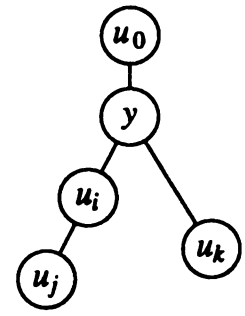
Since UB is fixed, $a(y)$ is unique by Theorem 4.1. The maximum level number of x_{max} means $||a(x_{ij})||$ is maximum. Therefore, $|E(T)|$ is minimum. This establishes the validity of the above algorithm.



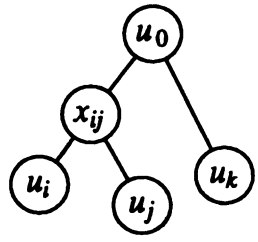
general pattern
T 3.1



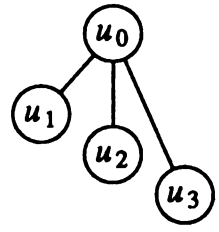
$y=x_{12}=x_{23}=x_{13}$
T 3.2



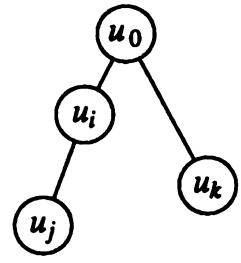
$x_{ij}=u_i$
T 3.3



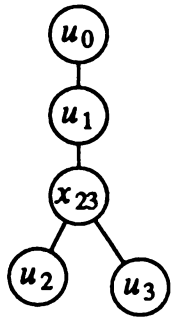
$y=u_0$
T 3.4



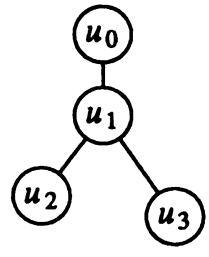
$u_0=y=x_{12}=x_{23}=x_{13}$
T 3.5



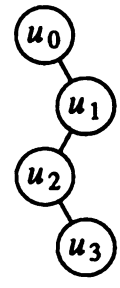
$y=u_0$; $x_{ij}=u_i$
T 3.6



$y=u_1$
T 3.7



$y=u_1=x_{23}=x_{12}=x_{13}$
T 3.8



$y=u_1$; $x_{23}=u_2$
T 3.9

Figure 4.10 One-to-three multicast tree patterns

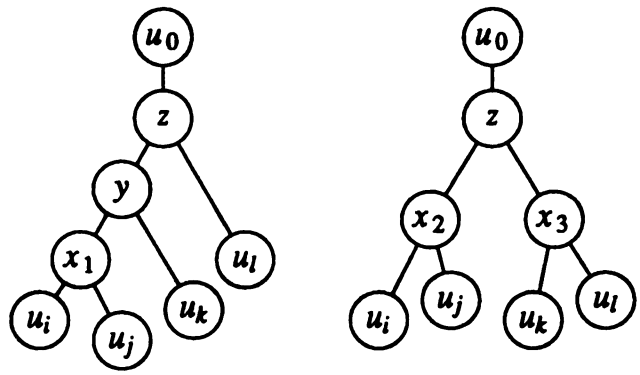
Algorithm One_to_Three_Multicast

input: $a(u_0)$, $a(u_1)$, $a(u_2)$, $a(u_3)$;
$a(y):=a(u_1)\&a(u_2)\ \&\ a(u_3)$;
$a(x_{12}):=a(u_1)\&a(u_2)$, ;
$a(x_{23}):=a(u_2)\&a(u_3)$, $a(x_{13}):=a(u_1)\&a(u_3)$;
$x_{max} := x_{ij}$, such that $\lambda(x_{ij}):= \max\{\lambda(x_{12}),\lambda(x_{23}),\lambda(x_{13})\}$;
form a path between x_{max} and u_i;
form a path between x_{max} and u_j;
form a path between x_{max} and y;
form a path between y and u_k; (* $k \neq i,j$ *)
form a path between y and u_0.

Figure 4.11 Multicast algorithm for three destination case

Finally, let us have an overview on four or more destination multicast. As we have seen, the possible configurations of the multicast trees grow dramatically when the number of destination nodes increases even from two to three. As the number of destinations increases from three to four, the problem becomes more complicated. The most important factor is that, in the case of four destination there is no unique general pattern; instead, there are two general patterns as shown in Figure 4.12. If we want to write an optimal algorithm, we have to deal with the two general patterns separately, then select the one having less intermediate nodes.

In general, if we have a k-destination multicast problem, one way of finding an OMT is to generate all the general patterns, consider each of them separately, and then select an optimal one. However, the number of general patterns grows very rapidly as the number of destinations increases. We have counted 3, 6, 11, and 23 general patterns for 5, 6, 7, and 8 destinations, respectively. We can see the trend of how rapidly the number of tree patterns and general patterns grows as the number of destinations increases. Also, the general patterns become more and more complicated.



General Pattern 4.1 General Pattern 4.2

Figure 4.12 One-to-four general multicast tree patterns

Can we find an efficient algorithm to solve the *OMT* problem? For a general graph, the answer is **no**. In fact, it has been shown in [ChEN87] that the problem of finding an *OMT* for a general k is NP-hard even if $G(V, E)$ is *bipartite*.

A problem which is very similar to the above problem is known as the *Steiner Tree* (ST) problem. It is the problem of finding the smallest subtree of a given graph, which contains a given subset of nodes [GaJo79]. It can be observed that if condition (b) in the definition of *OMT* is removed, *OMT* problem becomes a ST problem. It has been shown that ST problem is NP-complete when $G(V,E)$ is the hypercube graph [GrFo82].

The OMT model can be applied to any interconnection topology. From here on, we only consider multicast in the hypercube topology. Now the question is "Is the OMT problem still NP hard for the hypercube topology"? Based on the above observations, we conjecture that the OMT problem remains NP-hard even for hypercube topology.

CHAPTER 5

A DISTRIBUTED MULTICAST ALGORITHM FOR HYPERCUBE MULTIPROCESSORS

As discussed in the last chapter, it is impractical to find an OMT when the number of destinations is large. Also, the optimal algorithms discussed so far are all centralized in the sense that they require the entire routing be handled solely by the source node. Subsequently, this approach requires the information of the entire paths be carried by the message header, and thus increases the interprocessor communication overhead. In this chapter, we propose a distributed heuristic algorithm which has the following properties. First, the distance from the source node to each destination node in the multicast tree generated by the proposed algorithm is the same as that in an OMT. Second, the algorithm is simple and can be easily implemented in hardware. Third, it allows distributed routing. Finally, simulation results indicate the traffic generated by the algorithm is very close to the optimal solution and is better than existing multi-destination message delivery mechanisms.

5.1 UNDERLYING RATIONALE OF THE MULTICAST MULTICAST ALGORITHM

Observe that in a multicast tree only forward nodes are involved in passing the multicast message to some other nodes in the multicast tree. Initially, a node, called the *source* node, decides to send its message to some number of other nodes, i.e., it issues a

multicast message. By running the algorithm, the source node will decide which of its neighboring nodes should receive the message. A message received by a node v includes the data unit, the address of the source node, and a list of destination nodes (referred to as *destination list*) which are descendants of node v in the multicast tree. Detailed description of the message format will be given in Chapter 6.

Each node, upon receiving a multicast message, will perform the following functions. First, it will compare its own address against the addresses in the destination list of the received message. If there is a match, that matched address will be removed from the destination list and a copy of the data field will be sent to the local processor. Then, if the destination list is empty, the node is a leaf node in the multicast tree and no message will be further forwarded. However, if the destination list is not empty (which implies the node is a forward node), the node will execute the algorithm to determine its descending neighbors in the multicast tree. Depending on the number of descending neighbors, say m, the forward node will split its destination list into m disjoint destination sublists, each consisting of a set of destination nodes which are descendants of a particular descending neighbor. Each such destination sublist is put into a message header and sent to its corresponding descending neighbor.

Now the question is how each forward node decides which of its neighboring nodes to pass the received message. Consider a two-destination multicast, as shown in T2.1 (Figure 4.8). In a hypercube environment, suppose the relative addresses of two destination nodes u_1 and u_2 have 1's at p common bit positions ($p \leq \lambda(u_1) \leq \lambda(u_2)$), which define the intermediate node x. For a message to go from node u_0 to x, it may traverse through these dimensions in any order (by changing the values at these p bit positions from 0 to 1, one at a time, in any order). As discussed before, there are $p!$ different paths between nodes u_0 and x. From the point of view of this message delivery, any of these paths has the same effect since they all require the same number of time steps and create the same amount of traffic.

Now, suppose that node u_0 wants to include an additional node, u_3, in its destination list. In this case, we have the situation of T3.1 (Figure 4.10). Let $\lambda(y)=||a(u_1)\&a(u_2)\&a(u_3)||=q$, and $q<p$. In this situation, the q dimensions are a subset of the original p dimensions. Obviously, these q dimensions have to be traversed first, and then those $p-q$ dimensions. Otherwise, y would not be the *NCA* of u_1, u_2, and u_3.

In order to have an intuitive idea, let us take a look at Figure 4.1 again. Why is 4.1(b) better than 4.1(a)? Notice that $u_s \oplus u_1=011$. Relative addresses 011 indicates that a path from u_s to u_1 has to pass both dimension 0 and dimension 1, sooner or later. Similarly, we have $u_s \oplus u_2=101$, which means a path from u_s to u_2 has to pass dimensions 0 and 2 any way. The paths to both destinations have to pass a dimension 0. Thus, in Figure 4.1 (b), the source node first sends out the message to dimension 0 (node 001); then at node 001, the message is duplicated and sent out to the two destinations. By doing this, one unit of traffic is saved for that message delivery.

The above discussion suggests that each forward node can use the relative binary addresses of all its destination nodes to *vote* for preferred dimensions (bit positions). The process works as follows. Each of the n bit positions has a counter. For each destination, if its binary address has value 1 at c bit positions, the corresponding c counters are increased by 1. Then, if the counter of a particular bit position, say j, has the maximum number of 1's, the j-th dimensional neighbor of the forward node u_0 is selected. In case of a tie, select one of the tied bit positions at random. For simplicity, a lower bit position is selected in the actual implementation.

All the destination nodes whose binary addresses have value 1 at bit position j are selected to form a destination sublist in the message sent to the j-th dimensional neighbor. This implies that the neighbor is responsible for passing the message to all those destinations in the list. The same procedure continues for the remaining destinations which, of course, do not have 1 at their bit position j. Then we can find the second

destination sublist if the set of the remaining destination nodes is not empty. This procedure is repeated until all destination nodes have been resolved.

5.2 THE GREEDY MULTICAST ALGORITHM

As mentioned earlier, our multicast algorithm is executed by each forward node. We now present an hueristic algorithm for the multicast. In the following algorithm, $a(u_s)$ and $a(u_0)$ represents the actual addresses of the source node and current forward node, respectively, and $a(u_1), a(u_2), \cdots, a(u_k)$ represent the actual addresses of the k destination nodes. For the algorithm to work, it is not necessary to sort the destination addresses according to their distances from the source, as this was previously assumed in order to simplify the explanation of the multicast problem. However, a forward node has to perform an *XOR* operation on its own address and the addresses of its descending destinations to change their actual addresses to relative addresses. The corresponding relative addresses are $r_0(u_0)=a(u_0)\oplus a(u_0)=0$, and $r_0(u_i)=a(u_0)\oplus a(u_i)$, for $1\le i\le k$. The listing of the heuristic algorithm is given in Figure 5.1. The algorithm is also called GREEDY multicast algorithm, since it selects a dimension of maximum column-sum whenever it finds one.

Lemma 5.1: Given a source node u_s and a destination list $D=\{u_1, u_2, \cdots, u_i, \cdots, u_k\}$ in a Q_n, algorithm MULTICAST.1 passes a message from u_s to every destination $u_i \in D$ through a shortest path between u_s and u_i.

Proof: Let the path from u_s to u_i be $p=(u_s, v_1, \cdots, v_j, \cdots, v_{d-1}, u_i)$. It is implied that u_i is in the destination sublist received by every forward node v_j along the path. Assuming $a(v_j)$ and $a(v_{j+1})$ differ at bit position l, then u_i is included in the destination sublist sent from v_j to v_{j+1} only if $a(v_j)$ and $a(u_i)$ differ at bit position l. Also, $a(v_{j+1})$ and $a(u_i)$ must agree at bit position l. That is, each forward node passes messages to its children nodes only. Therefore, it is clear that for every destination u_i, if $||a(u_s)\oplus a(u_i)||=d$, then $||a(v_j)\oplus a(u_i)||=d-j$ holds for $1\le j\le d-1$. Starting from the

Input: Local address: $a(u_0)$;
Destination list: $D=\{a(u_1),a(u_2),\cdots,a(u_k)\}$.

Output: Destination sublist(s): $D_1, D_2, \cdots, D_g$
where $D_i \subseteq D$, for $1 \le i \le g$; and $D_i \cap D_j = \emptyset$, for $i \ne j$.

Algorithm MULTICAST.1:

1. (* Calculate relative addresses: *)
   $r_0(u_i)=b_{i(n-1)}b_{i(n-2)}..b_{ij}..b_{i0} \equiv a(u_0) \oplus a(u_i)$, for $1 \le i \le k$;
2. (* if local processor is a destination, send a copy to it *)
   If $r_0(u_i)=0$ for some $i \in [1,k]$, send the message to local processor;
3. (* calculate column sums *)
   $c_j=\sum_{i=1}^{k} b_{ij}$, for $0 \le j \le n-1$;
4. $p=0$; (* start loop *)
5. (* select a dimension with maximum column sum, *)
   (* lower dimension has higher priority in case of a tie *)
   Find smallest l, such that $c_l \ge c_j$ for all $0 \le j \le n-1$;
6. If $c_l=0$, stop.
7. $D_p=\emptyset$;
8. (* form a new destination sublist, reset corresponding rows *)
   For each $r_0(u_i)$, $1 \le i \le k$, if $b_{il}=1$, then
8.1. $D_p=D_p+\{r_0(u_i) \oplus a(u_0)\}$;
8.2. Set $r_0(u_i)=0$;
8.3. $c_j=c_j-b_{ij}$ for $0 \le j \le n-1$;
9. Put destination sublist D_p into message header, send out the message at l-th dimension (to node $a(u_0) \oplus 2^l$);
10. (* start the selection of another dimension *)
    $p=p+1$; Goto step 5.

Figure 5.1 Greedy multicast algorithm

source node u_s, after $d-1$ steps, the message will travel to node v_{d-1}, whose address has only 1 bit position in difference from $a(u_i)$. The message then travels to to u_i through the direct link between v_{d-1} and u_i. That is, the message travels from u_s to u_i in d steps if the addresses of u_s and u_i differ at d bit positions. The path is therefore a shortest one. ■

Lemma 5.2: Given a source node u_s and a destination list $D=\{u_1,u_2,\cdots,u_i,\cdots,u_k\}$ in a Q_n, by executing algorithm MULTICAST.1 at all forward nodes, every destination node $u_i \in D$ will receive the message exactly once.

Proof: Lemma 5.1 has shown that each destination node receives a copy of the message through a shortest path. In the proof of Lemma 5.1, we have also shown that a message will travel from a forward node v_j to a destination u_i only when u_i is in the destination sublist received by v_j. We note that during the execution of algorithm MULTICAST.1 at forward node v_j, each destination node in the destination sublist is passed to one of v_j's children nodes only. This is true for any forward node. Also note that a message always travels from a parent node to a child node. Therefore, no two nodes at the same level (with respect to the source node) could have the same destination node in the destination sublist it receives. Therefore, only one copy of the message is received by each destination. The conclusion follows. ■

Theorem 5.1: Given a source node u_s and a destination list $D=\{u_1,u_2,\cdots,u_i,\cdots,u_k\}$ in a Q_n, the edges selected by algorithm MULTICAST.1 at all forward nodes induce a Multicast Tree.

Proof: Let the subgraph formed by the edges selected by algorithm MULTICAST.1 be $T(V,E)$. It is obvious that T is a connected graph. We now show that there is no cycle in T. Note that if a node $u_x \notin D$ and u_x does not reside on a path from u_s to any $u_i \in D$, then $u_x \notin V(T)$. We now prove that the paths from u_s to all $u_i \in D$ do not form any cycle. We prove by contradiction. Assume there is a node $u_y \in V(T)$, such that there exist two paths from u_s to u_y in T. First, obviously it is impossible that $u_y \in D$, since

Lemma 5.2 has shown that the path from u_s to every $u_i \in D$ is unique. Now suppose $u_y \notin D$, and the two paths from u_s to u_y are $p=(u_s, v_1, \cdots, v_j, \cdots, v_{y-1}, u_y)$ and $p'=(u_s, v'_1, \cdots, v'_j, \cdots, v'_{y-1}, u_y)$. Assume j is the smallest number such that $v'_j \neq v_j$. Also assume $a(v_{j-1})$ and $a(v_j)$ differ at bit position l, and $a(v_{j-1})$ and $a(v'_j)$ differ at bit position l'. Then $a(v_{j-1})$ and $a(u_y)$ must differ at both bit positions l and l'. If dimension l is selected first by the algorithm, then the addresses of all destination nodes in the destination sublist received by v_j differ from $a(u_y)$ at bit position l, while the addresses of all destination nodes in the destination sublist received by v'_j must agree with $a(u_y)$ at bit position l. Thus, the message passed to v'_j will never travel through dimension l, therefore, will not reach node u_y. If dimension l' is selected first, the same situation will happen. Therefore, it is impossible that a cycle could exist in T, and thus T is a tree. Furthermore, Lemma 5.2 shows that every destination node receives a message through a unique shortest path, we conclude that T is a multicast tree.

Lemma 5.3: The multicast tree constructed by algorithm MULTICAST.1 is an Optimal Multicast Tree when the number of destinations is less than 3.

Proof: When the number of destination is one, that is, in a unicast case, it is obvious that the multicast algorithm will generate a path form the source to the destination exactly the same way as the unicast algorithm does. For the case of two-destination multicast, we show that algorithm MULTICAST.1 works in the same way as algorithm One_to_Two_Multicast (Figure 4.9) does. Suppose $D=\{u_1, u_2\}$. We have $r_s(u_i)=a(u_s) \oplus a(u_i)$ $(1 \leq i \leq 2)$. Assuming $||r_s(u_1) \& r_s(u_2)||=m$, then when executing the algorithm at u_s to find column-sums, m column-sums will have value 2. The m dimensions will be selected by the multicast algorithm one by one from the lowest dimension to the highest dimension along the path. After m steps, the message will reach the NCA of u_1 and u_2, which is node u_x with $a(u_x)=a(u_1) \& a(u_2)$. The order of traveling the m dimensions is immaterial. At node u_x, all column-sums will be either 1 or 0. The message will travel to u_1 and u_2 through two separate paths. As detailed in

the proof of algorithm One_to_Two_Multicast, it is optimal. ■

We would like to mention that the algorithm is distributed in the sense that each forward node only decides which neighboring nodes to pass the message. Only the destination addresses are carried in the message header. No intermediate node addresses need to be carried. Also, each forward node which receives a message will execute the same algorithm. The example in next section will make the idea clear, and show how a multicast tree can be generated by the algorithm.

5.3 AN ILLUSTRATIVE EXAMPLE

In this section, we illustrate through an example how the Greedy multicast algorithm works. Consider a 5-cube in which node 00110 (6) wants to send a message to nodes {00111 (7), 10100 (20), 11101 (29), 10010 (18), 00001 (1), 00000 (0)}. Initially, the source node 00110 is the only forward node which executes the greedy algorithm (u_0=u_s). The actual addresses of the source u_0, and destinations $u_1, \cdots, u_6$ are listed in Table 5.1.

The actual addresses of all destination nodes are first *XOR*ed with the actual address of u_0. The resulting relative addresses are put into a binary reference array $A[1..k, 0..n-1]$. Initially, row i of array A corresponds to the relative address of destination $u_i, (1 \le i \le k)$. The array is shown in Table 5.2.

The number of 1's in each row of array A indicates the distance between that destination and current node. The number of 1's in each column is counted to produce the vector *column_sum*, which has the values of (3, 1, 3, 4, 3) in this example. Bit position 1 has the maximum value of 4. Thus, bit position 1 (i.e., dimension 1) is selected to receive the message. Rows 2, 3, 5 and 6 which have value 1 at bit position 1 are picked up to form a destination sublist. The corresponding descending neighbor, thus, is 00110⊕00010=00100 and the message sent to this node contains the following destination addresses: $a(u_2)$,$a(u_3)$, $a(u_5)$ and $a(u_6)$. These four rows (rows 2, 3, 5 and 6) in

Table 5.1 The actual addresses of source (00110) and destinations

	Actual addresses $b_4b_3b_2b_1b_0$	Decimal
$a(u_0)$	0 0 1 1 0	6
$a(u_1)$	0 0 1 1 1	7
$a(u_2)$	1 0 1 0 0	20
$a(u_3)$	1 1 1 0 1	29
$a(u_4)$	1 0 0 1 0	18
$a(u_5)$	0 0 0 0 1	1
$a(u_6)$	0 0 0 0 0	0

Table 5.2 The reference array at node 00110

	Reference array 4 3 2 1 0	Distances
A[1, *]	0 0 0 0 1	1
A[2, *]	1 0 0 1 0	2
A[3, *]	1 1 0 1 1	4
A[4, *]	1 0 1 0 0	2
A[5, *]	0 0 1 1 1	3
A[6, *]	0 0 1 1 0	2
column_sum	3 1 3 4 3	

array A are then reset to all zeros. Array A now has new entries as shown in Table 5.3.

Now, the new column_sum becomes (1, 0, 1, 0, 1). Three bit positions, 4, 2, and 0 have the same maximum value of 1. Although Any one may be selected, we assume a lower bit position has a higher priority and thus bit position 0 is selected. As a result, destination address $a(u_1)$ forms another destination sublist included in the message sent to the descending neighbor 00110⊕00001=00111, which happens to be a destination node.

Table 5.3 The reference array after the first run of the algorithm

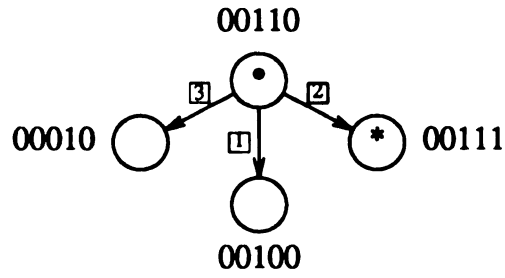
	Reference Array 4 3 2 1 0	Distances
A[1, *]	0 0 0 0 1	1
A[2, *]	0 0 0 0 0	
A[3, *]	0 0 0 0 0	
A[4, *]	1 0 1 0 0	2
A[5, *]	0 0 0 0 0	
A[6, *]	0 0 0 0 0	
column_sum	1 0 1 0 1	

Table 5.4 The reference array after the second run of the algorithm

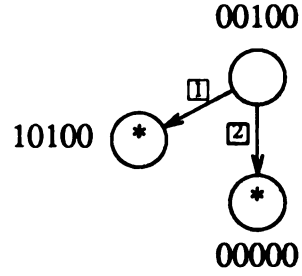
	Reference Array 4 3 2 1 0	Distances
A[1, *]	0 0 0 0 0	
A[2, *]	0 0 0 0 0	
A[3, *]	0 0 0 0 0	
A[4, *]	1 0 1 0 0	2
A[5, *]	0 0 0 0 0	
A[6, *]	0 0 0 0 0	
column_sum	1 0 1 0 0	

After reseting row 1 to zero, the only row left is row 4, as shown in Table 5.4. By repeating the same procedure, $a(u_4)$=10010 is the destination sublist to be included in the message sent to the descending neighbor 00010.

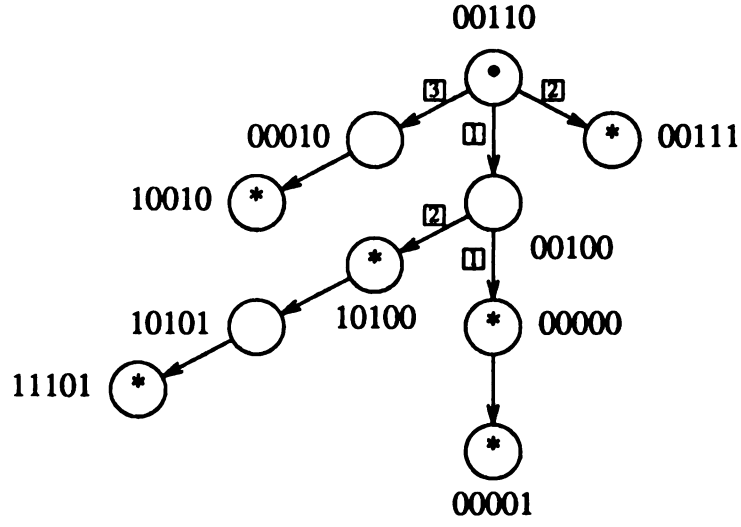
At this point, the multicast subtree with three descending neighbors, as shown in Figure 5.2(a), is formed. In this and following figures, an arrow represents a link, and the number inside a small square by a link indicates at which step of executing the algorithm that dimension is determined. Also, a node marked by "*" means it is a destination node, The source node is marked by a "•".



(a) The multicast subtree generated at node 00110



(b) The multicast subtree generated at node 00100



(c) The complete multicast tree

Figure 5.2 A multicast tree in a Q_5

Upon receiving a message from node 00110, node 00100 serves as a forward node. The corresponding multicast subtree of this node can be similarly generated and is shown in Figure 5.2(b).

By repeating this procedure at all forward nodes, the resulting complete multicast tree rooted at node 00110 is shown in Figure 5.2(c).

5.4 PERFORMANCE STUDY ON THE GREEDY ALGORITHM

In this section, we first estimate the time complexity of the greedy multicast algorithm, and then compare the performance of the algorithm with the optimal solution and other alternative methods for multi-destination message delivery.

5.4.1 Time complexity of the Greedy algorithm

We assume that the basic operations, such as addition, subtraction, comparison and assignment, have time complexity $O(1)$. Let n be the dimension of the hypercube and k be the number of destinations in the multicast. Then, in the algorithm, line 1 and 2 have time complexity of $O(k)$; line 3 has $O(kn)$. The number of loops between line 4 to 10 is at most n. Thus, we have $O(n^2)$ for line 5, $O(n)$ for lines 6 and 7, and $O(kn)$ for line 8. Lines 8.1 to 8.3 are executed at most k times regardless of the number of loops, which have the complexity of $O(3k+nk)$. Finally, we have $O(3n)$ for lines 4, 9 and 10. It's not difficult to figure out that the Greedy algorithm is of time complexity $O(4k+3n+3kn+n^2)$, or $O(nk+n^2)$ for large k and n.

In Section 5.3, we have proven that the greedy multicast algorithm guarantees a minimum message delivery time by providing shortest paths between source and each destination. Using the algorithm, the traffic generated for the multicast is also minimum when the number of destinations is less than 3. In general cases, the distribution of the destination (the locality of information) has great effect on the traffic generated by the message delivery. Several distributions have been suggested for the study of communication algorithms [ReFu87]. We would like to study the Greedy algorithm in *Uniform*

distribution and *Decreasing probability* distribution.

5.4.2 Performance under uniform distribution

Under the assumption of *uniform* routing distribution, the probability that node u_i sends a message to node u_j is the same for all $u_j \neq u_i$, $u_i, u_j \in V(Q_n)$. We would like to compare the performance of the Greedy algorithm with two alternative approaches for one-to-many communication: multiple one-to-one and broadcast. Issuing k one-to-one message deliveries for k-destination communication is the actual approach used in first generation hypercube multiprocessors. Implementing one-to-many using the broadcast communication, the router will not send the message to the local processor if the local address does not match any address in the destination list.

Figure 5.3 shows the amount of traffic generated by these different interprocessor communication methods in a 6-cube multiprocessor. A unit of traffic is measured as one message traverses over one link. In the figure, the number of destination nodes, k, is chosen within the range [1,63]. For a given $k \in [1,63]$, k destination addresses in the multicast set are selected at random (uniform distribution). Then, by executing a simulation program, the number of links involved in the message delivery is measured for each communication method. For each k, we repeat the simulation 1000 times, and the amount of traffic generated for a given k is averaged over the 1000 runs. The dashed curve, solid curve, and dotted line show the results from multiple one-to-one, Greedy algorithm and broadcast approach, respectively. The x-axis is the number of destinations, and the y-axis is the average traffic created by the message delivery. The greedy multicast algorithm always generates the least amount of traffic compared with the other two approaches.

In the broadcast method, the traffic generated is independent of the number of destination nodes and is $2^6-1=63$. For multiple one-to-one message delivery, under the uniform routing assumption, $\binom{n}{i}$ destinations have distance i from the source. The total number of destinations is 2^n-1. Thus, the average distance between a source and a

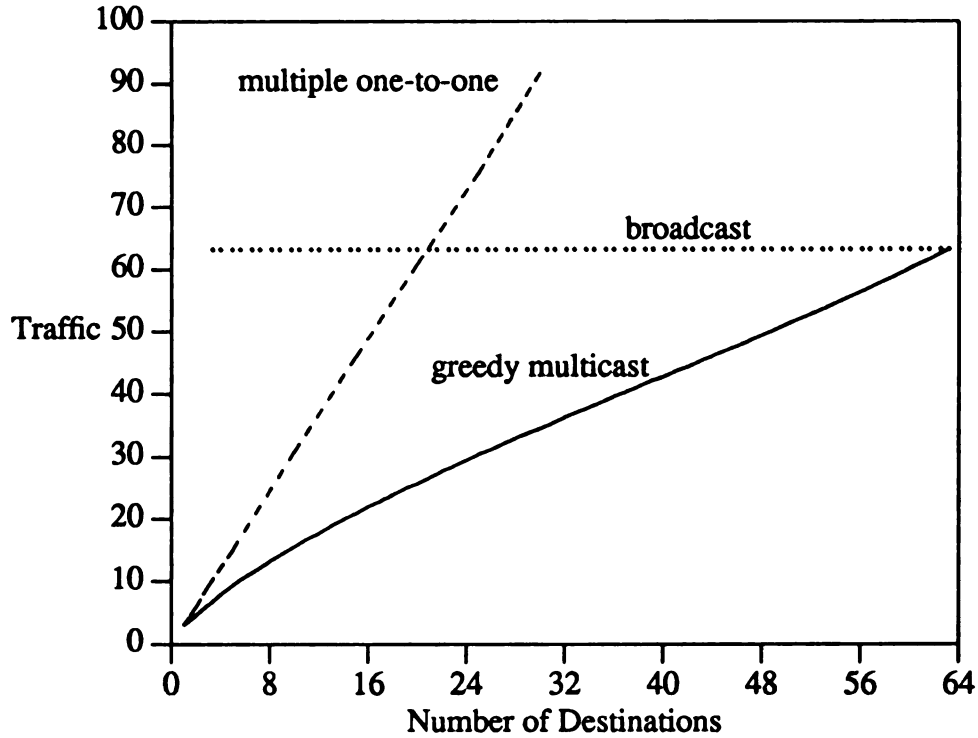


Figure 5.3 Comparison of three communication methods in a Q_6

destination, under uniform distribution, is

$$d_{mean} = \frac{1}{2^n - 1} \sum_{i=1}^{n} i \binom{n}{i} = n \times \frac{2^{n-1}}{2^n - 1}$$

For the case of n=6, d_{mean}=3.05. To send a message to a destination at h hops away will create h units of traffic. Thus, the average traffic generated for k destinations using multiple one-to-one communication is $3.05k$ for uniform distribution. It follows that when the number of destination nodes is greater than 20, even broadcast approach performs better than the multiple one-to-one approach. Note that multiple one-to-one approach is actually used in the first generation hypercube multiprocessors.

In order to better evaluate the performance of the Greedy algorithm, we also run simulation programs for an optimal algorithm and a multi-destination routing algorithm presented by Moler and Scott [MoSc86] with the name of *Spare global send.* The optimal solution is obtained by exhaustive searching and comparing of all possible paths for each given number of destinations (an exponential time complexity algorithm). The idea of the Spare global send can be briefly described as follows. Given a local (forward) node and a destination list, the algorithm first searches the destination list and selects one which is *closest* to the local node, a message is then sent to that node through a shortest path between the two nodes. Those destination nodes which are descendants of the selected node will receive the message through that node. The algorithm is repeated for the remaining nodes in the destination list until all destinations are dealt with. The algorithm is also distributed.

The results for the three algorithms are generated in a way similar to the curves in Figure 5.3, except that the simulation is for odd number of destinations, and for each given number of destinations the value is averaged over 100 runs instead of 1000 runs (since the optimal algorithm takes a long time). Figure 5.4 shows the three curves, where the x-axis is the number of destinations and the y-axis is the traffic created by the message delivery; the upper dashed curve, solid curve and lower dashed curve represent the result from spare global send, the Greedy multicast algorithm and the optimal solution, respectively. Also shown in Figure 5.4 is the dotted line, which is a lower bound of the multicast. For k destinations, the theoretical lower bound of traffic generated is obviously k, i.e., no non-destination forward nodes are needed.

From the curves, we can see that the performance of the greedy algorithm is better than the spare global send, and is very close to the optimal solution, especially when k is very small or very large. Note that it is proven in Section 5.2 that when $k<3$, the performance of the greedy algorithm is the same as that of the optimal algorithm. This can also be observed from the curves in Figures 5.4 and 5.5.

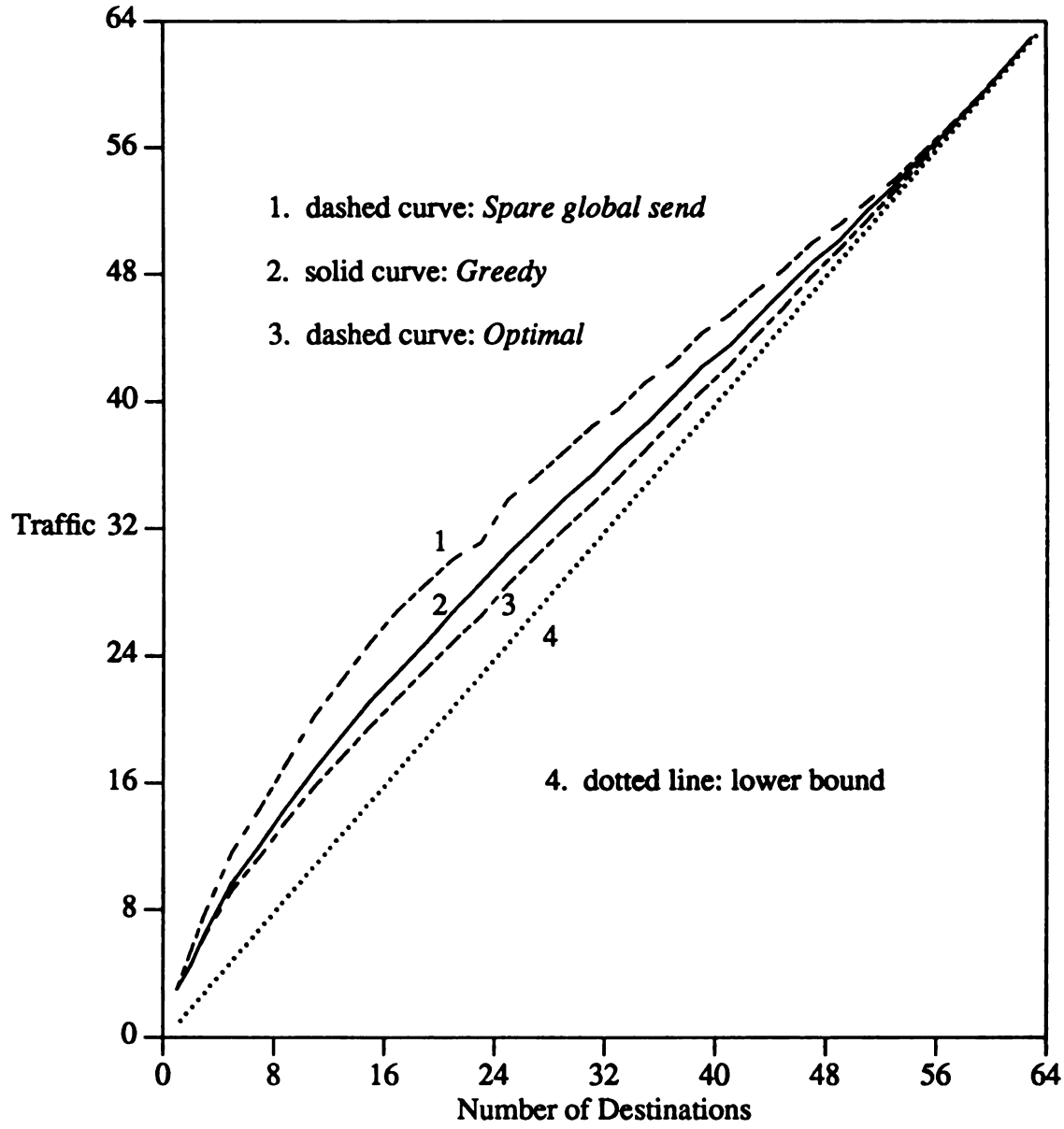


Figure 5.4 Performance comparison of three multicast algorithms

(under uniform distribution)

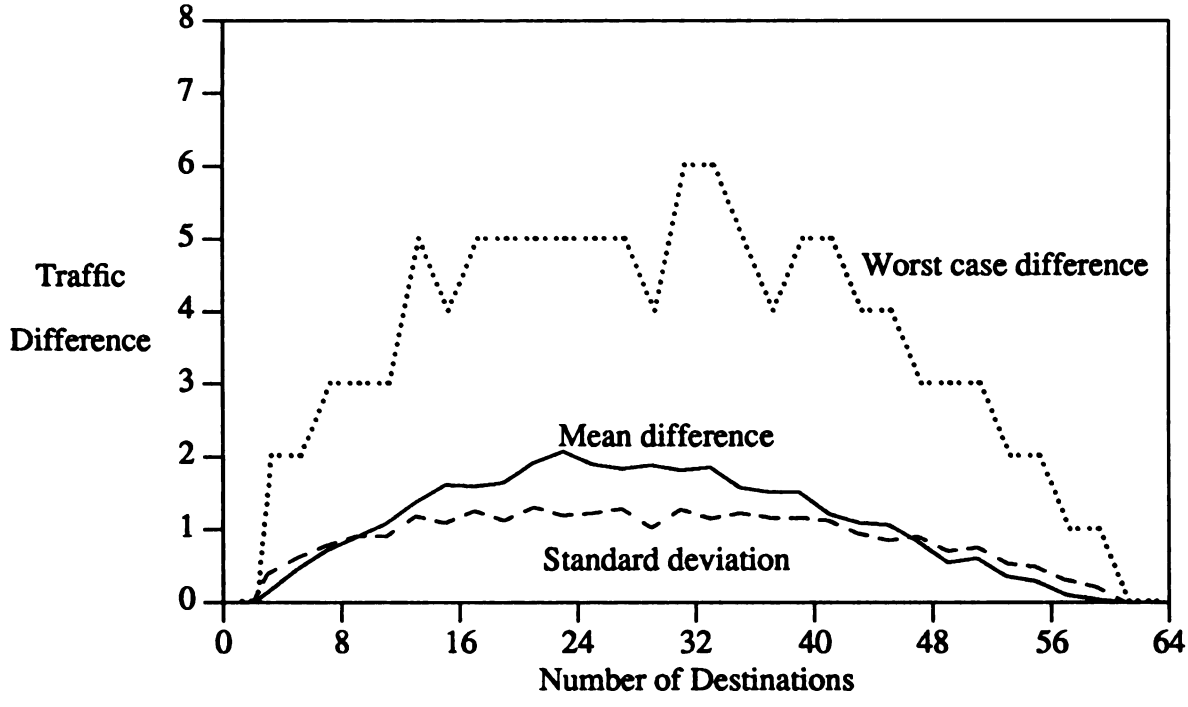


Figure 5.5 Performance comparison of Greedy algorithm with optimal solution

(under uniform distribution)

In order to more closely compare the performance of the Greedy algorithm with that of the optimal algorithm, we calculate the worst case difference between the traffics created by the Greedy algorithm and the optimal algorithm, the mean of the difference (μ), and the standard deviation of the difference (σ) during the 100 runs of the simulation programs for each given number of destinations. The two parameters μ and σ are calculated by the following formula:

$$\mu=\frac{1}{100}\sum_{i=1}^{100}(greedy_i-optimal_i)$$

$$\sigma=\sqrt{\frac{1}{100}\sum_{i=1}^{100}[(greedy_i-optimal_i)-\mu]^2}$$

where $greedy_i$ and $optimal_i$ are the traffics generated by the two algorithms, respectively, for i-th run of the algorithms.

Figure 5.5 shows the results, where the dotted, solid, and dashed line pieces represent the worst case differences, the mean, and the standard deviation of the differences, respectively, for the 100 runs.

5.4.3 Decreasing probability routing

A good scheduling algorithm in a hypercube multiprocessor should partition and map the data set to processors in a way that interprocessor communication is minimized. In other words, if the evaluation of one data partition requires information from another data partition, these two partitions should be allocated to adjacent processors [NiKP87]. In this case, those destination nodes are usually close to each other. Decreasing probability routing is a good assumption for this situation [ReFu87]. Under this assumption, the probability that a node sending a message to a destination of distance d decreases as the value d increases. Let $p(l_i)$ be the probability that a node of distance i happens to be a destination node in a multicast. Then, we have $p(l_{i+1})=kp(l_i)$, where $k<1$ is a constant determining how fast the probability decreases as the distance increases. Since there are $\binom{n}{i}$ nodes having distance i from the source node, the following equation holds:

$$\sum_{i=1}^{n} \binom{n}{i} p(l_i) = \sum_{i=1}^{n} \binom{n}{i} k^{i-1} p(l_1) = 1$$

Based on this distribution, the simulation results of the performance for k=0.5 is shown in Figures 5.6 and 5.7. Notice that the simulation is a very rough approximation of the decreasing probability distribution. Since a node can be selected as a destination node only once in a multicast set, when a number of destinations are selected, there is a factor of conditional probability. Consequently, the simulated performance reflects the actual destination distribution only when the number of destinations is very small. When the number of destinations is very large, the distribution is actually close to uniform distribution. As can be seen from Figures 5.6 and 5.7, the performance of the greedy multicast algorithm for descending probability distribution is much better than that of uniform distribution (Figures 5.4 and 5.5) when the number of destinations is small. From Figure 5.7, we can see that the average difference between the traffics of greedy and optimal solutions is less than one unit.

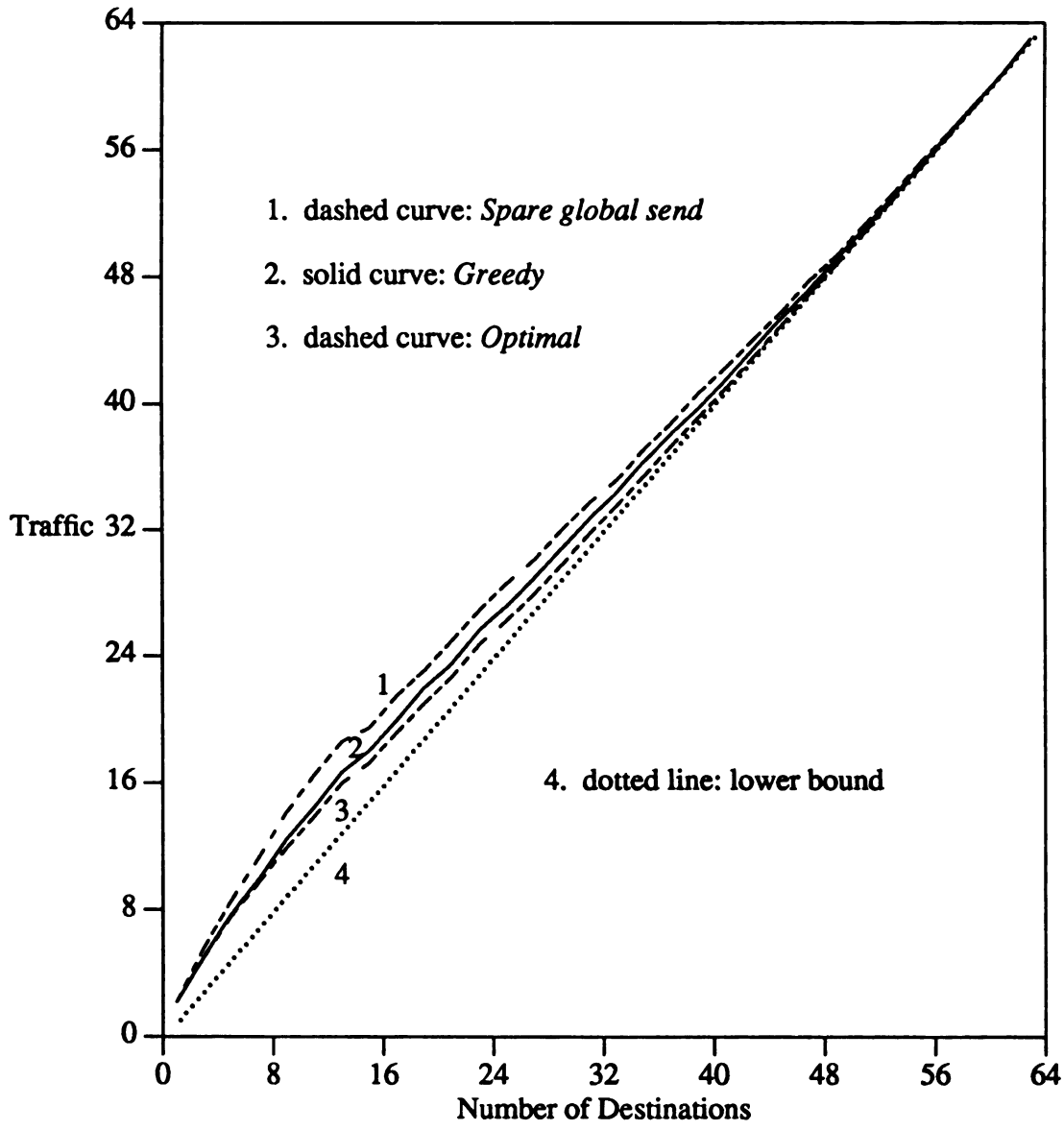


Figure 5.6 Performance comparison of 3 multicast methods

(under decreasing probability distribution)

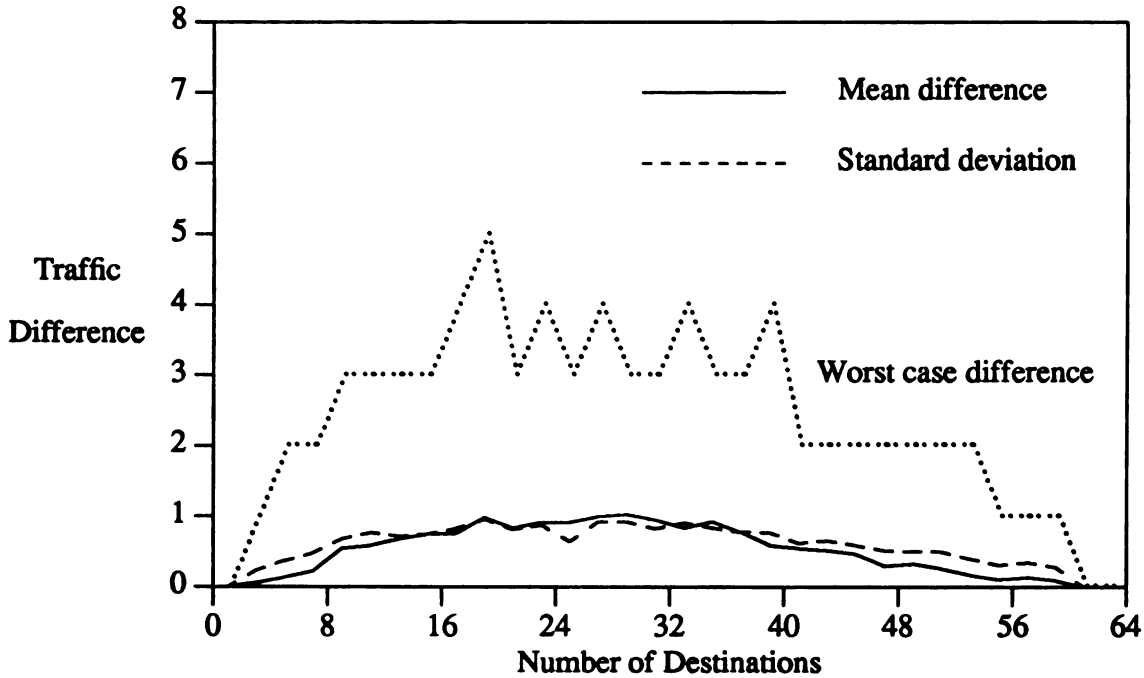


Figure 5.7 Performance comparison of Greedy algorithm with optimal solution

(under decreasing probability distribution)

CHAPTER 6

A HARDWARE ROUTER DESIGN FOR HYPERCUBE MULTIPROCESSORS

As indicated in previous chapters, the interprocessor communication mechanism is an important factor in determining the performance of multiprocessor systems. In first generation hypercube multiprocessors, one-to-one communication is the only service directly supported. In Intel iPSC, broadcast and multicast (called "spare global send") communications are implemented by subroutine calls on top of one-to-one communication [MoSc86]. As discussed in Section 3.7, the software approach for handling interprocessor communication is far too slow to match the speed of the processors.

Various techniques, such as DMA, used in NCUBE [HMSC86], or dedicated communication processor, used in Ametek [Amet86], have been used to reduce the communication overhead. However, since the first generation hypercube multiprocessors adopt the *store-and-forward* approach for message routing, the communication overhead is still unacceptable, especially for communication between non-neighboring nodes. The communication mechanism provided by these machines not only has the potential of generating more than required traffic in the system, but also introduces a great amount of delay due to the time spent to make routing decision.

Therefore, to design hardware devices dedicated to routing purpose has become a natural trend in the second generation DMMP design. In this chapter, we will present the architecture of a hardware router. Each processor in a hypercube is associated with a

router to handle interprocessor communication tasks. We first give some design considerations, then propose a message format to be used in the interprocessor communication. A major part of this chapter is devoted to the detailed discussion on the architecture of the router.

6.1 HARDWARE DESIGN CONSIDERATIONS

Some dedicated router chips have been developed and used to handle one-to-one communication in the second generation of hypercube multiprocessors. Dally and Seitz present the *torus routing chip* [DaSe87] to perform *virtual cut-through* routing. Based on a *virtual channel* method, the chip can provide deadlock-free one-to-one communication for general k-ary n-cube DMMPs. At a speed of 4MHz, the delay introduced at each intermediate node is 150 ns. In Intel's second generation hypercube iPSC/2, a router based on circuit switching is designed for one-to-one communication [iPSC88]. To establish a physical path between a source and a destination, a few microseconds is required at each node along the path. About 350μs is needed to transmit a message and receive an acknowledgement. Furthermore, some designs also include broadcast communication capability using the *wormhole* approach [Seit87].

However, multicast communication, although greatly demanded by various applications, is not directly supported by any existing DMMP. To design a versatile hardware router, which can directly support not only unicast and broadcast, but also multicast, is the major goal of this research. As indicated in Chapter 3, virtual cut-through packet switching is more versatile and better suited for our purpose.

Our goal is to design a VLSI router with the following features:

(1). *Fast*: Even for a complicated multicast message, the router decides a forwarding dimension in 1 μs or less.

(2). *Dedicated*: The router handles all communication tasks independent of local nodal processor. The nodal processor is involved in any communication task only when

the node is a source or a destination.

(3). *Versatile*: The router can efficiently handle any of the three basic communication types — unicast, broadcast, and multicast.

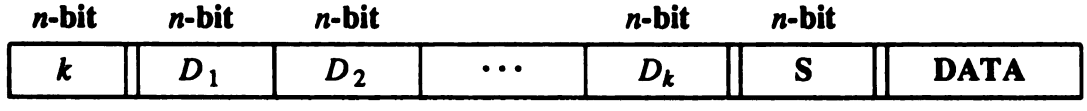
(4). *Distributed*: Routing decision is made by each forward node in a distributed manner. Each router only decides its immediate neighbor(s) to forward received messages.

(5). *Fault-tolerant*: The router should have certain fault-tolerant capability. More precisely, if each fault-free node has no more than one faulty neighbor, the router should be able to forward messages to fault-free destination(s) correctly.

In the remaining sections, we will discuss the architecture of a hardware router following a presentation of the frame format.

6.2. MULTI-DESTINATION MESSAGE FORMAT

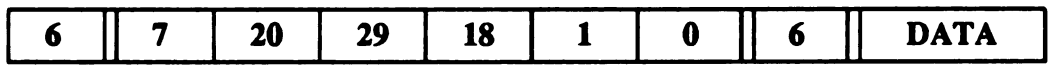
To support multicast communication, the message should carry all destination addresses as well as the address of the source. For broadcast communication, however, only the source address has to be specified. A general *multi-destination message* in an n-cube has the following frame format.



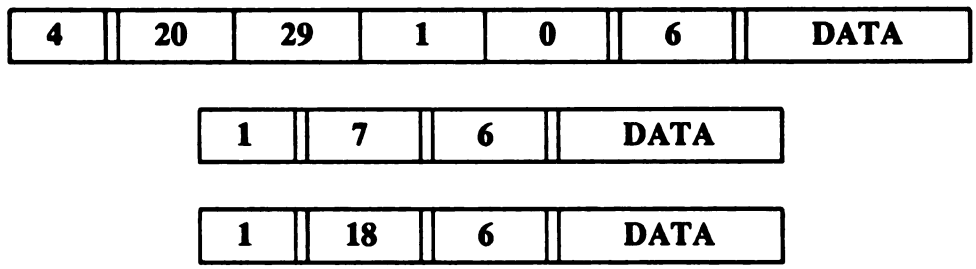
The first field in a message is the k field. For multicast and unicast, the value of k indicates the number of destination fields carried in the message. The k field is immediately followed by the k destination fields: $D_1, D_2, \cdots, D_k$. For broadcast message, however, no destination fields are carried, even though $k=2^n-1$ in that case. Thus, $k=1$ means it is a unicast message, and $k=2^n-1$ indicates a broadcast message. Otherwise, it is a multicast. For simplicity, fields such as start and end delimiters and checksum are not shown

in the message format. Messages may have variable lengths not longer than a maximum limit.

For example, the message generated by the nodal processor at node 6 as illustrated in Section 5.2 has the following format.



Based on the greedy multicast algorithm, the router at node 6 will generate three messages sent to nodes 4, 7 and 2, with the following formats, respectively.



Note that the source field "S" and the data field "DATA" will never be changed on the way to different destinations.

6.3. AN OVERVIEW OF THE ROUTER

In Chapter 4, we mentioned that multicast is the most general type of communication. In a broad sense, multicast could include all three types of communications, assuming $1 \le k \le 2^n-1$. In a narrow sense, it refers to the situation when $1 < k < 2^n-1$. However, in real applications, it is unlikely that a multicast message will be sent to all 2^n-2 possible destinations. Therefore, in our actual design, the hardware unit (*multicaster*) implements multicast algorithm for the case of $1 \le k \le m$ (notice that the unicast is included), where $m \ll 2^n-1$. For the case of $k > m$, the routing will be implemented by broadcast algorithm. From Figure 5.3, we can see that when the number of destinations is large, the traffic generated by broadcast is close to that generated by multicast. Thus, the

choice of m is a tradeoff between the space (hardware cost) and the time (system traffic).

In order to simplify the illustration, we show the diagrams of a hardware router for a 3-cube. In the following discussions, n denotes the dimension of the hypercube (n=3, in our example); m denotes the maximum number of destinations in a multicast the multicaster can handle (m=4, in our example); and k denotes the actual number of destinations in a particular multicast.

Each router in a 3-cube has three input channels from its three neighbors, three output channels to the three neighbors, and one input connection and one output connection to the local nodal processor, as depicted in Figure 2.2. Channel i is the directly connected link between the local node and the neighboring node at dimension i, that is, their binary addresses differ at bit position i only. Each input channel is associated with a *message handling unit* (MHU), through an *input gateway* (I_i for dimension i, $0 \le i \le n$). After processing is done in the router, the incoming message is sent out through one or more of the four *output gateways* (O_j for dimension j, $0 \le j \le n$) to the corresponding neighbor(s) or/and the local nodal processor.

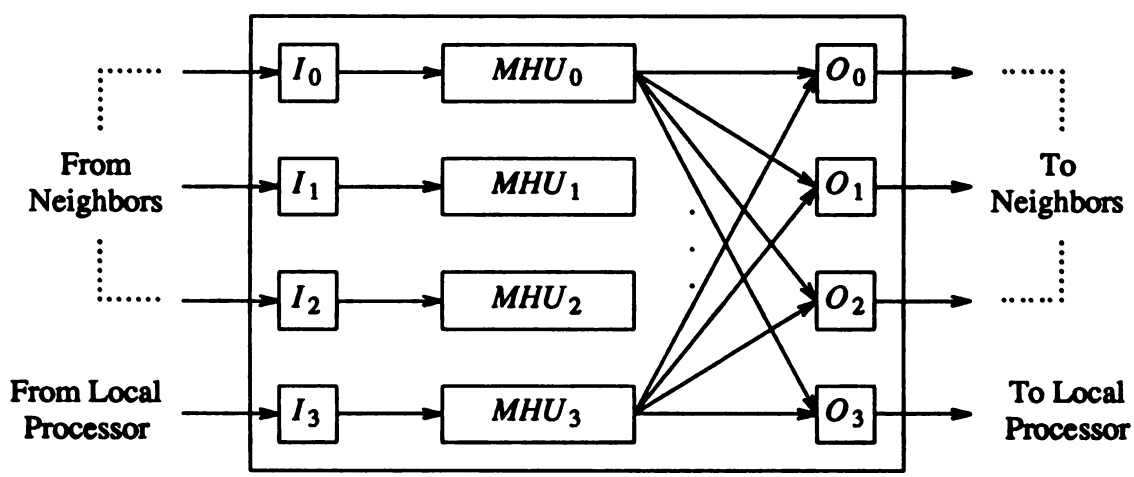


Figure 6.1. The block diagram of a router in a Q_3

In this study, what we are concerned with is the structure and activities in a router attached to the nodal processor. In the following discussions, from the point of view of a router in current node, the term *sending node* is used to represent the router in a neighboring (parent) node which sends a message to the current node or the nodal processor at the local node which originates a message. The term *source node* represents the node which originates a message. Thus, a sending node is not necessarily a source node. Similarly, a *receiving node* is the immediate receiver of a message — the router in a neighboring child node, which is not necessarily a destination node.

Each input gateway provides necessary interface between the channel and the MHU. Each output gateway acts as a multiplexer. An output gateway may receive messages from any of the four MHUs and send them out to the the receiving nodes specified by the MHUs. We will concentrate our discussion on the MHU design since it is the major component in a router.

6.4. MESSAGE HANDING UNIT (MHU)

Figure 6.2 shows the block diagram of a Message Handling Unit (MHU) which consists of the following modules:

(1). A *Controller Module*, which interacts with neighboring and local processors, controls the current state of the MHU, and synchronizes the activities of other modules in the MHU.

(2) A *Processing Module*, which is the essential component — actually executing all communication algorithms.

(3) A *Data Buffer Module*, which provides temporary buffering space for the data field of a message.

(4) An *Input Module*, which acts as a distributor. It sends k field, address fields, and source field to the processing module and sends the source field and data field to the data buffer module.

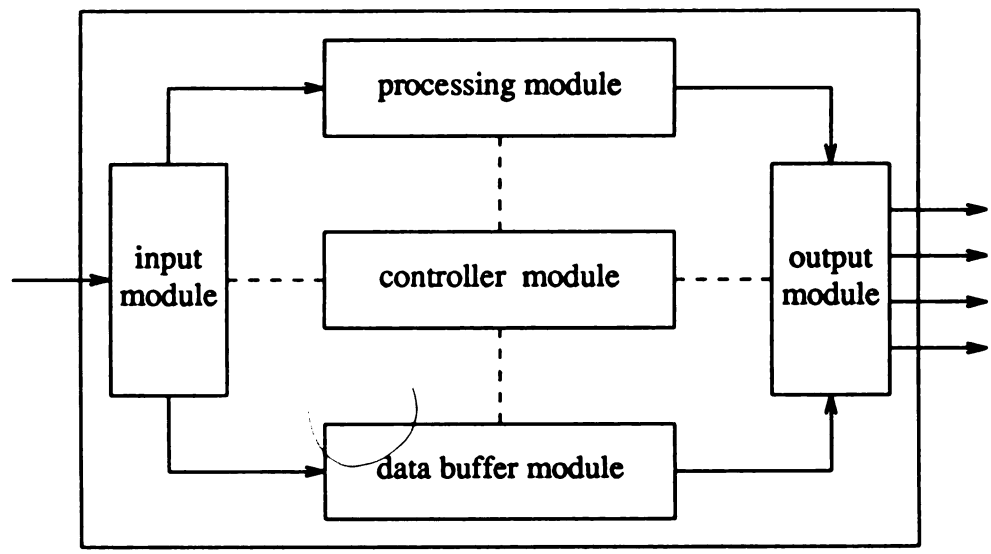


Figure 6.2. The block diagram of an MHU

(5) An *Output Module*, which generates proper output to the output gateways. The output module receives new message header(s) prepared by the processing module and the data field from the data buffer, and reassembles them to form new message(s). The message(s) are then sent to the output gateways according to the relative address(es) of receiving node(s), which are also provided by the processing module.

In the following sub-sections, we will briefly discuss the design of the controller module and the data buffer module with emphasis on the processing module.

6.4.1 The Controller Module

The function of the controller module is to interact with all sending and receiving nodes and coordinate all modules in the MHU by issuing proper control signals. Before a sending node sends out a message, it first sends a *request-to-send* signal to its intended receiving node. When the controller at the receiving node receives the request, it issues

an acknowledgement signal. Upon receiving the acknowledgement, the sender can then send out a message.

The MHU works in the following four states:

WAIT: it is in idle status and is waiting for the next *request-to-send* signal from the sender.

INPUT: The k field, address fields, and source field of an input message from the sender are received. The k field of the message header is checked. In case of $k<2^n-1$, the k address fields are checked to see if the local address is in the destination list.

PROCESSING:

Two things are to be done at this state: (1). Address processing is done at this time. The processing module entails finding proper output port(s) to forward the message. (2). At the same time, the input module sends the upcoming data field to data buffer module.

OUTPUT:During this stage, the new message header(s) is formed and sent out through selected output gateway(s) followed immediately by the data field pumped out from the data buffer module.

6.4.2 The Data Buffer Module

Once a sending node starts sending a message, it will not stop till the entire message has been sent out. When the processing module is making routing decision for the message after the message header is received, the data field of the message will keep coming in. Thus, a temporary buffering space is needed for the data field received during the address processing time. However, it may happen that two or more MHU's in a router want to send messages to the same neighbor at the same time, or a MHU can not receive an acknowledgement from a receiving node. In this case, the entire message may have to be buffered temporarily until the link is available. Thus, for flow control purpose, the capacity of the data buffer should be equal to the size of a maximum length

message, even though only a small part of the buffer is used in usual case.

6.5 THE PROCESSING MODULE

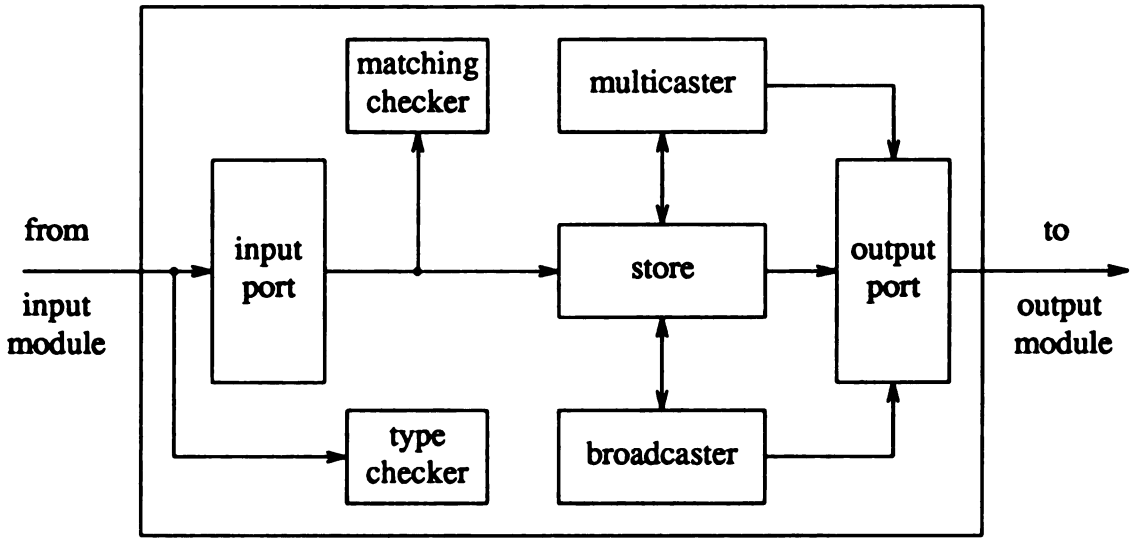


Figure 6.3 The Processing Module

The processing module is the main part in the MHU. As shown in Figure 6.3, it consists of following components: an *Input Port*, a *Store*, a *Broadcaster*, a *Multicaster*, a *Type Checker*, a *Matching Checker*, and an *Output Port*. All the connections between the processing module and the control module for signals are omitted in the figure.

The task of the processing module is to determine routing dimensions (the relative addresses of neighboring nodes) based on the algorithms MULTICAST.1 (described in Section 5.2) and BROADCAST.2 (described in Section 4.3). The relative addresses are directly sent to the output module and are then used to determine which output gateway to send the message.

The multicaster unit will be discussed in Section 6.6, while all the other units are briefly described in this section.

6.5.1 Input Port

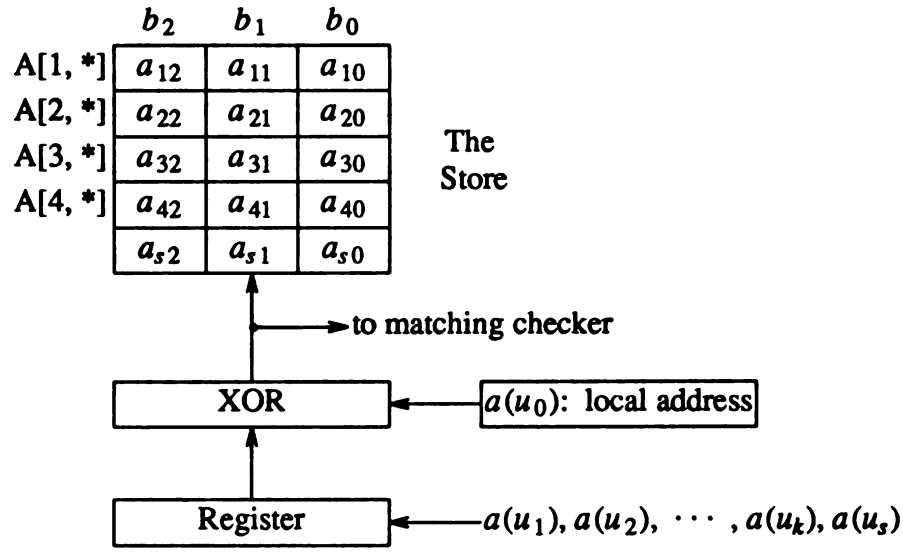


Figure 6.4. The Input port and the Store

The basic function of the Input Port is to convert the actual addresses of the source and all destinations in the message header into relative addresses with respect to the local address and then save them in the Store. As shown in Figure 6.4, the input destination addresses $a(u_i)$ $(1 \le i \le k)$ and source address $a(u_s)$ are first XORed with the local address $a(u_0)$ to form the relative addresses:

$r_0(u_i) \equiv a_{i(n-1)} a_{i(n-2)} .. a_{ij} .. a_{i0} = a(u_i) \oplus a(u_0)$, for $1 \le i \le k$;

Note that the relative source address is also calculated for the purpose of broadcast communication.

6.5.2 The Store

The Store in the processing unit has a $(m+1) \times n$ binary array (5×3 in our example as shown in Figure 6.4), which provides temporary storage space for the relative addresses of the source and up to m destination nodes. Initially, all entries in the array

are set to zeros. A subset of the Store, $A[1..m, 0..n-1]$, which initially holds the m relative destination addresses, is referred to as *reference array*. After each cycle (to be defined laster) of the operation in the MHU, a particular forwarding dimension, and thus a receiving node, is selected. The addresses of a subset of destinations are put into a destination sublist in a forwarding message and sent out along that dimension. The rows corresponding to those destination addresses are then reset to all zeros.

6.5.3 The Type Checker

The first field in an incoming message is the k field, which indicates the communication type. The Type Checker checks the k field in the message header and sends a *type* signal to the Controller. If $k>m$, then *type*=0 and the Broadcaster will be invoked. Otherwise, *type*=1 and the Multicaster will be activated by the Controller. The type check also informs Matching Checker the number of address fields, k, to be checked if $k<2^n-1$.

6.5.4 The Matching Checker

When the relative addresses are sent from Input Port to the Store, they are also sent to the Matching Checker. The function of the Matching Checker is to examine each destination address. Whenever a zero relative address is found, which indicates that the local node is one of the destination nodes, the Matching Checker sends a signal to the controller, so that a copy of the source field and the data field in the incoming message will be sent to the local processor.

6.5.5 The Output Port

The Output Port basically performs the following two functions:

(1) It passes the relative addresses of the receiving node(s) to the output module. Notice that there is exactly one bit position with value 1 in each of the relative addresses. The addresses actually indicate the routing dimensions, or channel *id* numbers. Thus, the output module can easily determine the forwarding dimension

accordingly. Furthermore, the k field of the message header provided by the Multicaster or Broadcaster will also be directly passed through the Output Port immediately following each dimension information.

(2) Similar to the Input Port, the Output Port performs *XOR* operation on the relative addresses in the destination sublist with the local address to form the actual addresses. In broadcast case, the Output Port sends out a relative neighboring address followed by a k field with value zero. In multicast case, the relative address of a neighboring node is followed by a non-zero k field and k actual destination addresses.

Note that the relative address of neighboring node is eventually consumed at output modules, while the k filed and the k following destination addresses are assembled into a new message header with the data field from the data buffer to form a new message.

6.5.6 The Broadcaster

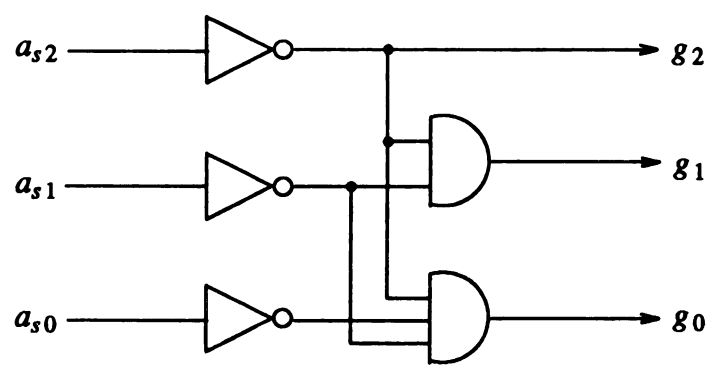


Figure 6.5 The Broadcaster for a fault-free Q_3

The Broadcaster is invoked for a broadcast communication ($k=2^n-1$) or a multicast communication with a large number of destinations ($m<k<2^n-1$). If $k=2^n-1$, a copy of the received message is sent to local processor. Otherwise, a copy of the received message is sent to the local processor only when the local address matches one of the

addresses in the destination list. The Broadcaster performs the broadcast algorithm discussed in Section 3.4 to decide further message forwarding. Assume the relative source address with respect to the local address is $a(u_s)=a_{s(n-1)}a_{s(n-2)}\cdots a_{s1}a_{s0}$. In the Broadcaster, a flag word $G=g_{n-1}g_{n-2}\cdots g_1g_0$ is generated first, such that

$$g_j = \begin{cases} \overline{a_{sj}} & \text{if } j=n-1 \\ \bigcap_{all\ p\geq j} \overline{a_{sp}} & \text{if } 0\leq j\leq n-2 \end{cases}.$$

where (and from here on) the symbols $\cup$ and $\cap$ stand for the logical OR and logical AND operations, respectively, for a number of bits. If $g_j=1$, then the received message is sent out through dimension j. Actually, the relative address of a receiving node with respect to the local node, a binary number which has value 1 at bit position j and value 0 at all other bit positions, is sent through the Output Port to the Output Module to decide proper output channel (dimension). The broadcaster can be easily constructed as shown in Figure 6.5.

6.6 THE MULTICASTER DESIGN

The Multicaster is the most complicated part of the MHU, which implements the greedy multicast algorithm. It calculates all n column sums of the relative destination addresses and finds out one with the maximum sum. A brute force approach using a number of counters and comparators will take too much space and time. In our approach a combinational circuit is designed to find out the current maximum bit position in one cycle. In other words, the determination of each outgoing message takes one cycle, where the cycle time is measured as the worst case signal delay in the combinational circuit, which is 1 μsec in our prototype design based on a 3 micron CMOS technology.

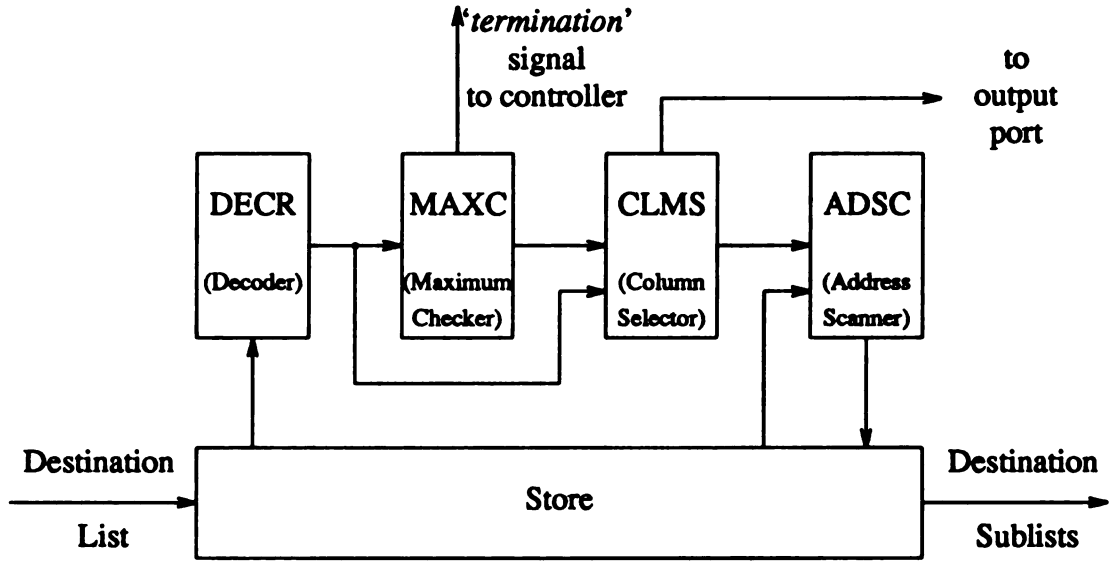


Figure 6.6 The Multicaster and the Store

Figure 6.6 shows the block diagram of a Multicaster and the Store. A Multicaster consists of a *Decoder* (DECR), a *Maximum Checker* (MAXC) to check the maximum number of 1's in the columns, a *Column Selector* (CLMS) to select a particular column which has the maximum number of 1's, and an *Address Scanner* (ADSC) which picks up all rows that have the value 1 at the selected column to form new address fields.

6.6.1 The Decoders (DECR)

There are n decoders in an MHU, each associated with a column. All the entries of a column in the reference array are directly connected to the inputs of a DECR. As shown in Figure 6.7, a DECR has m input lines and $m+1$ output lines. The m input lines to the $DECR_j$ are a_{ij} $(1 \le i \le m)$, which are the m elements (bits) of the j-th column in the reference array, $A[*,j]$. The m input lines are sent to a *Wallace tree* to get $\log_2 \lceil m+1 \rceil$ bits (lines) representing the number of 1's in the m input lines, followed by a decoder to generate $m+1$ output lines c_{jl} $(0 \le l \le m)$. The DECR circuitry counts the number of 1's in

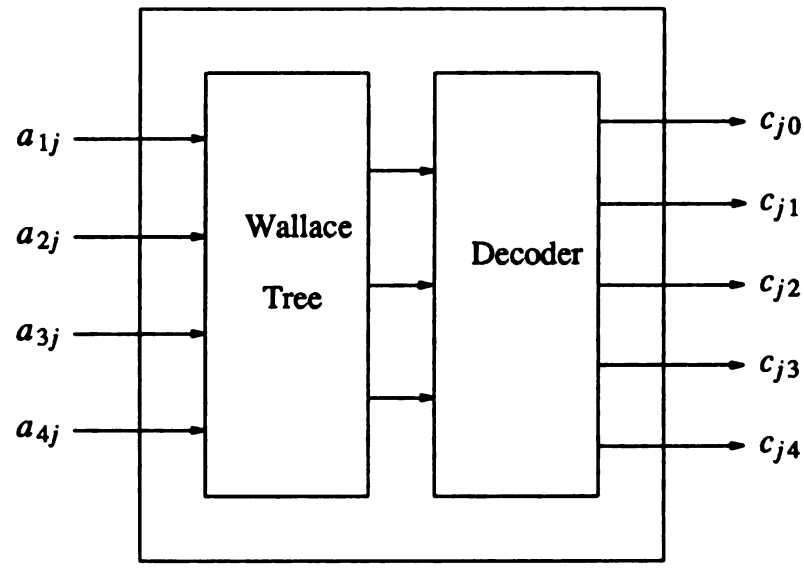


Figure 6.7 The Decoder (DECR) for the j-th column

all m input lines. Output line c_{jl}=1 implies that $\sum_{i=1}^{m} a_{ij}=l$. For any input combination, there is one and only one output line which has value 1.

After each cycle, those rows which have been removed to a destination sublist are reset to zeroes. Thus, c_{jl}=1 means $\sum_{i=1}^{m} a_{ij}=\sum_{i=1}^{k} a_{ij}=l$, which indicates that column j in the **current** reference array (i.e., at bit position j of all remaining binary relative address addresses) has l 1's.

6.6.2 Maximum Checker

After the DECRs find the number of 1's in all columns, the outputs from the DECRs are sent to a *Maximum Checker* (MAXC) to find the maximum number of 1's in all columns. Figure 6.8 is a logic diagram of the MAXC. The MAXC has $(m+1)\times n$ inputs, which are the outputs of the n DECRs, c_{jl} $(0\le j\le n-1,\ 0\le l\le m)$, and $m+1$ outputs, s_l $(0\le l\le m)$.

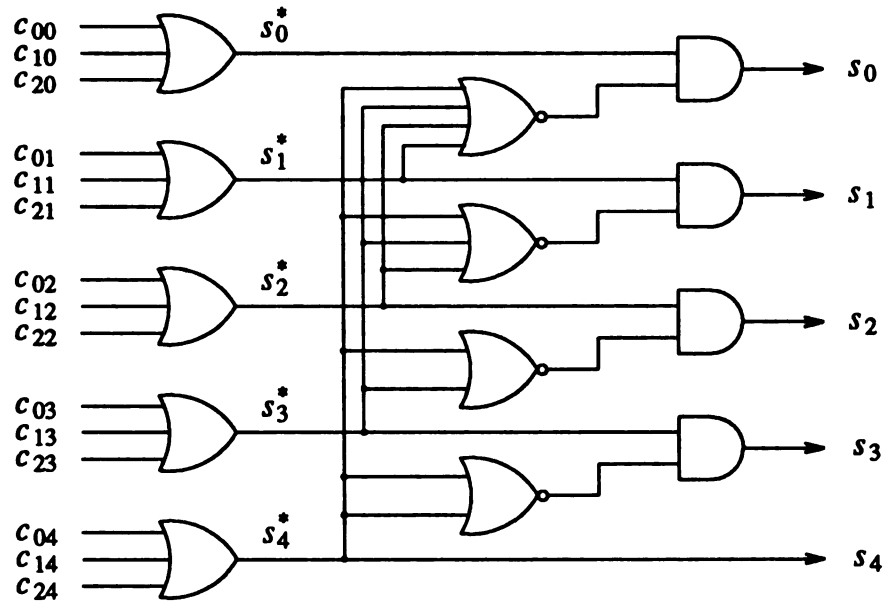


Figure 6.8 The Maximum column Checker (MAXC)

The intermediate values of the circuitry can be expressed as:

$$s_l^* = \bigcup_{j=0}^{n-1} c_{jl}$$

where $s_l^*=1$ means some column (at least one) has l 1's.

The outputs are:

$$s_l = \begin{cases} s_l^* & \text{if } l=n-1 \\ s_l^* \cap \left(\overline{\bigcup_{all\ p>l} s_p^*}\right) & \text{if } 0 \le l \le n-2 \end{cases}$$

The above logic ensures that there is one and only one of the $m+1$ output lines which has value 1. Furthermore, $s_l=1$ means that the maximum number of 1's in the columns is exactly l.

A special case is that s_0=1, which means that all columns have zero number of 1's. That is, the current reference array is all zeroes. In other words, all destination addresses have been resolved and the routing of the current message has been done. Thus, the output line of s_0 is used as a *termination* signal sent to the Controller.

6.6.3 The Column Selector

After finding the value of maximum number of 1's in the columns, the next step is to find which column has that maximum number. As pointed out in the discussion of greedy algorithm, more than one column may have the same maximum number of 1's. The greedy algorithm requires only one column be selected. To simplify our design, we select the column with the smallest column index among those columns. The Column Selecter (CLMS) is designed for this purpose. As shown in Figure 6.9, it has $(m+1)\times 2\times n$ inputs and n outputs.

The function of the CLMS can be expressed in the following equations:

$$y_j^* = \bigcup_{l=0}^{m} (c_{jl} s_l) \qquad \text{for } 0 \le j \le n-1$$

$y_j^* = 1$ means column j has the maximum number of 1's. To make the selected column unique, we let

$$y_j = \begin{cases} y_j^* & \text{if } j=0 \\ y_j^* \cap \overline{\bigcup_{all\ p<j} y_p^*} & \text{if } 1 \le j \le n-1 \end{cases}$$

Exactly one of the n outputs from the CLMS has value 1, and y_j=1 means column j has been selected. Then, all the remaining destinations whose relative addresses have value 1 in bit position j will receive the message through the j-th dimensional neighbor of the current node.

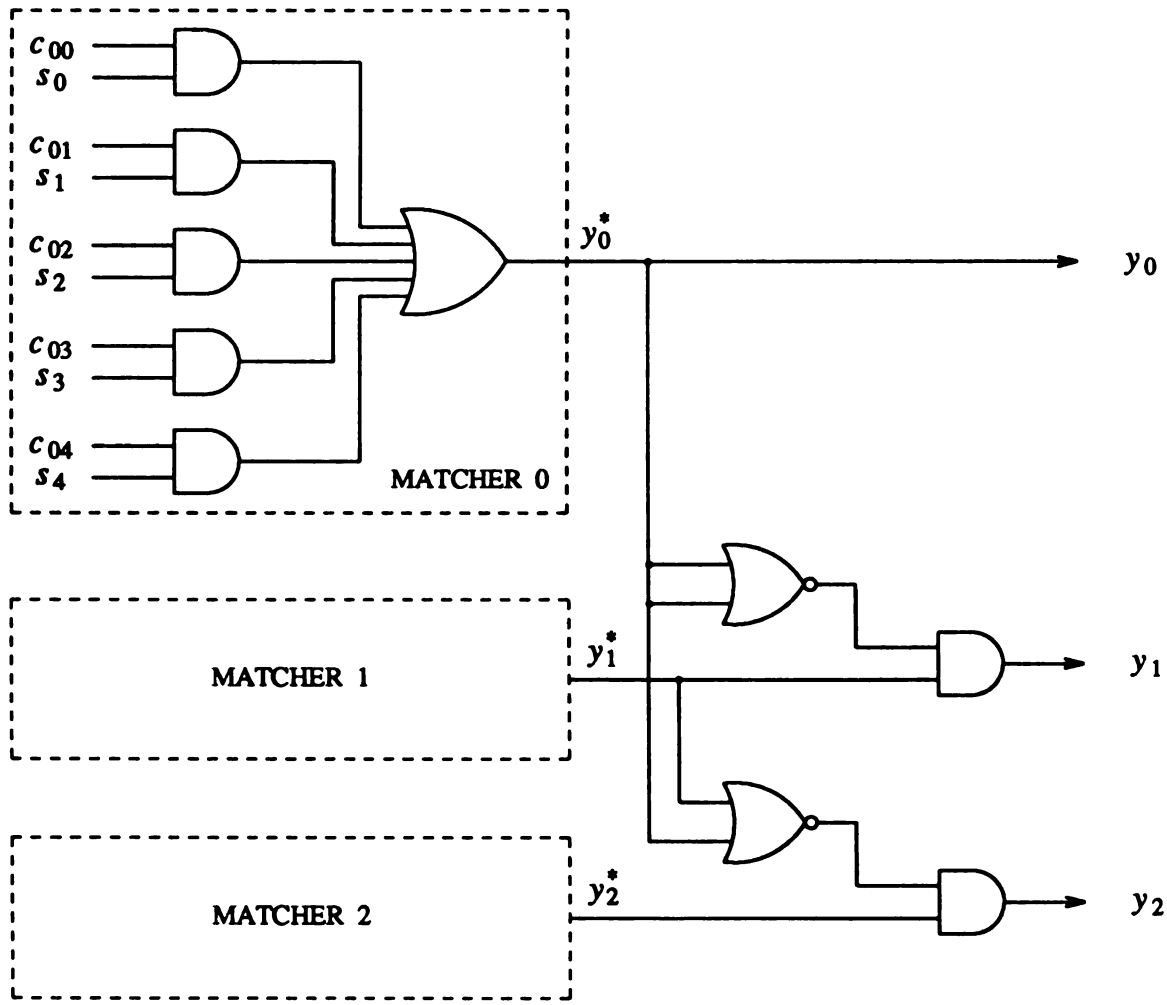


Figure 6.9. The Column Selecter (CLMS)

6.6.4 The Address Scanner

The n outputs from the CLMS selector are used to scan the reference array. That is, each row in the reference array, $(a_{i(n-1)}a_{i(n-2)} \cdots a_{i0})$ for $1 \le i \le m$, is compared with the vector $(y_{n-1}\ y_{n-2}\ \cdots\ y_0)$.

$$r_i = \bigcup_{j=0}^{n-1} (a_{ij} y_j) \qquad \text{for } 1 \le i \le m.$$

If r_i=1, then row i will be put into a destination sublist, and all the entries in that row will be set to zeros.

6.7 THE PROTOTYPE MHU CHIP

In this subsection, we summarize the operation of the MHU and the prototype design of a MHU. As soon as the Store has received all the relative destination addresses, the first receiving node will be determined after one cycle. As shown in previous sub-sections, the selection of a particular destination sublist is determined by a number of logic equations which are implemented by a combination circuitry. Thus, the cycle time is measured from the time the Store is updated until a destination sublist is selected. This procedure is repeated until a termination signal is received (i.e., the elements of the reference array in the Store become all zeroes).

A prototype of the MHU has been designed by a group of students taking CPS922 course in Winter 1987 at Michigan State University [DCCM87]. It is a simplified version of the MHU chip presented in this paper, which is for a 3-cube and does not perform broadcast algorithm (without Type Checker and Broadcaster). The chip has been fabricated by MOSIS based on a 3 micron CMOS technology. To shorten the design cycle, the Controller Module is designed using a PLA approach. Figure 12 illustrates the photograph of the chip. The cycle time is measured to be 1 μsec.

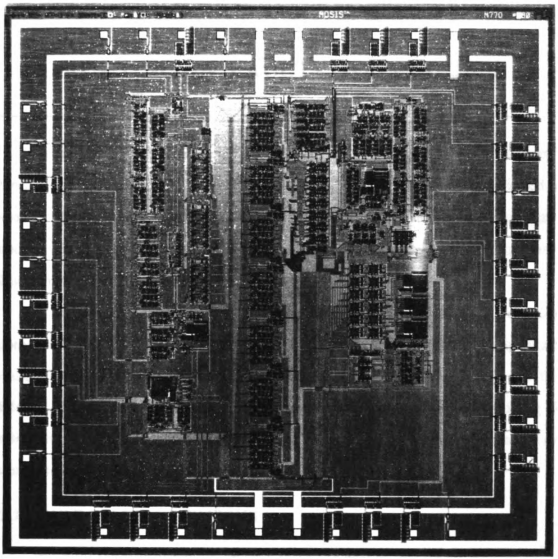


Figure 6.10 The prototype MHU chip

CHAPTER 7

ROUTING IN FAULTY HYPERCUBES

Hypercube multiprocessors are highly complex systems consisting of as many as thousands of processors interconnected through a hypercube topology. As indicated previously, the hypercube topology provides multiple routing paths between every pair of nodes. All message routing algorithms for hypercube multiprocessors are based on the Hamming code of the node addresses.

As the system becomes larger, however, the probability that some processors and communication links fail increases. When the failure of some components happens, the routing mechanism based on the Hamming code cannot be applied. Design of fault-tolerant routing mechanism is important especially for those applications requiring high reliability. This chapter addresses the issue of message routing in faulty hypercube multiprocessors.

7.1 FAULT-TOLERANT SYSTEMS

In a fault-tolerant system, in order to reach the goal of reliable computing, first the system should be able to detect the presence of failures and identify faulty components. Once the failures are located, explicit mechanisms for dealing with the effects are invoked. The process can be categorized as follows [Kim79, KuRe86]: (1) fault detection, (2) fault diagnosis (location), (3) system reconfiguration or repair, and (4) system recovery.

The first two steps are basic steps to locate the source of failure at appropriate level (typically node level in our case). Fault detection and location problem in hypercube multiprocessors have been studied in the literature [ArGr81, Bhat83, Hawk85, Kuhl80].

After faulty components have been detected and located, the third step involves physical reconfiguration of the system components around the faulty components, or more likely, logical relocation of the load of faulty nodes among other nodes. There are two approaches to achieve system reconfiguration [Aviz76]:

(1). Providing some spare hardware resource. The spare parts of the system either take part in the computing process or are in standby condition, ready to act automatically to preserve undisrupted continuation of the system. For up to a certain degree of failure, the system can retain its full computing capacity.

(2). Providing no spare components. A system achieves *partial* fault-tolerancy (''fail-soft'', ''gracefully degrading'') by reducing its full computing capacity and ''shrinking'' to a smaller system.

For the first approach as proposed in [Renn86], an extra subset of processors are extended to a hypercube and reachable through crossbar switches by each node at the $(n+1)$-th dimension (an extra port in each processor) in order to achieve fault tolerance. In this approach, the system can maintain its full computation capacity under any single fault. Some researchers have worked on the second approach which will be discussed in more detail later.

The last step involves restoring data and computations in the system to a consistent state. This may require recovery schemes such as rolling back computations to a pre-failure state and restarting.

We will concentrate our attention on the reconfiguration step, especially, on partial fault tolerancy in hypercube. More precisely, after the failure of one or a few processors are located, how can we route messages among those fault-free nodes in such a faulty environment?

7.2 DESIGN CONSIDERATIONS FOR FAULT-TOLERANT ROUTING

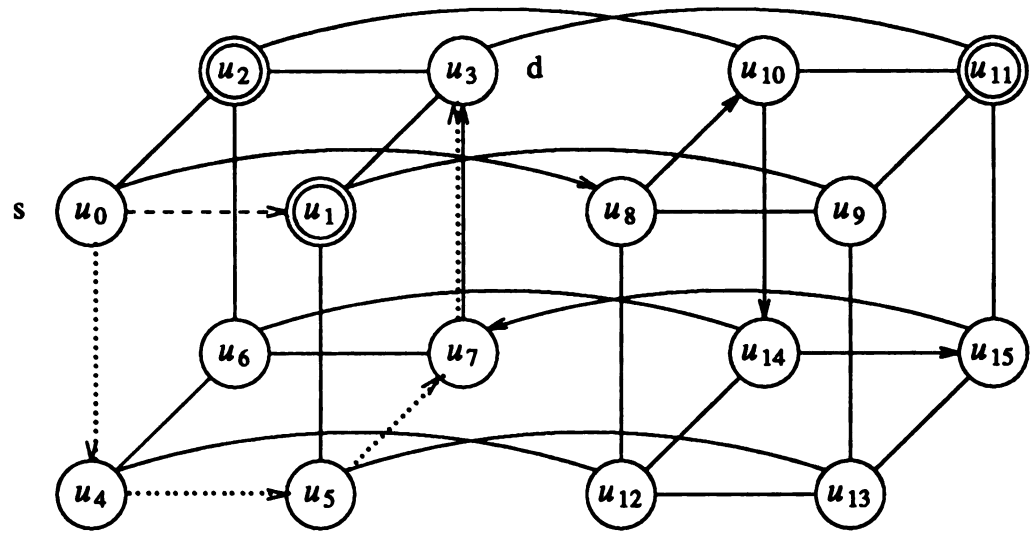


Figure 7.1 An example of routing in a faulty Q_4

Let us first demonstrate the difficulty of message routing in a faulty hypercube. Consider a Q_4 as shown in Figure 7.1. Suppose that nodes u_1,u_2 and u_{11} are faulty (double circled) and the source node u_0 wants to send a message to u_3. We examine the following three cases:

(A). Node u_0 does not know that u_1 is faulty. It tries to send a message to u_1 (dashed arrow) in order to pass the message to u_3, then, the message will get lost.

(B). Node u_0 knows that two of its neighbors u_1 and u_2 are faulty. It may send the message to u_8, u_8 then pass the message to u_{10}, and so on, resulting in a path of length 6 (u_0,u_8,u_{10},u_{14},u_{15},u_7,u_3), formed by the solid arrows.

(C). Node u_0 knows that u_1 and u_2 are faulty. It sends the message to u_4, then u_5, and so on, resulting in path (u_0,u_4,u_5,u_7,u_3), indicated by the dotted arrows, which is optimal with the shortest possible length of 4.

Note that case (A) can be avoided by proper fault diagnosis (detection-location) techniques. For example, through a reliable communication protocol, such as the acknowledgement and time-out scheme, the failure of u_1 and u_2 can be known to u_0. However, case (B) is more difficult to avoid.

Situation for broadcast and multicast is even more complicated. The question is what information the fault-free nodes have to know and how they can guide the message through the shortest possible path(s) to the destination(s). Let us first introduce some new notations, and then discuss what parameters should be considered in our study of fault-tolerant routing.

In a Q_n, the h-neighborhood of a node $u \in V(Q_n)$ is defined to be the set of nodes $H_u^h = \{v \mid d_{Q_n}(u,v) \le h,\ v \in V(Q_n)\}$. Let $F(Q_n) \subset V(Q_n)$ be the set of faulty nodes in Q_n. Also, let $F_h(u) = F(Q_n) \cap H_u^h$ be the subset of faulty nodes which are in the h-neighborhood of node u. A path from a source to a destination is a *feasible* path if it contains no faulty nodes.

The values for the following parameters should be determined in our discussion of fault-tolerant routing algorithms.

1. The *neighborhood radius h*. Suppose each fault-free node $u \in V(Q_n) - F(Q_n)$ keeps the status information of every node $v \in H_u^h$. Then what is the value of h? Two special cases are $h=1$ (u has the status information of neighboring nodes only) and $h=n$ (u has the global status information);
2. $Max(|F(Q_n)|)$. The maximum number of faulty nodes allowed in the system, such that the routing algorithms still work; and
3. $Max(|F_h(u)|)$. The maximum number of faulty nodes allowed within the h-neighborhood of a fault-free node u, for each $u \in V(Q_n) - F(Q_n)$, such that the routing algorithms still work.

The problem of routing in faulty hypercubes has been studied in literature [Hawk85, LeHa88]. Hawkes [Hawk85] addresses the fault diagnosis and one-to-one

fault-tolerant routing problems in generalized hypercube environments (r-ary n-*cube*). There are $(r-1)n$ node disjoint paths between any two nodes in an r-ary n-*cube*. These paths are divided into three *types*. If the distance between the source and the destination is t, then t of them have length t (*type*1), $(n-t)(r-1)$ of them have length $t+2$ (*type*2), and $(r-2)t$ of them have length $t+1$ (*type*3). When some nodes are faulty, the algorithm tries to find a shortest possible path which is feasible.

Lee and Hayes [LeHa88] introduce the concept of *unsafe* nodes to identify those fault-free nodes which may cause communication difficulties in faulty hypercubes. Based on the concept, algorithms for one-to-one and broadcast communication in faulty hypercubes for different values of h are proposed.

Katseff [Kats88] describes algorithms for one-to-one and broadcast communication in *incomplete hypercubes*, which are structures similar to hypercubes but consisting of an arbitrary number (N) of nodes where N is not necessarily a power of 2. The nodes have the addresses of 0 to N-1 interconnected by the same criterion as that of hypercube topology. That is, two nodes have a direct connection, if and only if their binary addresses differ at exactly one bit position.

For a highly reliable system, we do not expect many components to fail at the same time. However, one or a few components may fail at a time. Therefore, we make the following assumptions for the faulty hypercube multiprocessor under consideration.

1. We consider node failures only. A node is said to be faulty if its corresponding processor fails. When a node is faulty, all the links incident to that node are also faulty. Thus, they can be equivalently considered to have been removed from the system.
2. All nodes are in one of the two statuses: either faulty or fault-free. The source node and destination node(s) are all fault-free.
3. The failure of any node is known by all its neighboring nodes. That is, a fault-free node knows the status of each of its neighboring node (h=1). An n-bit vector

$S_f=f_{n-1}\cdots f_1 f_0$, called *Fault_Vector*, is maintained at each fault-free node. Each bit value in *Fault_Vector* indicates the status of a unique neighboring node, that is, $f_i=1$ indicates the neighboring node at dimension i is faulty.

4. Each fault-free node has at most one faulty neighboring node ($F_1(u)\leq 1$, for $u\in V(Q_n)-F(Q_n)$). This also implies $||S_f||\leq 1$.

If condition 4 is not satisfied, that is, a fault-free node finds two or more of its neighboring nodes are faulty, it should report the situation to the host, and then the system should be turned off to repair.

Now, what is the value of $F(Q_n)$? We first find an upper bound for $Max(|F(Q_n)|)$ regarding condition 4 in the above model.

Clearly, if $|F(Q_n)|>2^{n-1}$, condition 4 will not be satisfied since there must exist a node $u\in V(Q_n)-F(Q_n)$ such that u is adjacent with at least two nodes in $F(Q_n)$. Thus, $Max(|F(Q_n)|)\leq 2^{n-1}$, and the equality occurs when $F(Q_n)$ induces an $(n-1)$-cube.

However, if in addition to condition 4, it is required that no two faulty nodes be adjacent in Q_n, then the problem of finding the upper bound of $Max(|F(Q_n)|)$ is reduced to the following classical question in the theory of *error-correcting code*: "What is the largest group of n-bit binary vectors (codewords) such that the Hamming distance of any pair of codewords in the group is at least d?". If the size of the largest such group is denoted by $|C|$, then,

$$|C|\leq \frac{2^n}{\binom{n}{0}+\binom{n}{1}+\cdots+\binom{n}{t}}. \tag{7.1}$$

where $t=\lceil (d/2-1)\rceil$.

The above bound is known as the Hamming bound [Hamm50]. Clearly, with the above modification of condition 4, the Hamming bound gives an upper bound for $Max(|F(Q_n)|)$ with d=3. This implies

$$Max(|F(Q_n)|)\leq \frac{2^n}{n+1}. \tag{7.2}$$

For n=3,4,5,6,7,8,9,and10, the floors of the right hand side of Eq. (7.2) are calculated to be 2,3,5,9,16,28,51, and 93, respectively.

The above modification of condition 4 is material by the fact that when selecting a pair of nodes in random from $V(Q_n)$, the probability that the two nodes are adjacent is very small. For this reason, we consider the Hamming bound as a representative of the upper bound for $Max(|F(Q_n)|)$.

In order to have a quantitative idea about the fault model, we have run a simulation program to estimate the probability that condition 4 above is valid under different number of fault nodes. Figure 7.2 shows the results for 6-dimensional to 10-dimensional hypercubes, where the x-axis is the number of faulty nodes in the system and y-axis is the probability that the fault model is valid for the number of faulty nodes.

In the rest of this chapter, we first review some existing algorithms for unicast and broadcast and then provide the fault-tolerant revisions of the algorithms which can be easily implemented in hardware. Also, modification to the multicast algorithm presented in Chapter 5 will be presented to allow certain fault tolerance. The hardware implementation problem is always the major concern of this study. In Section 7.4, the discussion is focused on how the hardware router design presented in Chapter 6 can be modified so that it can handle all three types of interprocessor communication in faulty hypercubes.

If the above assumption is violated, that is, a fault-free node has more than one faulty neighbor, then the routing problem becomes more complicated. Section 7.5 briefly discusses this situation to see how we can deal with it.

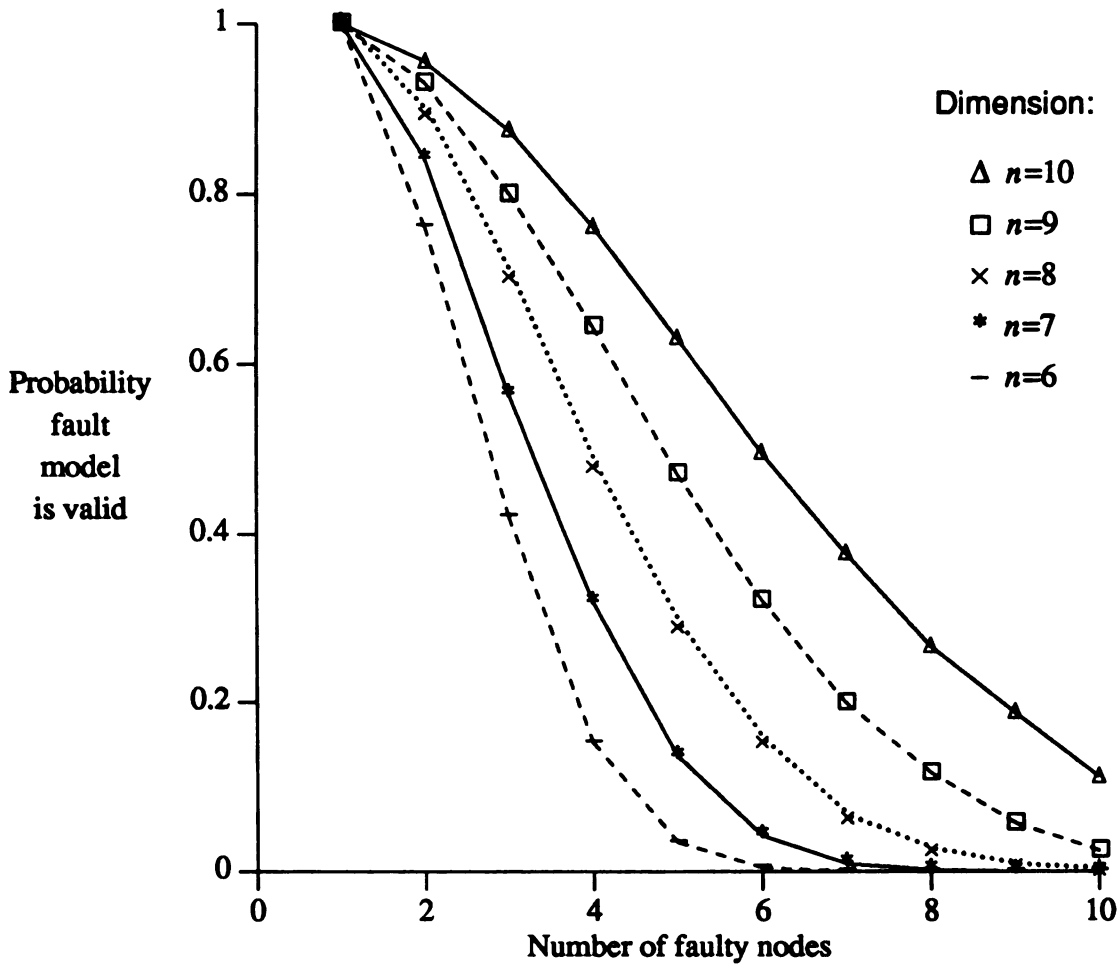


Figure 7.2 Simulation results of the faulty model

7.3 FAULT-TOLERANT ROUTING ALGORITHMS

Based on the model of faulty hypercube stated in Section 7.2, in this section, we first investigate how one-to-one message routing can be achieved. Then, we discuss how broadcast and multicast communications, which are more complicated in faulty hypercube, are handled.

7.3.1 Unicast in Faulty Hypercube

We start our discussion with unicast communication in a faulty hypercube under the above assumptions. Algorithm UNICAST.F1 (shown in Fig 7.3), a simple variant of algorithm UNICAST.1 of Section 4.3, will find a shortest path between any two fault-free nodes in a Q_n. The basic idea is that, similar to the case of fault-free hypercube, each node, upon receiving a message, first tries to send a message to the dimension corresponding to the right most bit position at which the addresses of current node and the destination node differ. If that dimension happens to be faulty, the node then sends the message to the second right most one. The algorithm is to be executed at every forward node along the path.

Theorem 7.1: Given two fault-free nodes u_s and u_d with distance $d(u_s,u_d)=d$ in a Q_n, algorithm UNICAST.F1 always finds a shortest path of length d from u_s to u_d, provided that every fault-free node in the Q_n has at most one faulty neighboring node.

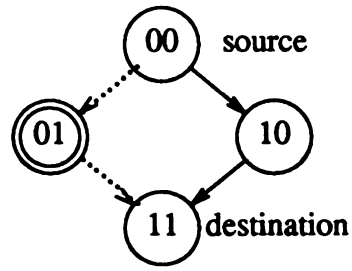
Proof: Since the distance between u_s and u_d is d, their addresses $a(u_s)$ and $a(u_d)$ differ at exactly d bit positions, and d disjoint paths between the two nodes can be constructed. As discussed in Section 4.3, if the system is fault-free, starting from u_s, by traversing through each of the d dimensions exactly once, the message will reach u_d in d steps. The order of which dimension is traversed first is immaterial. In a faulty hypercube, at any forward node u_j, suppose the distance $d(u_j,u_d)=g$. If $g\geq 2$, the algorithm tries to find a fault-free dimension among the g dimensions at which $a(u_j)$ and $a(u_d)$ differ, to pass the message. Since each fault-free node has at most one faulty neighbor,

```
Algorithm UNICAST.F1:
(* h=1; |F1(u)|≤1 for all u∈V−F *)
begin
  (* find the relative destination address with respect to the local address *)
  r0(ud) := a(u0)⊕a(ud);
  if (r0(ud)=0) then
     send message to local processor
  else
     i:=0;
     repeat
        if dimension Wi(r0(ud)) is fault-free then
           send message to dimension Wi(r0(ud))
        else i:=i+1;
     until message is sent;
end.
```

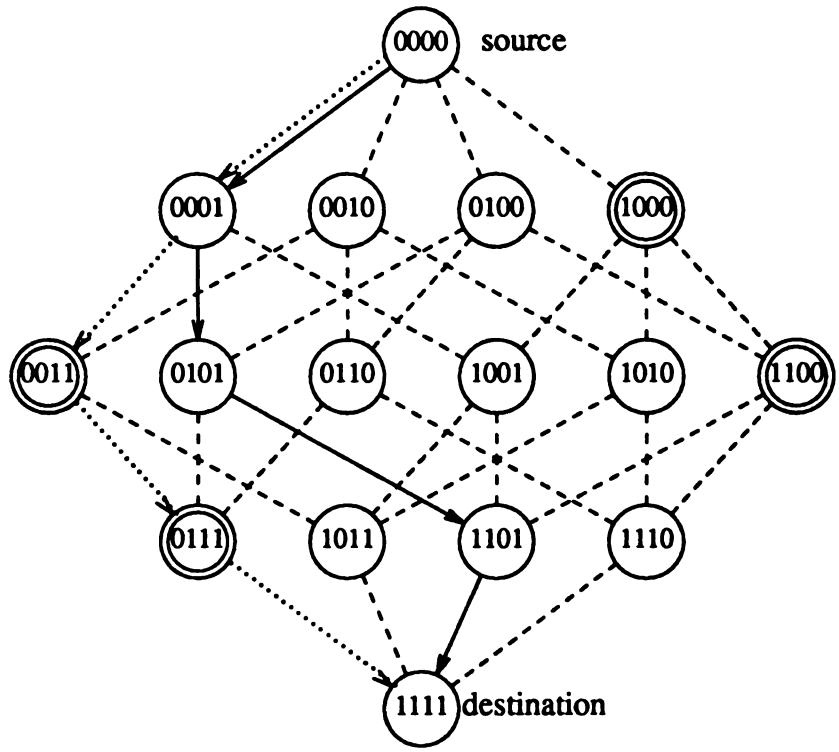
Figure 7.3 A unicast algorithm for faulty hypercubes

such a fault-free dimension always exists. If g=1, then u_d is a neighbor of u_j, and u_d is supposed to be fault-free, u_j will pass the message to u_d through the direct link between them. Therefore, a shortest path from u_s to u_d can always be found. ■

Figures 7.4 shows examples of unicast in a faulty 2-cube and a faulty 5-cube, where double circled nodes represent faulty nodes. The dotted lines form the path in fault-free hypercube found by algorithm UNICAST.1, while the solid lines form the path in faulty hypercube found by UNICAST.F1. In Figure 7.4 (b), 0000 is the source node, and 1111 is the destination node. The source first sends a message to node 0001. At node 0001, the algorithm first tries to pass message to node 0011 at dimension 1 ($W_1(0001 \oplus 1111)$=1), which, however, is found to be faulty. Thus, dimension 2 ($W_2(0001 \oplus 1111)$=2) is selected, and the message is then forwarded to node 0101. At 0101, the algorithm tries dimension 1 again ($W_1(0101 \oplus 1111)$=1), which, unfortunately, is found to be faulty again. Having had to select dimension 3



(a) In a 2-cube



(b) In a 4-cube

Double circled nodes represent faulty nodes
Dotted lines: path in fault-free hypercube
Solid lines: path in faulty hypercube

Figure 7.4 An example of unicast in faulty hypercubes

($W_2(0101 \oplus 1111)=3$), the message reaches node 1101. Finally, 1101 passes the message to the destination 1111 through dimension 1.

7.3.2 Broadcast in Faulty Hypercubes

In this subsection, we consider broadcast in faulty hypercubes. Let us compare the two broadcast algorithms: BROADCAST.2 and BROADCAST.3, presented in Section 4.3. Algorithm BROADCAST.2 decides routing based on the local and source addresses only. It does not seem modifiable to work in a faulty hypercube. Algorithm BROADCAST.3, however, calculates and sends different *Control* vectors to different dimensions at each forward node. Based on this approach, we develop a fault-tolerant algorithm BROADCAST.F3, which can correctly broadcast a message to all fault-free nodes in a fault hypercube. The algorithm is listed in Figure 7.5, where u_s refers to source node of the broadcast, u_0 refers to the local node which is currently executing the algorithm, *Control** is the control vector received by u_0, and $Control_j$ is the control vector sent out from u_0 to dimension j. Note that the major difference between Algorithm BROADCAST.F3 and BROADCAST.3 is in line 8. We prove the correctness of the algorithm in Lemma 7.1 and Lemma 7.2.

Lemma 7.1: Given nodes $u_s, u_d \in V(Q_n)-F(Q_n)$, algorithm BROADCAST.F3 finds a shortest path from u_s to u_d in exactly the same way as UNICAST.F1 does, which has been proven to be a shortest path between u_s and u_d.

Proof: Let the path from u_s to u_d constructed by Algorithm UNICAST.F1 be $p=(u_s, u_1, \cdots, u_i, \cdots, u_{d-1}, u_d)$. We note the following two facts about algorithm BROADCAST.F3. (1) Message is sent only to the dimension whose corresponding bit position has value 1 in *Control** and is fault-free; (2) For any bit position l, if $Control^*[l]=0$, then $Control_j[l]=0$ for any outgoing link j.

Algorithm BROADCAST.F3:

```
(* h=1; |F_1(u)|≤1 for all u∈V−F *)
1. begin
     (* if u_0=u_s, the router get a Control* vector with all bits set to 1 *)
     (* otherwise, u_0 receives a Control* vector from a parent node *)
     (* f_b=1, if dimension b is faulty, otherwise f_b=0 0≤b≤n−1 *)
2.   for j:=0 to n−1 do
3.     begin
4.       send message to local processor;
5.       if Control*[j]=1 and f_j=0 then
6.         begin
             (* form Control_j vector *)
7.           for b:=0 to n−1 do
8.             if Control*[b]=1 and ( b>j or f_b=1 )
                 then Control_j[b]:=1
9.             else Control_j[b]:=0;
10.          send message with Control_j to dimension j
11.        end;
12.    end;
13. end.
```

Figure 7.5 A broadcast algorithm for faulty hypercubes

We prove by induction on i that algorithm BROADCAST.F3 reaches u_i along path $(u_s, u_1, \cdots u_{i-1})$ and that for each bit position x at which $a(u_i)$ and $a(u_d)$ differ, *Control**$[x]$ at node u_i has value 1.

For i=1, the basis step of the induction, obviously u_1 receives messages from u_s. At u_s, all bit positions of *Control** are set to value 1 initially.

Now assuming the hypothesis is true for $i=t$. We show that it is true for $i=t+1$. At node u_t, let $l=W_1(a(u_t)\oplus a(u_d))$, if dimension l is fault-free, algorithm UNICAST.F1 routes the message from u_t to u_{t+1} through dimension l. That means u_{t+1} is the l-th dimensional neighbor of u_t. Algorithm BROADCAST.F3 will also route the message through dimension l, since *Control**$[l]$=1 by the hypothesis. Also by the hypothesis, *Control** has value 1 at all bit positions at which $a(u_t)$ and $a(u_d)$ differ. By Algorithm

BROADCAST.F3, $Control_l$ will also have value 1 at all those bit positions, except bit position l, which is reset to 0. Since $a(u_t)$ and $a(u_{t+1})$ differ at bit position l only, $Control_l$ is thus also properly set.

If, however, dimension l is faulty, which implies that $d(u_t,u_d)\geq 2$ (otherwise, u_d itself is faulty, it should not receive any message), then algorithm UNICAST.F1 routes the message from u_t to u_{t+1} through dimension $m=W_2(a(u_t)\oplus a(u_d))$, which always exists and is not faulty since each fault-free node has at most one faulty neighbor. In that case, u_{t+1} is the m-th dimensional neighbor of u_t. Algorithm BROADCAST.F3 will also route the message through dimension m, since $Control^*[m]=1$ by the hypothesis. $Control_m$, sending from u_t to u_{t+1} will have value 1 at bit position l and value 0 at bit position m. Other bit positions at which $a(u_{i+1})$ and $a(u_d)$ differ, $Control_m$ will have value 1. The positions in vector $Control_m$, at which $a(u_{i+1})$ and $a(u_d)$ agree, will be set to appropriate values depending on if u_d is a leaf node or a forward node in the broadcast tree.

By the induction step, it is proved that a message from u_s to u_d follows the same path as constructed by algorithm UNICAST.F1, which is shown to be a shortest path from u_s to u_d. ■

Lemma 7.2: Given nodes $u_s,u_d\in V(Q_n)-F(Q_n)$, exactly one shortest path from u_s to u_d will be constructed by executing algorithm BROADCAST.F3.

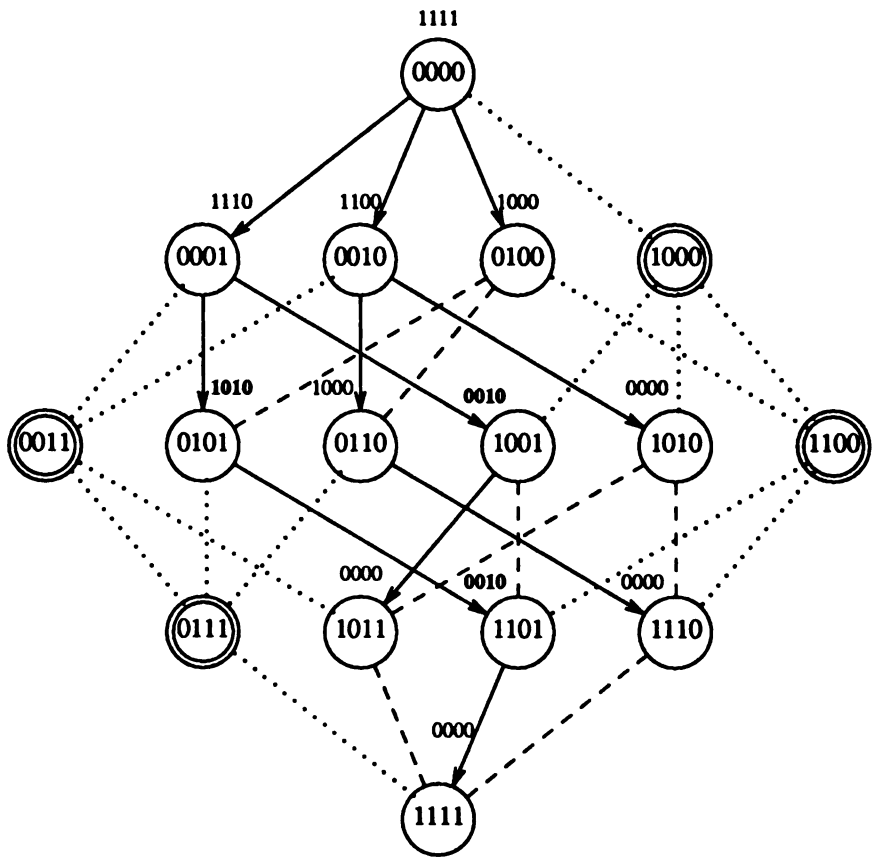
Proof: We prove it by contradiction. Suppose by executing Algorithm BROADCAST.F3, another path p' from u_s to u_d, besides path p, is formed, such that $p'\neq p$, where p is the path discussed in the proof of Lemma 7.1, and $p'=(u_s,u'_1,\cdots,u'_i,\cdots,u'_{d-1},u_d)$. Notice that, by the way Algorithm BROADCAST.F3 works, along any path from u_s to a destination, the message always traverses from a parent node to a child node. It never goes along the reverse direction. Thus, p' and p should have the same length d. Let i be the smallest integer such that $u_i\neq u'_i$. Let $l=W_1(a(u_{i-1})\oplus a(u_i))$ and $l'=W_1(a(u_{i-1})\oplus a(u'_i))$. Thus, $a(u_{i-1})$ differ form $a(u_d)$ at both bit positions l and l', and both dimensions are fault-free. Assuming $l>l'$, then, by

Algorithm BROADCAST.F3, $Control_{l'}$, received by u'_i, will be set properly as detailed in the proof of Lemmas 7.1, in particular, $Control_{l'}[l']=0$ and $Control_{l'}[l]=1$. However, by the execution of lines 7 to 9 in Algorithm BROADCAST.F3, $Control_l$, received by u_i, will have value zero at both bit positions l and l'. Since u_{i-1} and u_d differ at both bit positions l and l', and u_i differs from u_{i-1} at bit position l only, u_i and u_d must also differ at bit position l'. However, $Control_l[l']=0$, and, thus, the message will never go to u_d. Therefore, the existence of path p' is impossible. If we assume $l<l'$, the same contradiction will be generated. ■

Theorem 7.2: Algorithm BROADCAST.F3 is optimal. In other words, given a fault-free source node u_s and a set of faulty nodes $F(Q_n)$ in Q_n, algorithm BROADCAST.F3 broadcasts a message from u_s to every node $u \in V(Q_n)-F(Q_n)$ in smallest possible time steps, and the total traffic created by the message broadcasting is minimal, provided that every fault-free node has at most one faulty neighbor.

Proof: Notice that in broadcast communication, each fault-free node is a destination node. Lemma 7.1 and Lemma 7.2 not only ensure that each fault-free node receives exactly one copy of the message, but also imply the message traverses each link no more than once. The correctness of Theorem 7.2 follows. ■

Figure 7.6 depicts the broadcast tree generated by algorithm BROADCAST.F3 in a Q_4, assuming nodes 0011, 0111, 1000, and 1100 are faulty. Compared with the fault-free broadcast tree (Figure 4.7), the binary *Control* vectors at nodes 0101, 1001 and 1101 are different in the two figures. Because of the failure of node 0011, which is the dimension-1-neighbor of 0001, the *Control* vectors received by 0101 and 1001 are changed to 1010 and 0010 in stead of 1000 and 0000, respectively. Note that the change is at dimension 1, which reflects the failure of node 0011. Furthermore, since node 0111 is again faulty, that causes node 0101 to assign a *Control* vector of 0010 to node 1101, which then ensures the message is passed to 1111 at dimension 1.



Binary numbers outside circles: *Control* vectors;
Double circles: faulty nodes;
Dotted lines: faulty links;
Solid arrows: links used for the broadcast.
Dashed lines: links not involved in the broadcast;

Figure 7.6 A broadcast tree in a faulty Q_4 generated by BROADCAST.F2

Input: Local address: $a(u_0)$;
Destination list: $D=\{a(u_1),a(u_2),\cdots,a(u_k)\}$.

Output: Destination sublist(s): $D_1, D_2, \cdots, D_g$
where $D_i \subseteq D$, for $1 \le i \le g$; and $D_i \cap D_j = \emptyset$, for $i \ne j$.

Multicast Algorithm MULTICAST.F1:

(* f_b=1, if dimension b is faulty, otherwise f_b=0 $0 \le b \le n-1$ *)

1. (* Calculate relative addresses: *)
   $r_0(u_i)=b_{i(n-1)}b_{i(n-2)}..b_{ij}..b_{i0} \equiv a(u_i) \oplus a(u_0)$, for $1 \le i \le k$;
2. (* if local processor is a destination, send a copy to it *)
   If $r_0(u_i)=0$ for some $i \in [1,k]$, send the message to local processor;
3. (* calculate column sums *)

$$c_j = \begin{cases} \sum_{i=1}^{k} b_{ij} & \text{if } f_j=0 \\ 0 & \text{if } f_j=1 \end{cases}$$

   for $0 \le j \le n-1$;
4. p=0; (* start loop *)
5. Find smallest l, such that $c_l \ge c_j$ for all $0 \le j \le n-1$;
6. If c_l=0, stop.
7. $D_p=\emptyset$;
8. (* form a new destination sublist, reset related rows *)
   For each $r_0(u_i)$, $1 \le i \le k$, if b_{il}=1, then
8.1. $D_p=D_p+\{r_0(u_i) \oplus a(u_0)\}$;
8.2. Set $r_0(u_i)$=0;
8.3. $c_j=c_j-b_{ij}$ for $0 \le j \le n-1$;
9. Put destination sublist D_p into message header, send out the message at l-th dimension (to node $a(u_0) \oplus 2^l$);
10. (* start the selection of another dimension *)
    $p=p+1$; Goto step 5.

Figure 7.7 Greedy multicast algorithm for faulty hypercubes

7.3.3 Multicast in Faulty Hypercube

Now, let us discuss how the multicast algorithm MULTICAST.1 presented in Chapter 5 can be modified to handle faulty situations. As stated previously, it is required that every destination node receives the multicast message through a shortest path.

The idea is that at each forward node, when selecting a particular dimension to forward a message, the algorithm has to first check the *Fault_Vector* about the status of its neighboring nodes. If a dimension is faulty, it should not be selected. The algorithm has to select some candidate(s) from other dimensions for the message forwarding. A fault-tolerant revision of algorithm MULTICAST.1, the MULTICAST.F1 is listed in Figure 7.7.

Observing that, the difference between algorithm MULTICAST.F1 and algorithm MULTICAST.1 is mainly in step 3, whenever a dimension is faulty, its corresponding column-sum is forced to be zero. Therefore, it will never be selected.

Theorem 7.3: Given a source node u_s and a destination list: $D=\{u_1,u_2,\cdots,u_k\}$, $u_s,u_i \in V(Q_n)-F(Q_n)$, $1 \le i \le k$. Algorithm MULTICAST.F1 passes a message from u_s to every destination u_i through a shortest path between the two nodes, provided that each fault-free node in the system has at most one faulty neighbor.

Proof: It has been shown in Section 5.2 that the multicast tree generated by algorithm MULTICAST.1 guarantees a shortest path from the source to every destination. Here we show that the change made by MULTICAST.F1 will not cause any extra delay for message delivery. Suppose some forward node, say u_x, has a faulty neighbor, say u_f, at dimension j. Then u_f could only be in one of the following three situations.

1. Node u_f is not a destination node, but it would have been selected as a forward node to pass a message to some descendent nodes, if it had not been faulty. In this case, those destinations supposed to receive message from u_f must be at least 2 hops away from u_x (otherwise it would not be a child node of u_f). If they are exactly at 2 hops away, then there exists always an alternative shortest path to the

destinations, since u_x has at most one faulty neighbor. When the algorithm finds that dimension j is faulty, it will select the other dimension to pass the message. By the same argument, those destinations of 3 or more hops away from u_x will receive the message from u_x through a feasible shortest path.

2. Node u_f is a destination node, and it is also selected as a forward node to pass a message to some descendent nodes. Since any message is expected to be passed to fault-free nodes only, node u_j is not supposed to receive any message. For those fault-free destinations which were supposed to receive a message from u_f, the situation is the same as in case 1.
3. u_f is not selected to further pass the message, then it does not bother our multicast at all. ■

To illustrate how the algorithm works, let us consider the same example discussed in Section 5.3 but under some faulty situations. The multicast tree in fault-free case (Figure 5.2(c)) is redrawn here as Figure 7.8(a). In the figure, if node 00010 (the neighboring node of the source at dimension $W_1(00110 \oplus 10010)$) is faulty, then the source (00110) will simply select 10110 (at dimension $W_2(00110 \oplus 10010)$) to forward the message to destination 10010, instead of 00010. The resulting multicast tree has the same number of links.

However, if node 00100 (at dimension 1 of the source 00110) is faulty, the situation is different. In the fault-free case, as can be seen clearly in Figure 7.8(a), at the source node 00110, three neighboring nodes are: node 00100 is selected for forwarding messages to {00000,00001,10100,11101}; 00010 to 10010; and 00111 to itself. Now, since 00100 is faulty, the source can no longer pass any message to 00100. Let us see how the algorithm handles this situation. The initial reference array at the source node (Table 5.2) is relisted below as Table 7.1. Notice that the column-sum for dimension 1 (value 4) is no longer valid. After executing algorithm MULTICAST.F1, dimension 0 (to node 00111) will be selected first for forwarding message to (00111,00001,11101); and

Table 7.1 The reference array at node 00110 in faulty case

	Reference array 4 3 2 1 0	Distances to forward node
A[1, *]	0 0 0 0 1	1
A[2, *]	1 0 0 1 0	2
A[3, *]	1 1 0 1 1	4
A[4, *]	1 0 1 0 0	2
A[5, *]	0 0 1 1 1	3
A[6, *]	0 0 1 1 0	2
column_sum	3 1 3 ④3	

then 00010 to 00000 and 10010; and 10110 to 10100. The newly resulting multicast tree is shown in Figure 7.8(b).

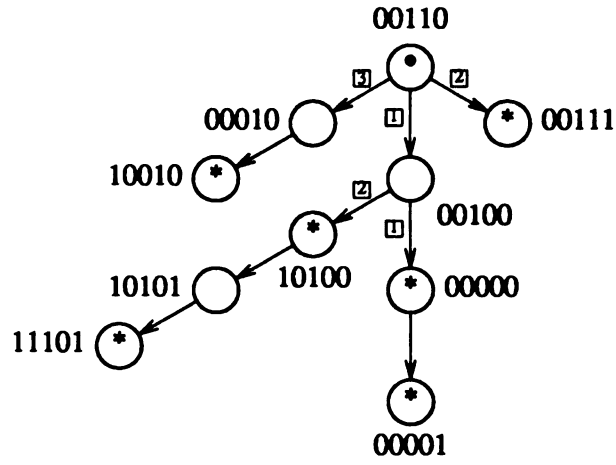
It can be easily checked out that the total traffic increases from 9 to 10 in this particular example. I would like to point out, however, the result is not always worse than faulty-free situation, since the greedy multicast itself is a heuristic algorithm.

7.4 A FAULT-TOLERANT ROUTER DESIGN

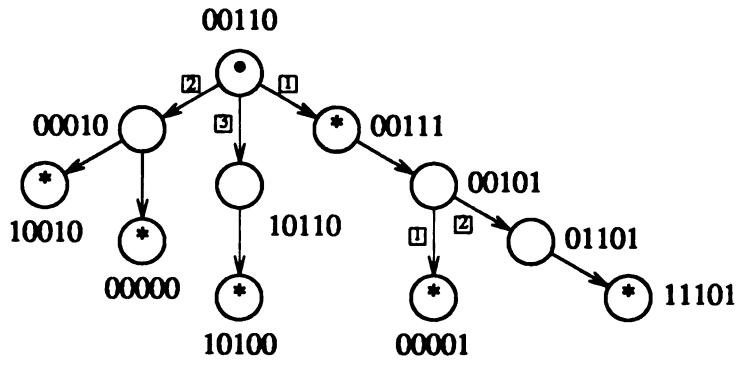
In the last section, we discussed algorithms for unicast, broadcast and multicast communication for faulty hypercubes under the assumption that each fault-free node in the system has no more than one faulty neighbor. As emphasized previously, the major goal of this research is to eventually provide a realistic VLSI hardware design, which not only provides efficient communication mechanism for fault-free hypercube, but also has certain fault-tolerant capability.

In this section, we discuss how the hardware router design presented in Chapter 6 can be made fault-tolerant with some minor modifications. We need to modify the Decoder in the Multicaster unit and redesign the Broadcaster unit.

In both algorithms BROADCAST.F3 and MULTICAST.F1, a *Fault_Vector* is needed at each node to keep the status of all n neighbors. In our hardware implementation, what we need for a *Fault_Vector* is just adding n flip-flops (f_i, for $0 \le i \le n-1$) as the indicator



(a). in a fault-free Q_5

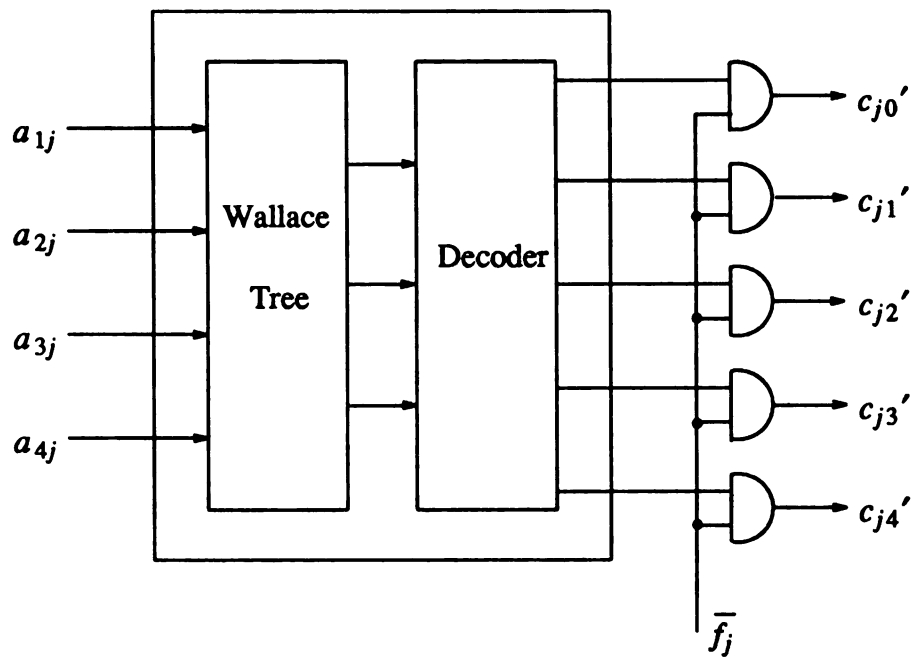


(b). in a faulty Q_5

"•": source node;
"*": destination node;
number in a small box: at which step the dimension is determined.

Figure 7.8 Comparison of multicast trees in a fault-free and a faulty Q_5

of the n dimensions, one for each dimension (neighboring node). Flip-flop j is set to one (f_j=1) if the neighboring node at dimension j is faulty; otherwise, it is reset to zero.



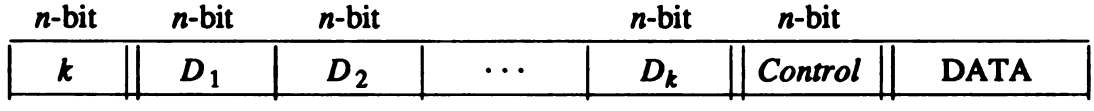
(* f_j=1, if dimension j is faulty *)

Figure 7.9 Decoder (DECR) for the j-th column for faulty Q_3

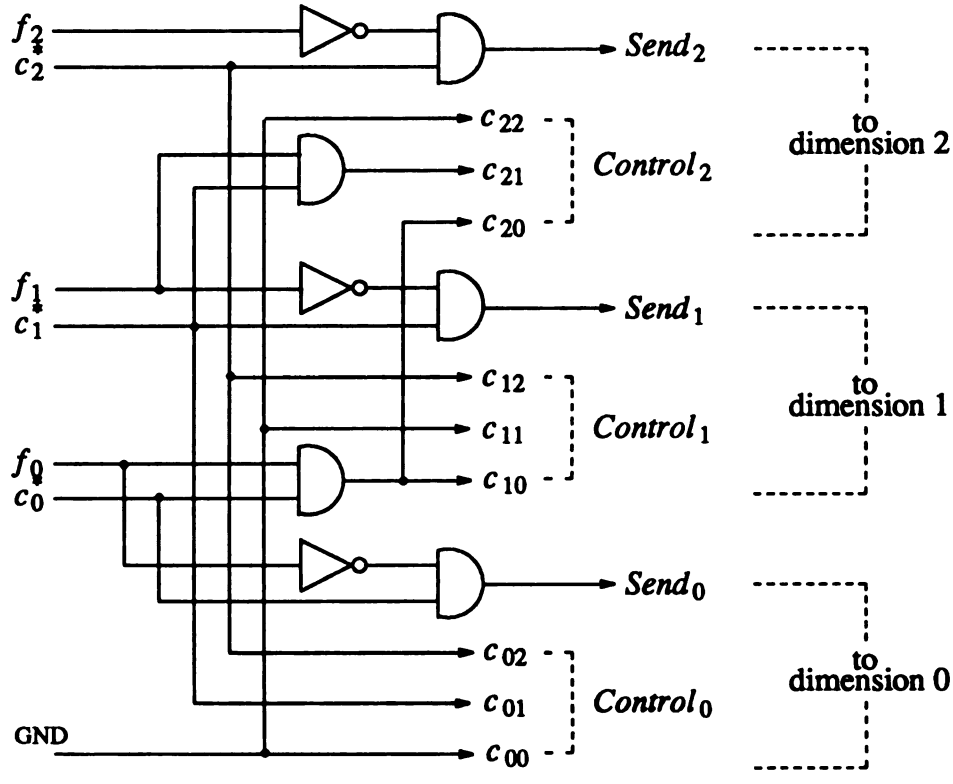
We first discuss the modification to the Multicaster. In the fault-free case, the column-sums for all n dimensions are directly sent to the Maximum Checker and then to Column Selector to select a dimension for message forwarding in each operation cycle. In a faulty hypercube, if dimension j is found faulty (f_j=1), it should never be selected. It can be realized by having the outputs of the Decoder ANDed with the complement of the fault indication flip-flop ($\overline{f_j}$) before being sent for further processing. Thus, if dimension j is faulty, the c_{ji} lines (for $0 \le i \le m$) will all be zeros[†]. This ensures that column j will never be selected. Figure 7.9 illustrates the fault-tolerant version of the Decoder.

As discussed previously, the Broadcaster unit design, an implementation of BROADCAST.2 as shown in Figure 6.5, is fairly simple for fault-free hypercubes. The dimension(s) which should receive messages can be easily determined based on the relative address of the local node with respect to the source node. For a faulty hypercube, we design a broadcaster to be an implementation of BROADCAST.F3. A binary vector *Control* has to be calculated for each selected dimension at a forward node. Algorithm BROADCAST.F3 does not look very straightforward. However, the actual hardware implementation, by using a few gates, looks simpler than the appearance of the algorithm. The diagram is shown in Figure 7.10.

Accordingly, a minor modification has to be made to the message format proposed in Section 6.2. As mentioned before, when $k=2^n-1$, there will be no destination fields, only a k field, a source address, and a data field will be contained in the message. To make the message format good for both fault-free and faulty cases, the source address will no longer be included in the message header. Instead, an n-bit *Control* field, immediately following the the k address fields, will be sent, replacing the position of source address. The general message format presented in Section 6.2 can be modified to the following form.



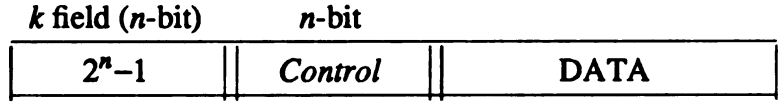
†Theoretically, c_{j0}' should be set to 1 to indicate the column-sum being zero. However, since we assume each fault-free node has at most one faulty neighbor, some column-sum must be non-zero, if the multicast has not finished. Therefore, this configuration will not affect the function of the Multicaster.



Input $Control^*=c_2^*c_1^*c_0^*$
If dimension i is faulty, then $f_i=1$ else $f_i=0$, $0 \le i \le n-1$
If $send_i=1$, "$Control_i$" will be sent to dimension i with the message.

Figure 7.10 Broadcaster for faulty hypercube

For multicast message, the *Control* will be set to all zeros. In the case of broadcast, the message format will look like:



In the case of $m<k<2^n-1$, there will be k destination fields between the k field and the *Control* field.

7.5 GENERAL FAULT-TOLERANT ROUTING PROBLEMS

In previous sections, we have discussed how the routing algorithms and their hardware implementation for fault-free hypercube can be modified to provide fault-tolerant routing for faulty hypercubes under the assumption that each fault-free node has no more than one faulty neighbor. However, it may happen that the assumption is violated. One possible policy is that if any node finds two or more of its neighbors are faulty, it sends a message to the host to report the situation. A proper action may be taken.

As a general problem, in this section, we consider the situation where a fault-free node may have more than one faulty neighboring node and the system is required to keep working. When the number of faulty nodes becomes large, the problem becomes complicated even for unicast communication. In the following discussion, we focus on unicast communication and software approaches only.

We discuss the problem for different neighborhood radius h. Neighborhood radius $h=j$ means each fault-free node has the status information of its j-neighborhood.

(1). $h=n$, $|F_1(u)| \le d-1$ for all $u \in V(Q_n)-F(Q_n)$.

In this case, every fault-free node has global status information. When a source wants to send a message to a destination at d hops away, every forward node (or more

generally, every fault-free node) is assumed to have no more than $d-1$ faulty neighbors. Condition $|F(Q_n)| \leq d-1$ will guarantee the requirement. The problem is relatively easy. Since there are d disjoint shortest paths of length d between the two nodes, at least one shortest path among them is still feasible. A natural way is to check a set of the d shortest paths, and find a feasible one, which is guaranteed to be shortest [Hawk85, LeHa88]. This is a special case of the situation (2) discussed below.

The advantage of this approach is that a shortest path can always be found. However, the algorithm is a centralized one in the sense that the source node has to obtain global status information and decides the entire path. Also, the problem of whether the algorithm is valid depends on the distance between the individual pair of nodes been considered.

(2). $h=n$, $|F_1(u)| \leq n-1$ for all $u \in V(Q_n)-F(Q_n)$.

Under this condition, each fault-free node has no more than $n-1$ faulty neighbors. Thus, there are n disjoint paths between any two nodes; d of them are shortest with length d; the other $n-d$ are of length $d+2$. Since $|F(Q_n)| \leq n-1$, we also have $|F_1(u)| \leq n-1$ for all $u \in V-F$. Thus, a path of length at most $d+2$ always exists.

There are two similar approaches to finding such a path. One approach [LeHa88] is to first check a set of d shortest paths and try to find a feasible one. If it does not succeed, then check the $n-d$ paths of length $d+2$. There must be one feasible.

The other approach [Hawk85] works in the following way. Given a set of n disjoint paths, d of which are of length d (shortest paths), and $n-d$ of which are of length $d+2$, the algorithm finds out the paths in which the faulty nodes reside, and marks those paths faulty. After all faulty paths have been identified, it selects a path of length d from the remaining feasible paths, if any. Otherwise, it selects a feasible path of length $d+2$. Since the number of faulty nodes is less than n, one of the above n paths must be feasible, which has length d or $d+2$, depending on the distribution of the faulty nodes.

Both approaches have to be implemented in a centralized way. However, neither method can guarantee optimal results in the sense that the algorithm may select a path of length $d+2$ while a shortest path of length d actually exists, since they check only a particular set of d disjoint paths among the $d!$ distinct ones.

Another approach is to check all $d!$ distinct paths of length d. If no one is feasible, then check a set of $n-d$ paths of length $d+2$, at least one of which is feasible. However, the computation time required by this approach makes it unattractive.

CHAPTER 8

SUMMARY AND DIRECTIONS FOR FUTURE RESEARCH

This chapter summarizes the major contribution of this dissertation research and outlines the directions for future research.

8.1 SUMMARY OF MAJOR CONTRIBUTIONS

This research has been motivated by a true need for providing a versatile, efficient and fault-tolerant interprocessor communication mechanism for DMMPs. Although DMMPs have been commercially available for only a few years, they have been demonstrated to be the most cost effective approach to construct massively parallel computing systems.

In order to fully explore the inherently massive computation power of DMMPs, several problems have to be solved, which include interconnection topology selection, communication hardware design, parallel operating system, fault-tolerant consideration, and parallel algorithm design. This research has aimed at two of the above problems: the communication hardware design and fault-tolerant consideration issues.

We have presented a graph-based model — the Optimal Multicast Tree to characterize all three types of interprocessor communication methods. Two parameters which are used as the measures of communication efficiency are time and traffic. Any communication problem in DMMPs can be regarded as the problem of finding a multicast tree which minimizes the two parameters.

A heuristic greedy algorithm has been proposed for multicast communication in hypercube multiprocessors. Multicast communication is highly demanded in many application areas, but not directly supported in any existing DMMP. This is the first time the multicast communication in a DMMP environment is systematically studied. The proposed algorithm guarantees that each destination receives the source message through a shortest path between the source and that destination. The traffic generated by a message delivery is very close to the optimal solution.

We have presented the architecture of a hardware router which is dedicated to performing the interprocessor communication tasks. Existing hypercube multiprocessors either do not provide broadcast and multicast at all or implement broadcast and multicast by subroutine calls (a software approach). The software implementation of communication algorithms based on the store-and-forward message forwarding scheme causes the communication mechanism to become the major source of bottleneck in the first generation DMMPs. The proposed hardware routing mechanism not only frees the nodal processor from routing computation, but also performs much better (3 orders of magnitude faster) than the nodal processor or communication coprocessor since it avoids the time consuming program execution and adopts relay fashion message forwarding. Using the hardware router, the data part of the message can be passed through all forward nodes with very little delay. Therefore, it will greatly speed-up interprocessor communication and improve the overall system performance.

Fault-tolerance is another important issue in massively parallel computer systems. Current VLSI technology makes large computer systems very reliable. It is unlikely that many components in a DMMP will fail at the same time. However, the situation when one or a few components fail may happen. Based on this consideration, fault-tolerant communication algorithms for the three types of interprocessor communications have been proposed for the case when each fault-free node has no more than one faulty neighboring node. The hardware implementation of the fault-tolerant interprocessor

communication algorithms is also presented. This is the first time a hardware communication device for DMMPs with fault-tolerant capability has been proposed. The proposed fault-tolerant communication mechanism has a great practical importance, especially for very large scale systems.

In summary, we have presented the modeling, algorithm development and hardware implementation of a novel interprocessor communication mechanism for DMMPs. The proposed mechanism has the following unique features. First, it adopts a relay approach for message forwarding to significantly reduce the communication latency. Second, it is distributed in the sense it does not require the source node to specify the global routing path(s). Third, it is versatile since it directly supports not only unicast and broadcast, but also multicast which is highly demanded but is not directly supported in any existing DMMP. Finally, it has certain fault-tolerant capabilities. The proposed communication mechanism can be readily applied to future generations of DMMPs.

8.2 DIRECTIONS FOR FUTURE RESEARCH

The following areas need to be further studied.

8.2.1 Multicast in other interconnection topologies

We have studied the interprocessor communication problem in DMMPs. The fundamental issues been studied and the OMT model are for any interconnection topologies. The communication algorithms presented in this study are basically for hypercube multiprocessors. Hypercube is the most popular interconnection topology currently used in DMMPs. However, some other interconnection topologies, for example, the 2-D mesh, also have their unique advantages. For a large scale system, processor/memory pairs interconnected through a 2D-mesh pattern are much easier to layout in VLSI implementation than those interconnected as a hypercube. In fact, the 2-D mesh topology is used in the Ametek Series 2010 DMMP. At first look, the message routing

problems in a 2-D mesh seems easier than those in a hypercube. However, finding an OMT is still not a trivial task. It will be interesting to see how multicast communication can be done in a 2-D mesh and other topologies, for example, the cube-connected-cycle and general k-ary n-cube.

8.2.2 Formal proof of NP-hardness for multicast in hypercube

As mentioned previously, we emphasized the algorithms which can be efficiently implemented in hardware. In fact, in the early stage of this research, we have investigated several other algorithms, and found that the greedy multicast algorithm proposed in this thesis has a very important feature — ease of hardware implementation. Even though the number of edges in a multicast tree generated by the greedy algorithm is very close to that of an OMT for any given multicast set, the algorithm does not guarantee an OMT if the number of destinations is greater than two. We conjectured that the OMT problem in hypercube topology is NP-hard based on the similarity between the OMT problem and the Steiner Tree problem which has been shown to be NP-complete in hypercube topology, and the fact that the number of general multicast tree patterns grows very rapidly as the dimension of the hypercube increases. In order to provide a solid theoretical foundation, we need to formally prove that conjecture. Another related question is "Is the OMT problem NP-hard in a 2-D mesh?" Our conjecture for this question is that it is not NP-hard.

8.2.3 Fault-tolerant considerations for more general cases

In Chapter 7, we have studied routing problems in faulty hypercubes. We proposed algorithms for all three types of communications and the related hardware implementation based on our model that each fault-free node has no more than one faulty neighboring node. As mentioned before, in a highly reliable system, we do not expect many components to fail at the same time. A system with one faulty node could be the case of a faulty system most of the time. Therefore, the fault-tolerant communication mechanism presented in this thesis has a great impact upon real operation of DMMPs.

However, as a general fault-tolerant routing problem, we have to consider the situation when our model does not fit. Similar to other fault-tolerant problems, this is a more difficult problem, but worth pursuing further.

8.2.4 Language support for multicast communication

Various parallel programming languages have been proposed recently to facilitate the use of DMMPs. These languages are either newly designed languages, such as CSP and OCCAM [Perr87], or enhancement of existing languages, such as Concurrent C [GeRo85] and Argonne's Macro C [Boyl87]. All these message-passing programming languages only support various forms of one-to-one communication. In these languages, multicast has to be performed as multiple one-to-one communications. With the direct hardware support of multicast and broadcast communication capability, new language primitives should be added to take advantage of these hardware features. Also needed in the languages is the declaration of groups of multiple destinations. Members of these groups should be dynamically configured and should be application dependent. As far as we know, no existing parallel programming languages support these features. Language support for multicast communication deserves further investigation.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Amet86] Ametek Computer Research Division, "Ametek system 14 user's guide: C edition," Version 2.0, May 1986.

[ArGr81] Armstrong, J. and F. Gray, "Fault diagnosis in a boolean *n* cube array of microprocessors," *IEEE Trans. Comput.*, Vol. C-30, No. 8, Aug. 1981, pp. 587-590.

[Aviz76] Avizienis, A., "Fault-tolerant systems," *IEEE Trans. Comput.*,, Vol. C-25, No. 12, 1976, pp. 1304-1312.

[BaPa86] Barhen, J. and J. F. Palmer, "The hypercube in robotics and machine intelligence," *Computers in Mechanical Engineering*, March 1986.

[Batc80] Batcher, K., "Design of a massively parallel processor," *IEEE Trans. Comput.*, Vol. C-29, Sept. 1980, pp. 836-840.

[BBKK68] Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer", *IEEE Trans. on Computers*, Vol. 17, No. 8, Aug. 1968, pp. 746-757.

[BBN87] BBN Advanced Computers Inc., *Butterfly products overview*, Oct. 1987.

[Bhat83] Bhat, K., "An efficient approach for fault diagnosis in a boolean n-cube array of multi processors," *IEEE Trans. Comput.*, Vol. C-32, No. 11, Nov. 1983, pp. 1070-1071.

[Boyl87] Boyle, J., et al., *Portable Programs for Parallel Processors,* Holt, Rienhart and Winston Inc., 1987.

[BrSc86] Brandenburg, J. E. and D. S. Scott, "Embeddings of communication trees and grids into hypercubes," *iPSC Technical Report*, No. 1, 1986, INTEL Scientific Computers.

[ChBN81] Chou, W., A. W. Bragg, and A. A. Nilsson, "The need for adaptive routing in the chaotic and unbalanced traffic environment", *IEEE Trans. Commun.* Vol. COM-29, No. 4, April 1981, pp. 481-490.

[ChEN87] Choi, Y., A. Esfahanian, and L. Ni, "One-to-k Communication in Distributed-memory multiprocessors," *Proc. of the 25-th Annual Allerton Conference on Communication, Control, and Computing*, pp. 268-270, September 1987.

[DaSe86] Dally, W. J. and C. L. Seitz, "The torus routing chip," *Distributed Computing*, Vol. 1, No. 3, 1986, pp. 187-196. pp. 531-540.

[DaSe87] Dally, W. J. and Seitz, C. L., "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Comput.*, Vol. C-36, No. 5, May, 1987, pp. 547-553.

[DCCM87] Driscoll, M., T. Chen, W. Chou, J. Miller, and P. Prins, "Router design project," Term project for CPS922, Winter 1987. Dept. CPS, Michigan State University, March 1987.

[DePa78] Despain, A. M., and Patterson, A., "X-tree: a tree structured multiprocessor computer architecture," *Proc. 5-th Intl. Symp. on Computer Architecture*, 1978, pp. 144-151.

[DoDu85] Dongarra, J. J., and I.S. Duff, "Advanced architecture computers," *Technical Memorandum No. 57*, Mathematics and Computer Science Division, Argonne National Laboratory, October 1985.

[Duni87] Duningan, T. H., "Hypercube Performace," *Hypercube Multiprocessors 1987*, Ed., M.T. Heath, SIAM, 1987, pp. 178-191.

[Feng81] Feng, T., "A survey of interconnection networks," *IEEE Computer*, Dec. 1981, pp. 12-27.

[FoKo86] Fox, G. C. and A. Kolawa, "Implementation of the high performance crystalline operating system on Intel iPSC hypercube," In M. T. Heath, editor, *Hypercube Multiprocessors 1986*, pp. 269-271, SIAM, 1986.

[Fold77] Foldes, S. "A characterization of hypercubes," *Discrete Math.*, Vol. 17, pp. 155-159, 1977.

[FoOt84] Fox, G. and S. Otto, "Algorithms for concurrent processors," *Physics Today*, 37, 5 (May, 1984), pp. 50-59.

[Fox83] Fox, G. C., "The impact of specialized processors in elementary particles physics," *Conf. on Scientific Calculation with Ensemble Computers*, Pauda, Italy, 1983.

[GaJo79] Garey, M. and Johnson, D., *Computers and intractability, a guide to the theory of NP-completeness*, Freeman, 1979.

[GeRo85] Gehani, N. H. and W. D. Roome, "Concurrent C," *Technical Report*, AT&T Bell Labs, 1985.

[GKLS83] Gajski, D. D., D.J. Kuck, D.H. Lawrie and A.H. Sameh, "Cedar - A large scale multiprocessor," *Proc. of the 1983 Int'l Conf. on Parallel Processing*, pp. 524-529, August 1983.

[GrFo82] Graham, R. L. and L. R. Foulds, "Unlikelyhood that minimal phylogenies for a realistic biological study can be constructed in reasonable computational time", *Mathematical Biosciences*, 60, pp. 133-142 (1982).

[GrRe86] Grunwald, D. C. and D. A. Reed, "Benchmarking hypercube hardware and software," *Technical Report UIUCDCS-R-86-1303*, Department of Computer Science, University of Illinois, November 1986.

[GuHS86] Gustafson, J., S. Hawkinson, and K. Scott, "The architecture of a homogeneous vector supercomputer," *Proc. 1986 Intl. Conf. on Parallel Processing*, Aug. 1986, pp. 649-652.

[GuMB88] Gustafson, J., G. Montry, and R. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM Journal on Scientific and Statistical Computing*, Vol. 9, No. 4, July 1988.

[Hamm50] Hamming, R. W., "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, 29, 1950, pp. 147-160.

[Hara72] Harary, F., *Graph Theory*, Addison-Wesley, Reading, MA, 1972.

[Hawk85] Hawkes, L., "A regular fault-tolerant architecture for interconnection networks," *IEEE Trans. Comput.*, Vol. C-34, No. 7, July 1985, pp. 677-680.

[Hill85] Hillis, W., *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.

[HLSM82] Hayes, L., R. Lau, D. Siewiorek, and D. Mizell, "A survey of highly parallel computing," *IEEE Computer*, Jan. 1982, pp. 9-24.

[HMSC86] Hayes, J., T. Mudge, Q. Stout, S. Colley, and J. Palmer, "Architecture of a hypercube supercomputer," *Proc. 1986 Intl. Conf. on Parallel Processing*, Aug. 1986, pp. 653-660.

[HoJo86] Ho, C. and S. Johnson, "Distributed routing algorithms for broadcasting and personalized communication in hypercubes," *Proc. 1986 Intl. Conf. on Parallel Processing*, Aug. 1986, pp. 640-648.

[HwBr84] Hwang, K., and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., 1984.

[HwGh87] Hwang, K, and J. Ghosh, "Hypernet: a communication-efficient architecture for costructing massively parallel computers," *IEEE Trans. Comp.*, Vol. C-36, No. 12, Dec. 1987, pp. 1450-1466.

[iPSC88] INTEL Scientific Computers, "iPSC/2," Product description, order number: 280110-001. Intel Corporation, 1988.

[Kats88] Katseff, H. P., "Incomplete Hypercube," *IEEE Trans. Comp.*, Vol. 37, No. 5, May 1988, pp. 604-608.

[KeKl79] Kermani, P., and Kleinrock, L., "Virtual cut-through: a new computer communication switching technique, " *Computer Networks*, Vol. 3, pp. 267-286, 1979.

[Kim79] Kim, K. H., "Error detection, reconfiguration and testing in distributed processing systems," *Proc. 1st Int'l Conf. Distributed Computer Systems*, Oct. 1979, pp. 284-295.

[KrVC86] Krumme, D., K. Venkataraman, and G. Cybenko, "Hypercube embedding is NP-complete," in *Hypercube Multiprocessor 1986* (M. Heath ed.), Philadelphia, 1986, pp. 148-157.

[Kuhl80] Kuhl, J. G., "Fault-diagnosis in computing networks," ECS Thch. Rep. 80-1, Dep. Elec. Comput. Eng. Univ. of Iowa, Iowa City, IA, Aug. 1980.

[KuRe86] Kuhl, J. and S. Reddy, "Fault-tolerance considerations in large, multiple-processor systems," *IEEE Computer,* March 1986, pp. 56-67.

[LaEN88a] Lan, Y., A. H. Esfahanian, and L. M. Ni, "Distributed Multi-destination Routing in Hypercube Multiprocessors", *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, January 19-20, 1988.

[LaEN88b] Lan, Y., A. Esfahanian, and L. M. Ni, "Multicast in Hypercube Multiprocessors," *Proc. of the 1988 Phoenix Conference on Computers and Communications*, March 1988, pp. 27-30.

[LaNE88] Lan, Y., L. M. Ni, and A. Esfahanian, "Relay Approach Message Routing in Hypercube Multiprocessors," *Proc. of The Third International Conference on Supercomputing*, pp. 174-182, May 1988.

[LeHa88] Lee, T. C. and J. P. Hayes, "Routing and broadcasting in hypercube computers," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, January 19-20, 1988.

[MuBA87] Mudge, T. N., G. D. Buzzard and T. S. Abdel-Rahman, "A high performance operating system for the NCUBE," In *Hypercube Multiprocessors 1986*, pp. 90-99.

[MoSc86] Moler, C., and D. Scott, "Communication utilities for the iPSC," *iPSC Technical Report*, No. 2, 1986. INTEL Scientific Computers.

[Myer86] Myers, W., "Getting the cycles out of a supercomputer," *IEEE Computer*, March, 1986, pp. 89-100.

[NiKP87] Ni, L.M., King, C.T. and Prins, P., "Parallel algorithm design considerations for hypercube multiprocessors," *Proc. of 1987 Int'l Conf. on Parallel Processing*, August 1987, pp. 717-720.

[Olso85] Olson, R., "Parallel processing in a message-based operating system," *IEEE Software*, July 1985, pp. 39-49.

[Patt85] Patton, P. C., "Multiprocessors: architecture and applications," *IEEE Computer*, Vol. 18, No. 6, June, 1985, pp. 29-40.

[Perr87] Perrott, R. H., *Parallel Programming*, Addison-Wesley Publishing Company, 1987.

[PrVu81] Preparata F. and Vuillemin J., "The cube-connected cycles: a versatile network for parallel computation," *Commu. of ACM*, Vol. 24, No. 5, May 1981, pp. 300-309.

[PTLP85] Peterson, J., J. Tuazon, D. Liberman, and M. Pniel, "The MARK III hypercube-ensemble concurrent computer," *Proc. Intl. Conf. on Parallel Processing*, Aug. 1985, pp. 71-73.

[ReFu87] Reed, D. A. and R. M. Fujimoto, *Multicomputer networks: message-based parallel processing*, The MIT Press, Cambridge, Massachusetts, 1987.

[Renn86] Rennels, D., "On implementing fault-tolerance in binary hypercubes," *Proc. 16-th Intl. Symp. on Fault-tolerant Computing*, 1986, pp. 344-349.

[SaSc85a] Saad, Y. and M. Schultz, "Topological properties of hypercubes," *Res. Rep.*, YALEU/DCS/RR-389, Dep. Comput. Sci., Yale Univ. 1985.

[SaSc85b] Saad, Y. and M. Schultz, "Data communication in hypercubes," *Res. Rep.*, YALEU/DCS/RR-428, Dep. Comput. Sci., Yale Univ. 1985.

[SASL85] Schneck, P. B., D. Austin, S. L. Squires, J. Lehmann, D. Mizell, and K. Wallgren, "Parallel processor programs in the federal government," *IEEE Computer*, Vol. 18, No. 6, May, 1985, pp. 43-56.

[Seit85] Seitz, C., "The COSMIC Cube," *Communications of the ACM*, Vol. 28, No. 1, Jan. 1985, pp. 22-33.

[Seit87] Seitz, C., Private communication with L. Ni., Sept. 1987.

[ShFi87] Shih, Y., and Fier J., "Hyprecube systems and key applications," in *Parallel Processing for Supercomputing and AI"*, Chapter 6, ed. Hwang and DeGroot, McGraw-Hill Book Company, New York, 1987.

[SuBa77] Sullivan, H. and T. Bashkow, "A large scale, homogeneous, fully distributed parallel machine, I," *Proc. 4th Symp. on Computer Architecture* , pp. 105-117, 1977.

[Tane81] Tanenbaum, A. S., *Computer Networks.* Englewood Cliffs, NJ, Prentice-Hall, 1981.

[TPPL85] Tuazon, J., J. Peterson, M. Pniel, and D. Liberrman, "Caltech Mark II hypercube concurrent processor," *1985 International Conf. on Parallel Processing*, 1985, pp. 666-673.

[Wile87] Wiley, P., "A parallel architecture comes of age at last," *IEEE Spectrum*, Vol. 24, No. 6, June, 1987, pp. 46-50.