1991

Parallel Computation on Hypercube-Like Machines.

Kyung Hee Kwon
Louisiana State University and Agricultural & Mechanical College

Order Number 9219552

Parallel computation on hypercube-like machines

Kwon, Kyung Hee, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1991

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

PARALLEL COMPUTATION ON HYPERCUBE-LIKE MACHINES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Kyung Hee Kwon
B.S., Korea University, 1976
M.S., Old Dominion University, 1986
December 1991

Acknowledgements

I am deeply indebted to my major advisor, Dr. Si-Qing Zheng, for his enormous help in shaping the contents and forms of my research. I would like to thank Dr. Donald H. Kraft and Dr. Said Bettayeb for their remarkable efforts in editing this dissertation. I would also like to express my sincere appreciation to the other members of my reading committee, Dr. Alexander Skavantzos, Dr. Bush Jones, Dr. Doris Carver, and Dr. William Hoffman for their advice and encouragement.

While working around the Department of Computer Science, I have met a number of wonderful faculty members and graduate students. Even though I can not name each of them individually, I wish to express my gratitude to them for their encouragement in helping me to finish this study.

Finally, I wish to thank my mother, Sang Sun Lee, my wife, Sung Sook Kim, and my two sons, Taek Woo Kwon and Taek Soo Kwon, for their support and tolerance throughout the period of this work.

Table of Contents

List of Tables

List of Figures

Abstract

The hypercube interconnection network has been recognized to be very suitable for a parallel computing architecture due to its attractive topological properties. Recently, several modified hypercubes have been proposed to improve the performance of a hypercube. This dissertation deals with two modified hypercubes, the X-hypercube and the Z-cube. The X-hypercube is a variant of the hypercube, with the same amount of hardware but a diameter of only $\lceil (n+1)/2 \rceil$ in a hypercube of dimension n. The Z-cube has only 75 percent of the edges of a hypercube with the same number vertices and the same diameter as the hypercube.

In this dissertation, we investigate some topological properties and the effectiveness of the X-hypercube and the Z-cube in their combinatorial and computational aspects. We give the optimal or nearly optimal data communication algorithms including routing, broadcasting, and census function for the X-hypercube and the Z-cube. We also give the optimal embedding algorithms between the X-hypercube and the hypercube. It is shown that the average distance between vertices in a X-hypercube is roughly 13/16 of that in a hypercube. This implies that a X-hypercube achieves the better average communication performance than a hypercube. In addition, a set of fundamental SIMD algorithms for a X-hypercube is given.

Our results indicate that the X-hypercube makes an improvement in performance over the hypercube, but not as much as the reduction in a diameter, and the Z-cube is a good alternative for the hypercube as far as the VLSI implementation is of major concern.

Chapter 1

Introduction

1.1. Overview

There are two broad classes of parallel computer architectures with a large number of processors. The first class of architectures is multicomputer systems with distributed memories. In this type of machine, each processor has its own local memory and processors are connected by a static or fixed interconnection network. The communication among the processors is achieved by message passing through the network, and computation is data driven. By message passing, it is meant that data or possibly codes are transferred from a processor to another processor by traveling across a sequence of nearest neighbor nodes. Synchronization is driven by data in the sense that computation in some node is performed only when its necessary data are available. One major advantage of such architectures is the simplicity of their design.

The second class of parallel computer is multiprocessor systems with shared memories. In a shared memory organization, a set of processors and a common bank of memory units are connected through a fast bus or interconnection network. Variations on this scheme are numerous, but the essential feature here is how to access a shared memory. The main advantage of such second configuration is that it enables us to make the data access transparent to the user, who may regard data as being held in a large memory which is readily accessible to any processor. This greatly facilitates the programming of the machine, but memory conflicts can lead to degraded performance. On the other hand, shared memory models can not easily take advantage of

proximity of data in problems where communication is local. Moreover, the switching network becomes exceedingly complex to build as the number of nodes increases. This raises the problem of reliability, as the probability of failure increases proportionally with the number of components. In contrast, the first models can easily be made fault tolerant by shutting down failing nodes [SS86,LD90,Akl89,Qui87].

Therefore, the multicomputer based on interconnection networks has been recognized as a major option in the design of parallel computer. Many interconnection networks, including mesh, shuffle exchange, cross-bar, permutation and hypercube have been suggested as the basis for parallel computing architectures. Some of these design requirements are low node connectivity, small diameter, simple routing algorithm, regularity, symmetry, and fault tolerance [RF87]. Among these choices, the hypercube interconnection network has received much attention due to its topological suitability for parallel processing. Hereinafter, a hypercube interconnection network will be referred to as hypercube. A n-dimensional hypercube has 2^n nodes, with each node having n links. If the nodes of the hypercube are numbered from 0 to 2^n-1, then the connection scheme can be defined by the set of edges that can be drawn between any two nodes whose numbering differ by one bit position in their binary representation.

For the past several years, three kinds of modified hypercubes have been introduced. The *twisted cube* has been suggested by Hilbers, Koopman, and Snepscheut in 1987 [HKS87], the *X–hypercube* by Sung [Sun88] in 1988, and the *multiply twisted cube* by Efe in 1988 [EBSS88]. None of these modified hypercubes are isomorphic to each other. They have the same structural complexity as the hypercube, i.e., k-dimensional modified hypercubes have the same number of nodes, and

the same number of links per node as k-dimensional hypercube, but the diameter is reduced by almost half. Thus the communication delay in these modified hypercubes are expected much lower than that in hypercube. This means that these modified hypercubes have the advantage over the hypercube when the data communication is of major concerns.

These modified hypercube architectures do not necessarily guarantee that faster and more efficient algorithms will follow. The algorithmic aspects of these new schemes should be investigated to take advantage of their merits. Our interest in the algorithms concerns data communication since the major advantage of modified hypercubes is a reduced diameter. The typical communication algorithms are routing, broadcasting, and census. Routing provides the shortest path from the source node to the destination, allowing any two processors in the network to exchange their data. Broadcasting sends a message from a processor to all other processors in the multi-computer system. Many parallel algorithms use broadcasting as a sub-algorithm in their computation. The census function gathers information about the state of the network and sends it to a central location. We are also interested in communication-intensive problems that occur as sub-problems in larger computation tasks. For example, prefix computation, packing operation, and sorting algorithms are considered as such problems.

1.2. Previous Studies.

The key for obtaining modified hypercubes is to choose the set of edges from hypercube and to twist them systematically. By twisting the edges, the symmetry of

hypercube may be broken a little. However, the diameter can be reduced by almost half. There are three distinctive modified hypercubes as stated previously. Whatever they are called, there is a basic idea underlying all of them. In this section, we review how these modified hypercubes are defined, and briefly introduce earlier investigations.

1.2.1. Twisted Cube

Hilbers, et al. [HKS87] suggest a twisted cube by twisting some edges of the hypercube as follows. Here, n is an odd number and P_x stands for the sum modulo 2 of the bits in x.

Definition 1.1:

(1) $\{0,1\}$ is the edge of the twisted 1-cube.

(2) For any sequence x of n bits, and for any bits a, b, c, and d, such that $a \neq c$ and $b \neq d$, the twisted (n+2)-cube contains edges $\{abx, cdx\}$ if $P_x = 0$, and $\{abx, adx\}$ if $P_x = 1$, and $\{abx, cbx\}$, and for any sequence x and y of n bits such that $\{x, y\}$ is an edge of the twisted n-cube, and for any bits a and b, the twisted $(n+2)$-cube also contains the edge $\{abx, aby\}$.

Figure 1 shows twisted n-cube for $n = 3$ and $n = 5$. The shortest path algorithm from a source node to a destination node has been studied. Abraham and Padmanabhan [AP89] have given a distributed routing algorithm and investigated important topological properties such as distance distribution and average distance. They compare the performance of a twisted cube with that of a hypercube and conclude

that the twisted cube delivers an improvement in performance over the hypercube.

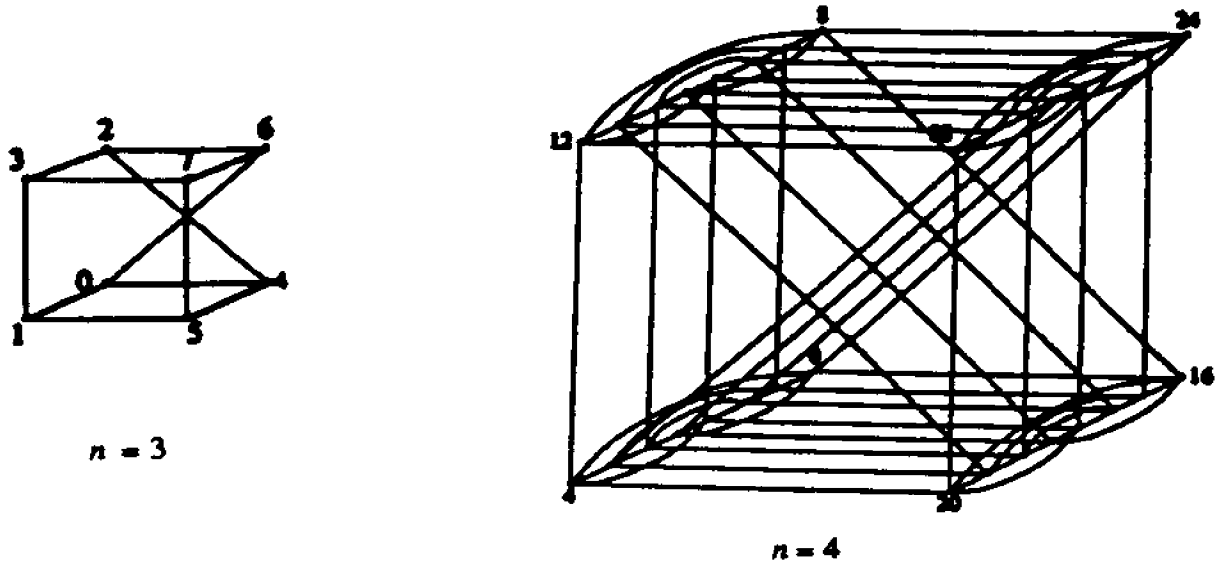


Figure 1: Twisted n-cube

1.2.2. X-Hypercube

The X-hypercube has been proposed by Y. Sung [Sun88]. The X-hypercube of dimension n consists of 2^n nodes, and each node has exactly n links. For $n < 2$, it is the same as a hypercube. For $n = 2$, it is either a hypercube of square shape or a hypercube of twisted shape. For $n > 2$, it is recursively defined as follows:

Definition 1.2

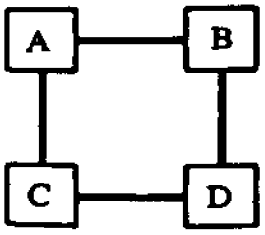
An X-hypercube of dimension n is constructed by two boxes, each box is an X-hypercube of dimension $n-1$.

(1) If n is odd, Box-1 has a square shape as depicted in figure 2(a), where A, B, C, and D are boxes of an $(n-3)$-dimensional X-hypercube; for the sake of clarity, the connection between Box A and Box C in the figure 2(a) is shown

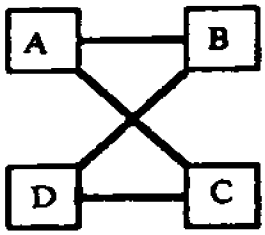
by a shaded trunk, and so on. A node in Box A is connected to its corresponding node in Box B, etc. Box-2 has a similar structure as Box-1 but has an X shape as shown in figure 2(b), where A, B, C, and D are boxes exactly as in Box-1. Notice that the adjacency relation in Box-1 has been exactly preserved in Box-2; by sense we may say that Box-1 is isomorphic to Box-2. An interconnection between Box-1 and Box-2 is illustrated in figure 2(c). Figure 2(c) gives the complete structure of an X-hypercube of dimension n, where n is odd.

(2) If n is even, a n-dimensional X-hypercube is also made from two boxes, each box is identical to the one described in part (1). Figure 3 illustrates the two boxes and the interconnection between them.

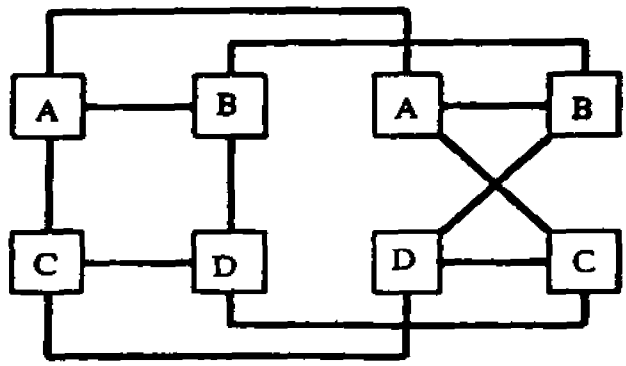
But this recursive construction does not give a clear insight of node connectivity. Sung discusses the shortest path between two nodes and the embedding of Hamiltonian circuit and binary tree [Sun88]. But the optimal shortest path and the embedding of a binary tree still remain as open problems.



(a) Box-1



(b) Box-2



(c) Interconnection between Box-1 and Box-2

Figure 2: X-Hypercube of odd dimension

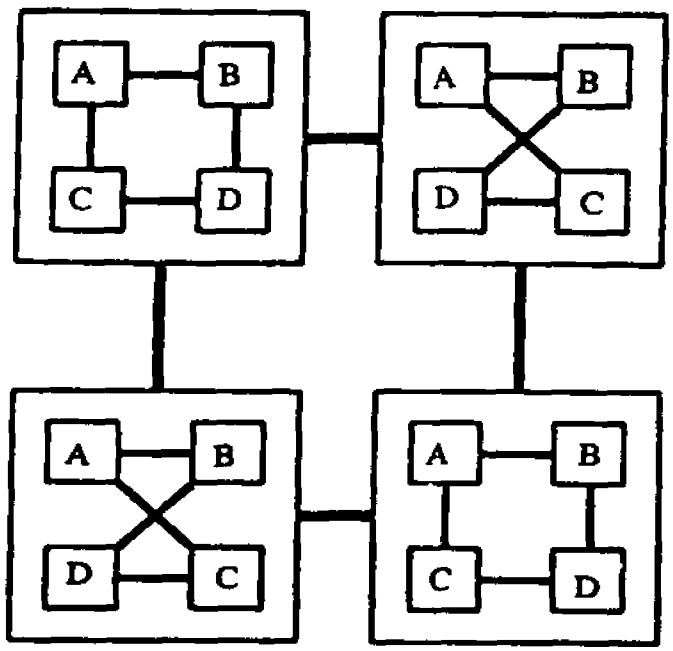


Figure 3: X-Hypercube of even dimension

1.2.3. Multiply-Twisted Cube

Efe, et al. [EBSS88] propose the multiply-twisted cube as follows. Two binary strings $x=x_1x_0$ and $y=y_1y_0$ are pair-related, denoted $x \sim y$, if and only if $(x,y) \in \{(00,00), (10,10), (01,11), (11,01)\}$.

Definition 1.3:

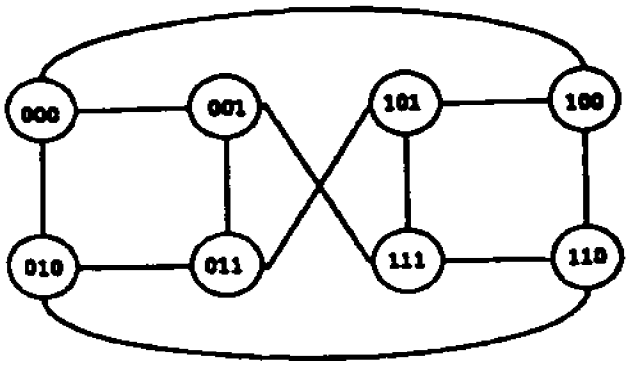
The n dimensional multiply-twisted cube, denoted MQ_n, is the labeled graph defined inductively as follows. MQ_1 is the complete graph on two vertices with label 0 and 1. For $n > 1$, MQ_n contains MQ_{n-1}^0 and MQ_{n-1}^1 joined according to the following rule: the vertex $u = 0u_{n-2}...u_0$ from MQ_{n-1}^0 and the vertex $v = 1v_{n-2}...v_0$ from MQ_{n-1}^1 are adjacent in MQ_n if and only if

(1) $u_{n-2} = v_{n-2}$ if n is even, and

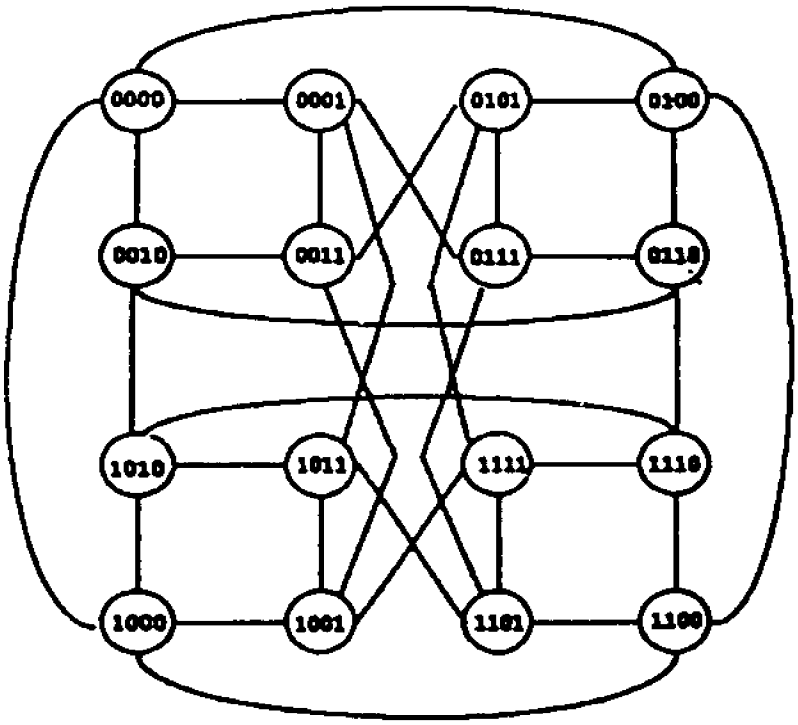
(2) for $0 \le i < \lceil (n-1)/2 \rceil$, $u_{2i+1}u_{2i} \sim v_{2i+1}v_{2i}$.

Figure 4 shows MQ_n for $n = 3$ and $n = 4$. In [EBSS88], basic topological properties and the shortest path algorithm of multiply-twisted cube are investigated. The data communication algorithms are given by Zheng [Zhe90]. The average distance between vertices has been calculated by Young and Zheng [YZ91].

The most fundamental issues for the twisted cube and the multiply twisted cube have already been investigated. The multiply twisted hypercube has been studied in great detail. Compared to the other two modified hypercubes, the X-hypercube has been left unexplored in many aspects. Therefore, the main focus of this dissertation is to investigate the topological properties of the X-hypercube more extensively and to explore its efficiency in parallel computation.



$n = 3$



$n = 4$

Figure 4: MQ_n for $n = 3$ and $n = 4$

1.3. Organization of the Dissertation

Chapter 2 gives a new definition of the X-hypercube in terms of its connectivity. In order for the interconnection network to be utilized in full for parallel computation, the connectivity between vertices should be very clear. For example, two adjacent vertices in a hypercube are defined explicitly as follows: two vertices are connected if and only if the binary labels of two vertices differ in exactly one bit. In fact, the original definition of the X-hypercube is not so formal. We redefine the X-hypercube. Based on this new definition, the optimal algorithms for the embedding of a hypercube on a X-hypercube, and vice versa, are developed. This implies that the hypercube can be efficiently simulated by the X-hypercube, and vice versa. To substantiate this, the algorithm to compute sum of n elements on n dimensional X-hypercube is presented.

In order for a parallel machine to run efficiently, it must allow any two processors to communicate with each other through the shortest path. Chapter 3 presents two algorithms, *PATH_1* and *PATH_2*, to compute a shortest path between any two vertices of an n- dimensional X-hypercube in $O(n)$ and $O(n^2)$ time, respectively. *PATH_1* is desirable in circuit switching networks with centralized control, while *PATH_2* is good for packet switching network with distributed control.

The average distance between vertices has an important meaning in data communication performance. That is, it characterizes the average case data communication performance. Chapter 4 shows that the average distance between vertices in X-hypercube is almost 13/16 of that in a hypercube. A recurrence relation on total distance between vertices in X-hypercube is also given.

Chapter 5 is concerned with the problem of data communication, such as the data broadcasting and the census function. As major results in this chapter, four major algorithms, *BROADCAST*, *MIN1*,*MIN2*, and *ROUTE1* are presented. *BROADCAST* broadcasts a message from any processor to all other processors of a X-hypercube in $O(n)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps. *MIN1* finds the minimum among 2^n numbers stored in a n-dimensional X-hypercube and send it to any processor in $O(n)$ computation steps and n communication steps. *MIN2* performs the same census function in $O(n^2)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps. *ROUTE1* sends a message from any processor to any other processor in $O(n)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps.

There exist many important communication intensive kernel algorithms which occur as sub-problems in larger computation tasks. It is natural to examine these basic parallel communication-intensive algorithms after specifying parallel communication algorithms. Chapter 6 presents a many-to-many routing algorithm, *ROUTE2* which runs in $O(n)$ time for a n dimensional X-hypercube. The algorithm *PREFIX* computes prefix sums in $O(n)$ time. Based on *ROUTE2* and *PREFIX*, the algorithms to pack and uppack the data, *PACK* and *UPPACK*, are presented having $O(n)$ time complexity. Next, this chapter consider the problem of sorting 2^n keys on a n dimensional X-hypercube. Two sorting algorithm, *RADIX* and *BITONIC-SORT-XH*, are given. *RADIX* implements the radix exchange sort in $O(mn)$ time complexity, where m is the number of bits in binary number to be sorted. *BITONIC-SORT-XH* shows that Batcher's sorting method can be applied on a X-hypercube in $O(n^2)$ time. All these algorithms are based on the connectivity of X-hypercubes.

Chapter 7 introduces a new interconnection network called the *Z−cube* which has been proposed by Zheng and Latifi [ZL91]. It has a similar structure to the hypercube and its vertex degree is 3/4 of that of a hypercube. The Z-cube is more suitable for VLSI implementation than the hypercube because it has lower vertex degree and smaller number of edges. Algorithms for data communication on Z-cube are also explored.

Finally, Chapter 8 offers some concluding remarks and open problems for future research.

Chapter 2

New Definitions of a X-hypercube and Its Applications

2.1. Introduction

A hypercube machine is a multicomputer system whose processors and data links connecting processors correspond to the vertices and edges of a hypercube graph. The hypercube network has been considered as one of most popular interconnection networks for parallel computations, due to its desirable topological properties, such as small diameter, low vertex degree, and structural regularity. When a hypercube machine is abstracted as a graph, the processors are treated as vertices and data links are treated as edges. Each vertex is given a unique binary label, and the connectivity between vertices can be easily determined by inspecting the labels associated with vertices.

In contrast, the X-hypercube is less regular. In fact, the original definition of the X-hypercube, which appeared in [Sun88], is not so formal. In [Zhe89], a formal definition of X-hypercube is introduced. However, this definition does not provide explicit conditions for the connectivity of vertices. The analysis of the algorithmic aspects and topological properties of X-hypercube in [Zhe89] are based on the notion of *array arrangement of vertices*, which is used to derive the connections between vertices. In addition to the inconvenience, finding the connections by using the array arrangement involves computing overhead. Compared to the hypercube, one of the major disadvantages of the X-hypercube is the fact that it is hard to use, due to its more complicated connectivity. In this chapter, we give an alternative definition for

the X-hypercube. We show that using this definition, the connectivity between any two vertices in a the X-hypercube can be easily determined by scanning the labels of the vertices. We also show how to use this definition to implement simulations between hypercube and X-hypercube.

2.2. Definitions of a X-Hypercube

A n-dimensional X-hypercube, which is denoted as Q_n^X, is a graph of 2^n vertices. To simplify our presentation, we define the n-dimensional *companion X-hypercube*, denoted as Q_n^C, in parallel with Q_n^X. Each node in Q_n^X, or Q_n^C, is labeled by a distinct n-bit binary number in B_n, by which we denote the set of all possible n-bit binary numbers $b_n b_{n-1} \ldots b_1$. We use # to denote the concatenation operation on two binary numbers, i.e., for two binary numbers b_1 and b_2, $b_1 \# b_2$ is the binary number consisting of $|b_1|+|b_2|$ bits obtained by concatenating b_1 and b_2, where $|b|$ is the number of bits in b. We use $b\#B_n$ to denote the set of binary numbers obtained by concatenating the binary number b with all numbers in B_n, i.e., $b\#B_n = \{b\#b' \mid b' \in B_n\}$. The formal recursive definition of Q_n^X (and Q_n^C) given in [Zhe89] is as follows:

Definition 2.1:

$Q_1^X = (V_1^X, E_1^X)$, where

$V_1^X = \{0, 1\}$, and

$E_1^X = \{[0,1]\}$.

$Q_1^C = (V_1^C, E_1^C)$ is identical to Q_1^X.

For $n > 1$ and n is odd,

$Q_n^X = (V_n^X, E_n^X)$, where

$V_n^X = B_n$, and

$E_n^X = \{[0\#v_i, 0\#v_j] \mid v_i, v_j \in B_{n-1} \text{ and } [v_i, v_j] \in E_{n-1}^X\} \cup$
$\{[1\#v_i, 1\#v_j] \mid v_i, v_j \in B_{n-1} \text{ and } [v_i, v_j] \in E_{n-1}^C\} \cup$
$\{(0\#v_i, 1\#v_j) \mid v_i, v_j \in B_{n-1} \text{ and } v_i = v_j\}$.

$Q_n^C = (V_n^C, E_n^C)$, where

$V_n^C = B_n$, and

$E_n^C = \{[0\#v_i, 0\#v_j] \mid v_i, v_j \in B_{n-1} \text{ and } [v_i, v_j] \in E_{n-1}^C\} \cup$
$\{[1\#v_i, 1\#v_j] \mid v_i, v_j \in B_{n-1} \text{ and } [v_i, v_j] \in E_{n-1}^X\} \cup$
$\{[0\#v_i, 1\#v_j] \mid v_i, v_j \in B_{n-1} \text{ and } v_i = v_j\}$.

For $n > 1$ and n is even,

$Q_n^X = (V_n^X, E_n^X)$, where

$V_n^X = B_n$, and

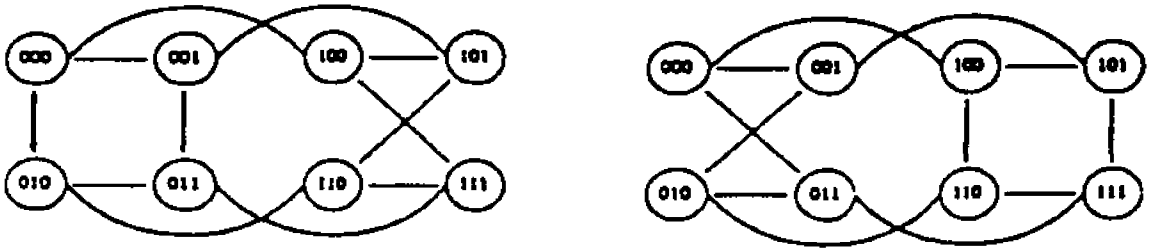
$E_n^X = \{[0\#v_i, 0\#v_j] \mid v_i, v_j \in B_{n-1} \text{ and } [v_i, v_j] \in E_{n-1}^X\} \cup$
$\{[1\#v_i, 1\#v_j] \mid v_i, v_j \in B_{n-1} \text{ and } [v_i, v_j] \in E_{n-1}^C\} \cup$
$\{[00\#v_i, 10\#v_j], [01\#v_i, 11\#v_j] \mid v_i, v_j \in B_{n-2} \text{ and } v_i = v_j\}$.

$Q_n^C = (V_n^C, E_n^C)$

$V_n^C = B_n$, and

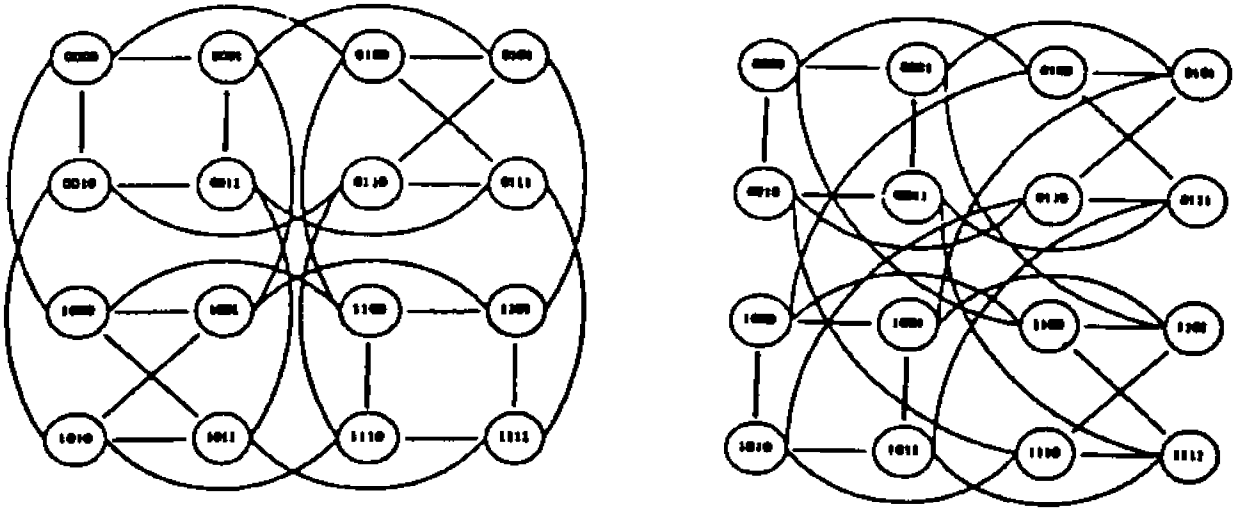
$E_n^C = \{[0\#v_i, 0\#v_j], [1\#v_i, 1\#v_j] \mid v_i, v_j \in B_{n-1} \text{ and } [v_i, v_j] \in$
$E_{n-1}^X\} \cup \{[00\#v_i, 11\#v_j], [01\#v_i, 10\#v_j] \mid v_i, v_j \in B_{n-2} \text{ and}$
$v_i = v_j\}$

□



(a) Q_3^X (b) Q_3^C

Figure 5: Q_3^X and Q_3^C



(a) Q_4^X (b) Q_4^C

Figure 6: Q_4^X and Q_4^C

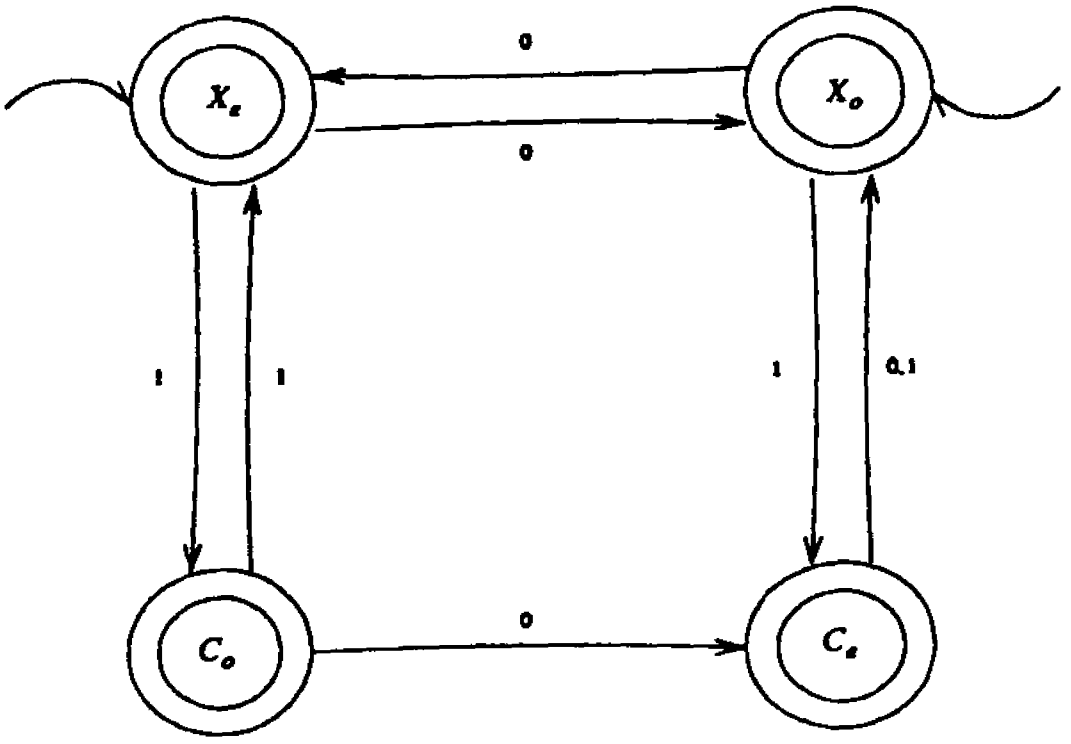


Figure 7: Automata A_n

In figure 5 and figure 6, we show several hypercubes and X-hypercubes of low dimensions. For reasons that will soon be apparent, we define a finite state automata $A_n = (S, B, T, q_0, F)$, where $S = \{X_e, X_o, C_e, C_o\}$ is the set of states in A_n; $B = \{0, 1\}$ is the input alphabet; q_0 is the initial state ($q_0 = X_e$ if n is even, and $q_0 = X_o$ if n is odd); T is the transition function $S \times B$ to S, and $F = S$ is the set of final states. The transition function T is defined in the transition diagram shown in figure 7. One additional constrain on A_n is that the binary strings that can be accepted by A_n have length no longer than n. We define $prefix(n, v; d)$ as the substring of v obtained by deleting the rightmost d bits of v. We say that $prefix(n, v; d)$ is of type X_e (or X_o, C_e and C_o) if by left-to-right scanning $prefix(n, v; d)$ the state X_e (or X_o, C_e and C_o respectively) of A_n is reached after $n-d$ state transitions. For two distinct binary strings $u = u_n u_{n-1} \ldots u_1$ and $v = v_n v_{n-1} \ldots v_1$, we define $d(n, u, v)$ as the maximum i such that $u_i \neq v_i$.

Definition 2.2:

A X-hypercube of dimension n is a graph Q_n^X with 2^n vertices, each of which is labeled by a distinct n-bit binary number. Any two vertices, $u = u_n u_{n-1} \ldots u_1$ and $v = v_n v_{n-1} \ldots v_1$, are connected by an edge if and only if one of the following two conditions holds:

(1) $prefix(n, u; d(n, u, v))$ is of type C_e $u_{d(n,u,v)} u_{d(n,u,v)-1} = \overline{v_{d(n,u,v)} v_{d(n,u,v)-1}}$, and $u_i = v_i$ for $i \neq d(n, u, v)$, and $i \neq d(n, u, v)-1$;

(2) $prefix(n, u; d(n, u, v))$ is not of type C_e and $u_i = v_i$ for $i \neq d(n, u, v)$.

□

To verify the equivalence of definition 2.1 and definition 2.2, let us look more closely at the structure of the X-hypercube. For Q_n^X, we call the subgraph induced by a vertex subset $\{b_n b_{n-1} \ldots b_1 \mid b_n b_{n-1} \ldots b_{d+1} = c_n c_{n-1} \ldots c_{d+1}\}$, where $c_n c_{n-1} \ldots c_{d+1}$ is a constant and $1 \leq d < n$, a *d-dimensional subcube of* Q_n^X. *Clearly,* Q_n^X is recursively defined by its subcube. We say that a d-dimensional subcube of Q_n^X induced by a vertex subset $\{b_n b_{n-1} \ldots b_1 \mid b_n b_{n-1} \ldots b_{d+1} = c_n c_{n-1} \ldots c_{d+1}\}$ is of type X_e (C_e) if by looking at the d least significant bits of the labels, the connections of vertices satisfy the definition of Q_d^X (Q_d^C) and d is an even number. Similarly, we define types X_o and C_o of a d-dimensional subcube of Q_n^X. Note that the only difference between X_o and X_e (C_o and C_e) is that d is an odd number for X_o (C_o). Directly following definition 2.1, we know that two vertices u and v of Q_n^X are connected by an edge if and only if they are connected by an edge in the subcube of the smallest dimension that contains both of u and v. Thus, the problem of determining whether or not u and v are connected is reduced to determining whether or not they are connected in the smallest subcube containing them. It is easy to see that $d(n,u,v)$ indicate the dimension of the smallest subcube of Q_n^X that contains u and v, and the type of *prefix(n,v;d(n,u,v))* tells the type of such a subcube. By definition 2.1, we conclude that

Theorem 2.1: Definition 2.1 and definition 2.2 for X-hypercube are equivalent.

□

It should be pointed out that the type of $prefix(n,u;d)$ can be effectively computed using A_n. To determine the type of $prefix(n,u;d)$, we need to scan u =

$u_n u_{n-1} \ldots u_1$ from left to right and make $n-d$ state transitions in A_n. The automata A_n, together with the notion *prefix* $(n, v; d)$, is not only useful for determining whether or not two vertices in Q_n^X are connected by an edge, it can also be used efficiently solving the following decision problems:

(i) Given a vertex u in Q_n^X, determine all its adjacent vertices in Q_n^X;

(ii) Given a vertex $u = u_n u_{n-1} \ldots u_1$ in Q_n^X, determine the vertex $v = v_n v_{n-1} \ldots v_1$ that is connected to u such that $u_i = v_i$ for $n \geq i > d$ and $u_d \neq v_d$; and

(iii) Given a vertex u in Q_n^X, determine the types of all subcube of Q_n^X that contain u.

All these operations are useful for investigating the algorithmic aspects and combinatoric structures of X-hypercube machines. For example, we may define an edge connecting two vertices $u = u_n u_{n-1} \ldots u_1$ and $v = v_n v_{n-1} \ldots v_1$ in Q_n^X such that $u_i = v_i$ for $n \geq i > d$ and $u_d \neq v_d$ as a *d-dimensional edge of Q_n^X*. The operation (ii) can be used to find all d-dimensional edges of Q_n^X. The above listed operations are very useful for the divide-and-conquer paradigm for designing efficient parallel algorithms on X-hypercube machines, as indicated in the previous studies on conventional hypercube machines.

2.3. Embedding of a Hypercube and Its Application

In this section, we show how the new definition can be used to derive results in the computational aspects of the X-hypercube and the conventional hypercube. First, let us consider embeddings between the hypercube and the X-hypercube.

Let G and H be two simple undirected graphs. An *embedding* of G in H is a one-to-one mapping of the vertices of G into the vertices of H, together with a specification of the path in H connecting the images of the endpoints of each edge in G. The *dilation* of the embedding is the maximum length of these paths in H, and the *congestion* of the embedding is the maximum number of edges of G whose corresponding mapped paths in H include a single edge in H. Graph embeddings can be used as a model for simulating one computer architecture by another. The parameters dilation and congestion are used to measure the efficiencies of such simulations. The following algorithm embeds a n-dimensional hypercube Q_n into a n-dimensional X-hypercube Q_n^X.

```
procedure EMBED1( Q_n, Q_n^X)
  for every edge [x, y]do
    d = d(n,x,y);
    if the type of prefix(n,x;d) is C_e then
      case (x_d x_{d-1}, y_d y_{d-1}) of
        (00,10),(10,00) : z_d z_{d-1} = 01;
        (01,11),(11,01) : z_d z_{d-1} = 10;
      endcase
      let z_i = x_i for i ≠ d and i ≠ d-1;
      associate [x, z] and [z, y] in Q_n^X with [x, y] in Q_n;
    else
      associate [x, y] in Q_n^X with [x, y] in Q_n;
    endif
  endfor
end EMBED1
```

For embedding Q_n^X into a Q_n, we give the following algorithm:

```
procedure EMBED2( Q_n^X, Q_n )
  for every edge [x, y]do
    d = d(n,x,y);
    if the type of prefix(n,x;d) is C_e then
      case (x_d x_{d-1}, y_d y_{d-1}) of
```

```
            (01,10),(10,01) : z_d z_{d-1} = 00;
            (00,11),(11,00) : z_d z_{d-1} = 10;
          endcase
          let z_i = x_i for i ≠ d and i ≠ d-1;
          associate [x, z] and [z, y] in Q_n with [x, y] in Q_n^X;
        else
          associate [x, y] in Q_n with [x, y] in Q_n^X;
        endif
      endfor
    end EMBED2
```

Theorem 2.2: Q_n can be embedded into Q_n^X with dilation 2 and congestion 2, and Q_n^X can be embedded into Q_n with dilation 2 and congestion 2.

Proof: Since the proof for two parts of the theorem are similar, we only give the proof for the first part, i.e., Q_n can be embedded into Q_n^X with dilation 2 and congestion 2. Obviously, the embedding constructed by algorithm *EMBED* 1 is of dilation 2. By algorithm *EMBED* 1, we know that any d-dimensional edge in Q_n is either mapped to the edge in Q_n^X connecting the vertices with the same labels, or mapped to two d-dimensional edges, x and y, in a d-dimensional subcube of type C_e of Q_n^X. Thus, we only need to consider mappings of d-dimensional edges $[x, y]$ to d-dimensional edges. Note that the value of d under consideration is even. If prefix(n,x;d) is C_e, then the edge $[x, y]$ in Q_n^X, where $x_d x_{d-1} = 01(10)$, $y_d y_{d-1} = 10(01)$, and $x_i = y_i$ for $i \neq d$ and $i \neq d-1$, is used exactly twice in the embedding, and the edge $[x, y]$ in Q_n^X, where $x_{d-1} \neq y_{d-1}$, and $x_i = y_i$ for $i \neq d$-1, is also used exactly twice in the embedding. Therefore, the congestion of the embedding constructed by algorithm *EMBED* 1 is 2.

□

By theorem 2.2, we know that any algorithm on a hypercube machine can be implemented on a X-hypercube machine with almost the same performance. For

example, sorting 2^n numbers on a n-dimensional hypercube machine requires $O(n^2)$ time. This can also be achieved on a n-dimensional X-hypercube easily by embedding as follows. The symbol $\Leftarrow$ denotes such a communication of a data item from an adjacent processor's local memory to the active processor's local memory.

```
procedure BITONIC MERGE SORT
  for i=0 to n-1 do
    for j=i downto 0 do
      d=2^j;
      for all P_k where 0 ≤ k ≤2^n -1 do
        if k mod 2d < d then
          if prefix(n,k;j+1) is of type C_e then
            q=2^(j-1);
            t_(k+q) ⇐a_(k+d);
            t_k ⇐t_(k+q);
          else
            t_k ⇐a_(k+d);
          endif
          if k mod 2^(i+2) < 2^(i+1) then
            b_k =max(t_k ,a_k );
            a_k =min(t_k ,a_k );
          else
            b_k =min(t_k ,a_k );
            a_k =max(t_k ,a_k );
          endif
        endif
        if k mod 2d ≥ d then
          if prefix(n,k;j+1) is of type C_e then
            q=2^(j-1);
            t_(k-q) ⇐b_(k-d);
            a_k ⇐t_(k-q);
          else
            a_k ⇐b_(k-d);
          endif
        endif
      endfor
    endfor
  endfor
end BITONIC MERGE SORT
```

2.4. Summary

We have presented an alternative formal definition for X-hypercube. As with the definition of a conventional hypercube, this concise definition explicitly provides the conditions of connectivity of vertices. As examples, we have shown how to derive simple proofs of some known results on embeddings between a hypercube and a X-hypercube. We also have shown how to use the connectivity conditions given in the definition to express the bitonic merge sort algorithm for a X-hypercube. This new definition will be very useful for further research in the parallel computation on X-hypercube interconnection networks and multicomputer systems, as demonstrated in subsequent chapters.

Chapter 3

Finding a Shortest Path in a X-Hypercube

3.1. Introduction

In this chapter, we investigate the problem of finding a shortest path between two vertices in a X-hypercube. This problem corresponds to data communication between two processors in a multicomputer system with a X-hypercube interconnection structure. Let $(u_1, u_2, ..., u_r)$ be a shortest path from u_1 to u_r computed by an algorithm A. We say that algorithm A computes this path in linear order if vertex u_{i+1} is generated after u_i is generated by A; otherwise, we say that algorithm A computes this path in non-linear order. An efficient algorithm that computes shortest paths in linear order is desirable in a packet switching network with distributed control. For the circuit switching networks of centralized control, the order in which the vertices on the shortest path are generated by an algorithm is not important.

In [Sun88], a simple mechanism for routing a message from one processor to another processor in a multicomputer system with a Q_n^X interconnection is given. Each processor P_i has a precomputed routing table of 2^n entries stored in its local memory, with each entry t_j keeping the address (binary label) k of the processor P_k connected to P_i that is the immediate successor of P_i in a shortest path from P_i to P_j. Although this method supports fast packet switching data transmission, it consumes considerable space in each local memory.

We present shortest path algorithms which compute a shortest path between any two vertices of Q_n^X in non-linear order and linear order in $O(n)$ and $O(n^2)$ time,

respectively. These algorithms can be used for data communication in a multicomputer system with a X-hypercube interconnection network. For convenience, we define a single vertex as a 0-dimensional X-hypercube, and its type is not important.

We define an *m-dimensional subcube* (or in short *m−cube*) of Q_n^X as a subgraph of Q_n^X induced by vertices in $b_n \ldots b_{m+1}\#B_m$, where $b_n \ldots b_{m+1}$ is a binary constant of $n-m$ bits. Clearly, Q_n^X is uniquely partitioned into exactly 2^{n-m} disjoint m-cube. We use S_m to denote an m-cube of Q_n^X, and use $type(S_m)$, which is of value either X or C, to denote the type of S_m. We say that two m-cubes, S_m^1 and S_m^2 of Q_n^X, are *adjacent* if there exists a vertex u in S_m^1 that is connected to a vertex v in S_m^2. In fact, by the definition of a X-hypercube, if two m-cubes, S_m^1 and S_m^2 of Q_n^X, are adjacent, then the vertices of S_m^1 and vertices in S_m^2 are pairwisely connected. More specifically, two distinct m-cubes, S_m^1 and S_m^2 of Q_n^X, are adjacent if and only if vertex $u = u_n u_{n-1} \ldots u_1$ in S_m^1 and vertex $v = v_n v_{n-1} \ldots v_1$ in S_m^2 such that $u_m u_{m-1} \ldots u_1 = v_m v_{m-1} \ldots v_1$ are connected by an edge. The concept of a subcube is very important in analyzing the structure and connectivity of a X-hypercube. We are particularly interested in the subcubes of even dimension. We call a sequence $P_m = (S_m^1, S_m^2, \ldots, S_m^r)$ of distinct m-dimensional subcubes, where m must be even, of Q_n^X as an *m-dimensional subcube path* (in short, a *m−cube path*) from S_m^1 to S_m^r in Q_n^X if S_m^i and S_m^{i+1} are adjacent. We also use $P_m = (S_m^1, S_m^2, \ldots, S_m^r)$ to represent the subgraph of Q_n^X induced by vertices of S_m^i, $1 \le i \le r$. In a m-cube path, each subcube S_m^i can be considered as a "super vertex". For $P_m = (S_m^1\ S_m^2, \ldots, S_m^r)$, we define $type\ (P_m) = type(S_m^1) type(S_m^2) \ldots type(S_m^r)$. Using regular expression notation, we know that $type(P_m) \in (X \mid C)^+$. We say that an m-cube path $P_m = (S_m^1\ S_m^2, \ldots, S_m^r)$ is *hybrid* if it contains m-cube of

different types, otherwise, we say it is a uniform m-cube path. For any vertex u in Q_n^X we use $u(S_m)$ to denote the vertex u' in S_m such that $u'_m u'_{m-1} \ldots u'_1 = u_m u_{m-1} \ldots u_1$, and we use $S_m(u)$ to denote the m-dimensional subcube of Q_n^X that contains u. Obviously, if two m-cubes, S_m^1 and S_m^2, are adjacent in Q_n^X, then $u(S_m^1)$ and $u(S_m^2)$ are connected by an edge. For a subgraph G of Q_n^X that contains m-cube S_m^1 and S_m^2 of Q_n^X, we use $P_m[S_m^1, S_m^2; G]$ and $P_m^*[S_m^1, S_m^2; G]$ to denote an m-cube path and a shortest m-cube path from S_m^1 to S_m^2 in G, respectively, if they exist.

Thus, our objective is to find a shortest 0-cube path $P_0^*[s,d; Q_n^X]$ from s, the *source vertex*, to d, the *destination vertex*. For simplicity, we denote such a 0-cube path as $P^*[s,d; Q_n^X]$. Given two m-cube paths $P_m^1[S_m^1, S_m^2; G_1]$ and $P_m^2[S_m^2, S_m^3; G_2]$ in two subgraphs G_1 and G_2 of Q_n^X such that S_m^2 is a subgraph of G_1 and G_2, we use | as the concatenation operator for these two m-cube paths. That is, $P_m^1[S_m^1, S_m^2; G_1] \mid P_m^2[S_m^2, S_m^3; G_2]$ is an m-cube path from S_m^1 to S_m^3 in Q_n^X obtained by concatenating $P_m^1[S_m^1, S_m^2; G_1]$ and $P_m^2[S_m^2, S_m^3; G_2]$. The length of a path $P_m[S_m^1, S_m^2; G]$, denoted as $L(P_m[S_m^1, S_m^2; G])$, is the number of m-cubes in $P_m[S_m^1, S_m^2; G]$ minus 1. All these notations are similarly defined with respect to Q_n^C.

For convenience, we define the graph $C_m = (Q_m^1, Q_m^2, \ldots Q_m^r)$ as an *m-cube chain*, where m is even and Q_m^i is a graph whose vertices are labeled by m-bit binary numbers satisfying the edge connections of either Q_m^X or Q_m^C. Furthermore, any vertex in Q_m^i labeled u is connected to the vertex in Q_m^{i+1} labeled u, but not connected to any vertex in Q_m^j such that $j > i+1$. Vertex u in Q_m^i is denoted as $u(Q_m^i)$. Note that an m-cube path $P_m = (S_m^1, S_m^2, \ldots, S_m^r)$ in Q_n^X is an m-cube chain, if we ignore the $n-m$

leftmost bits of all vertices in P_m. However, for a given chain C_m, it is not always possible to find a subgraph in Q_n^X that is isomorphic to C_m. We call a chain $C_m = (Q_m^1, Q_m^2, \ldots Q_m^r)$ a *hybrid chain* if there exists i and j such that $type(Q_m^i) \neq type(Q_m^j)$. Note that m-cube paths P_m and m-cube chains C_m are defined only for even m. For an m-chain $C_m = (Q_m^1, Q_m^2, \ldots Q_m^r)$, we use $P^*[Q_m^i, Q_m^j; C_m]$ to denote a shortest path from $u(Q_m^i)$ to $u(Q_m^j)$. The concatenation operator "|" is defined for paths in m-cube chains in the same way as that for the paths in m-cube paths.

3.2. Shortest Path Algorithms

Although the X-hypercubes Q_n^X are recursively defined, their structure is not as regular as that for a corresponding conventional hypercube. We first derive a couple of simple properties for Q_n^X and Q_n^C. Then, we show how to use these properties to design efficient shortest path algorithms.

Lemma 3.1: The following statements hold:

(a) $|L(P^*[u,v; Q_n^X]) - L(P^*[u,v; Q_n^C])| \leq 1$ for even n;

(b) $L(P^*[u,v; Q_n^X]) = L(P^*[u,v; Q_n^C])$, where u and v are in different $(n-1)$-dimensional subcubes, for odd n.

(c) For two m-cube chains $C_m^1 = (Q_m^{1,1}, Q_m^{1,2}, \ldots, Q_m^{1,r})$ and $C_m^2 = (Q_m^{2,1}, Q_m^{2,2}, \ldots, Q_m^{2,s})$,

(i) if $r = s$ and both C_m^1 and C_m^2 are hybrid, or both C_m^1 and C_m^2 are uniform, then

$$L(P^*[u(Q_m^{1,1}), v(Q_m^{1,r}); C_m^1]) = L(P^*[u(Q_m^{2,1}), v(Q_m^{2,s}); C_m^2]);$$

(ii) if $r = s$, C_m^1 is hybrid and C_m^2 is uniform, then

$$L(P^*[u(Q_m^{1,1}),v(Q_m^{1,r});C_m^1]) \le L(P^*[u(Q_m^{2,1}),v(Q_m^{2,s});C_m^2]);$$

(iii) if $r > s$, then $L(P^*[u(Q_m^{1,1}),v(Q_m^{1,r});C_m^1]) \ge L(P^*[u(Q_m^{2,1}),v(Q_m^{2,s});C_m^2])$.

Proof: The proof is by induction. To simplify our proof, we divide our proof into three parts.

Part 1: (c) holds if (a) and (b) are true for $n \le m+1$.

Consider (i) first. If both C_m^1 and C_m^2 are not hybrid, the (i) of (c) is obviously true. The proof for the case that both C_m^1 and C_m^2 are hybrid directly follows from the following claim:

Claim: For any hybrid m-cube chain $C_m = (Q_m^1, Q_m^2, ..., Q_m^t)$, let i be an index such that *type* $(Q_m^i) \ne$ *type* (Q_m^{i+1}). If (b) is true for $n \le m+1$, then

$$(u(Q_m^1), ..., u(Q_m^i)) \mid P^*[u(Q_m^i), v(Q_m^{i+1}); (Q_m^i, Q_m^{i+1})] \mid (v(Q_m^{i+1}), ..., v(Q_m^t))$$

is a shortest path from $u(Q_m^1)$ to $v(Q_m^t)$ in C_m.

Proof of the Claim: Let $C_m = (Q_m^1, Q_m^2, ..., Q_m^t)$ be any m-cube chain that contains l m-cube of type X and $t-l$ m-cube of type C. We define the operation of interchanging the positions of two adjacent m-cube Q_m^i and Q_m^{i+1} of C_m as follows: delete edges connecting $u(Q_m^{i-1})$ and $u(Q_m^i)$ and edges connecting $u(Q_m^{i+1})$ and $u(Q_m^{i+2})$, introduce edges connecting $u(Q_m^{i-1})$ and $u(Q_m^{i+1})$ and edges connecting $u(Q_m^i)$ and $u(Q_m^{i+2})$. Clearly, after this operation, we obtain a new m-cube chain if original two m-cube Q_m^i and Q_m^{i+1} are of different

types. By performing a sequence of operations of interchanging the positions of two adjacent m-cube of different types, we can obtain a new chain $C'_m = (Q'^1_m, Q'^2_m, ..., Q'^t_m)$, such that $type(C'_m) = X^l C^{t-l}$. By (b), we know that (i) holds for C_m and any intermediate m-cube chain obtained in this transformation process. Hence

$$L(P^*[(u(Q^1_m), v(Q^t_m); C_m]) = L(P^*[(u(Q'^1_m), v(Q'^t_m); C'_m]).$$

Consider any shortest path

$$\begin{aligned} & P^*[(u(Q'^1_m), v(Q'^t_m); C'_m] \\ = & P^*[u(Q'^1_m), w(Q'^l_m); (Q'^1_m, ..., Q'^l_m)] \mid (w(Q'^l_m), w(Q'^{l+1}_m)) \mid \\ & P^*[w(Q'^{l+1}_m), v(Q'^t_m); (Q'^{l+1}_m, ..., Q'^t_m)]. \end{aligned}$$

By the definition of X-hypercube, we know that

$$\begin{aligned} P^1 = & (u(Q'^1_m), ..., u(Q'^l_m)) \mid P^*[u(Q'^l_m), w(Q'^l_m); Q'^l_m] \mid \\ & (w(Q'^l_m), w(Q'^{l+1}_m)) \mid P^*[w(Q'^{l+1}_m), v(Q'^{l+1}_m); Q'^{l+1}_m] \mid \\ & (v(Q'^{l+1}_m), ..., v(Q'^t_m)) \end{aligned}$$

is also a shortest path from $u(Q'^1_m)$ to $v(Q'^X_m)$ in C'_m. Let

$$\begin{aligned} P^2 = & (u(Q'^1_m), ..., u(Q'^l_m)) \mid P^*[u(Q'^l_m), v(Q'^{l+1}_m); (Q'^l_m, Q'^{l+1}_m)] \\ & \mid (v(Q'^{l+1}_m), ..., v(Q'^t_m)). \end{aligned}$$

Clearly, $L(P^1) = L(P^2)$. This implies that

$$(u(Q^1_m), ..., u(Q^i_m)) \mid P^*[u(Q^i_m), v(Q^{i+1}_m); (Q^i_m, Q^{i+1}_m)] \mid (v(Q^{i+1}_m), ..., v(Q^t_m))$$

where i is an integer such that $type(Q_m^i) \neq type(Q_m^{i+1})$, is a shortest path $P^*[u(Q_m^1), v(Q_m^t); C_m]$. This completes the proof of the claim.

Applying this claim to C_m^1 and C_m^2, we know that (i) holds if (b) is true for $n \leq m+1$. Then, (ii) and (iii) of (c) directly follow (a), (b) and the proof of above claim.

Part 2: (a) and (b) holds for $n = k$ if (a) holds for $n < k$ and (c) holds for all even m such that $m < k$.

First, let us consider (a). Since (a) and (c) hold for all even m such that $m < n$, we only need to consider the case that u and v are in different $(n-2)$-dimensional subcubes. By (c), it is easy to see that we only need to consider the shortest paths from u to v in the shortest $(n-2)$-cube paths from $S_{n-2}(u)$ to $S_{n-2}(v)$. By inspection, we know that for any pair of $S_{n-2}(u)$ and $S_{n-2}(v)$, any $P_{n-2}^*[S_{n-2}(u), S_{n-2}(v); Q_n^X]$ (resp. $P_{n-2}^*[S_{n-2}(u), S_{n-2}(v); Q_n^C]$) is hybrid. Since $|L(P_{n-2}^*[S_{n-2}(u), S_{n-2}(v); Q_n^X]) - L(P_{n-2}^*[S_{n-2}(u), S_{n-2}(v); Q_n^C])| \leq 1$, by (i) and (ii) of (c), we know that (a) holds.

Consider (b). Since (a) and (c) hold for all even m such that $m < n$, we only need to consider the case that u and v are in different $(n-1)$-dimensional subcubes. By inspection, we can easily verify that for any u and v that are in the different $(n-1)$-dimensional subcubes of Q_n^X (resp. Q_n^C), $P_{n-3}^*[S_{n-3}(u), S_{n-3}(v); Q_n^X]$ and $P_{n-3}^*[S_{n-3}(u), S_{n-3}(v); Q_n^C]$ consists of the same number of $(n-3)$-cubes, $P_{n-3}^*[S_{n-3}(u), S_{n-3}(v); Q_n^X]$ is hybrid if and only if $P_{n-3}^*[S_{n-3}(u), S_{n-3}(v); Q_n^C]$ is hybrid. Summarizing these simple facts, we know that (b) holds for $n = k$ if if (a) holds for $n < k$ and (c) holds for all even m such that $m < k$.

Part 3: (a), (b) and (c) hold simultaneously.

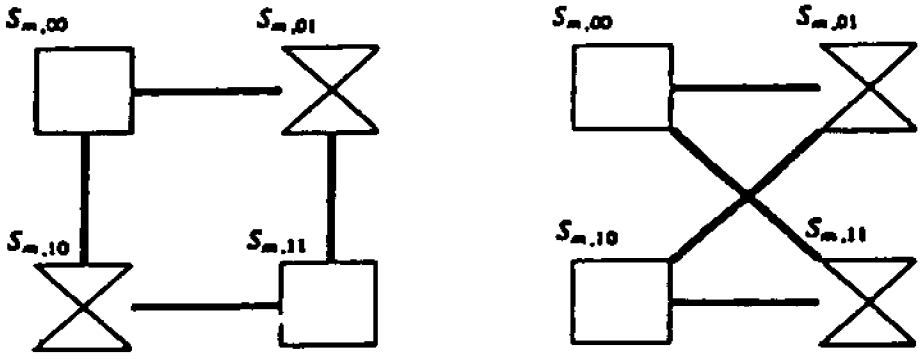
The proof is by induction. By inspection, we know that (a) and (b) hold for $n = 2$ and $n = 3$, respectively. By *Part 1* we know that (c) holds for $m = 2$. Then, by *Part 2*, we know that (a) holds for $n = 4$, and (b) holds for $n = 5$. The induction is carried out by alternatively using (a),(b), and (c). This completes the proof of the lemma.

□

Lemma 3.1 is a fundamental fact regarding the length of shortest paths in Q_n^X and Q_n^C. Suppose that for a given pair u and v all shortest m-cube paths $P_m[S_m(u), S_m(v); Q_n^X]$ are hybrid. Lemma 3.1 implies that the problem of finding a shortest path $P^*[u, v; Q_n^X]$ can be reduced to the problem of finding a shortest path $P^*[u', v'; S_{m+1}]$ as follows. For hybrid m-cube path $P_m[S_m(u), S_m(v); Q_n^X]$, let i be an integer such that $type(S_m^i) \neq type(S_m^{i+1})$. Then, by the proof of lemma 3.1,

$$(u(S_m^1), u(S_m^2), \ldots, u(S_m^i)) \mid P^*[u(S_m^i), v(S_m^{i+1}); (S_m^i, S_m^{i+1})] \mid (v(S_m^{i+1}), v(S_m^{i+2}), \ldots, v(S_m^r))$$

is a shortest path from u to v in Q_n^X.



(a) S_m of the type X (b) S_m of the type C

Figure 8: Four (m-2)-dimensional subcubes of S_m for even m

Let $S(u,v)$ be the smallest subcube of Q_n^X (resp. Q_n^C) that contains both u and v. We use $dim(S(u,v))$ to denote the dimension of $S(u,v)$. We use $S_{m,xy}$, where $x, y \in \{0, 1\}$, to represent the four $(m-2)$-dimensional subcubes of S_m, respectively, as shown in figure 8.

Lemma 3.2: For any u and v in Q_n^X, either all shortest m-cube paths from $S_m(u)$ to $S_m(v)$ in Q_n^X are hybrid or all shortest m-cube paths from $S_m(u)$ to $S_m(v)$ in Q_n^X are uniform.

Proof: By inspection, it is easy to verify that if n is even, then the lemma holds for $m = n$ and $m = n - 2$; and if n is odd, then the lemma holds for $m = n - 1$ and $m = n - 3$. Now, we prove the lemma by induction. Suppose that the lemma holds for $m \leq k+2$, where k is an even number, and consider the case that $m = k$.

Let $P_k^*[S_k(u), S_k(v); Q_n^X] = (S_k^1, S_k^2, ..., S_k^r)$ be any shortest k-cube from $S_k(u)$ to $S_k(v)$ in Q_n^X, let $P_{k+2}^*[S_{k+2}(u), S_{k+2}(v); Q_n^X] = (S_{k+2}^1, S_{k+2}^2, ..., S_{k+2}^s)$ be any shortest $(k+2)$-cube path in Q_n^X from $S_{k+2}(u)$ to $S_{k+2}(v)$ in Q_n^X with $P_k^*[S_k(u), S_k(v); Q_n^X]$ as a subgraph. If $P_k^*[S_k(u), S_k(v); Q_n^X$ is not hybrid, then by the definition of Q_n^X, we know that it must be that $r = s$ and exactly one of the following conditions holds:

(i) $S_k^i = S_{k+2,00}^i$ for $1 \leq i \leq r$;

(ii) $S_k^i = S_{k+2,01}^i$ for $1 \leq i \leq r$;

(iii) $S_k^i = S_{k+2,10}^i$ for $1 \leq i \leq r$ and all $S_{k+2,10}^i$ have the same type;

(iv) $S_k^i = S_{k+2,11}^i$ for $1 \leq i \leq r$ and all $S_{k+2,11}^i$ have the same type.

Let $P'_k[S_k(u), S_k(v); Q_n^X] = (S'^1_k, S'^2_k, ..., S'^r_k)$ be any shortest k-cube path among all

hybrid k-cube paths from $S_k(u)$ to $S_k(v)$ in Q_n^X and let $P'_{k+2}[S_{k+2}(u), S_{k+2}(v); Q_n^X] = (S'^1_{k+2}, S'^2_{k+2}, ..., S'^q_{k+2})$ be the shortest $(k+2)$-cube path in Q_n^X from $S_{k+2}(u)$ to $S_{k+2}(v)$ with $P'_k[S'_k(u), S'_k(v); Q_n^X]$ as a subgraph. Then, none of the following conditions can hold:

(a) $S'^i_k = S'^i_{k+2,00}$ for $1 \le i \le p$;

(b) $S'^i_k = S'^i_{k+2,01}$ for $1 \le i \le p$;

(c) $S'^i_k = S'^i_{k+2,10}$ for $1 \le i \le p$ and all $S'^i_{k+2,10}$ have the same type;

(d) $S'^i_k = S'^i_{k+2,11}$ for $1 \le i \le p$ and all $S^i_{k+2,11}$ have the same type.

Clearly, $p \ge r$. If u is in $S^1_{k+2,00}$ or u is in $S^1_{k+2,01}$, then since (a) and (b) do not hold, there must be some i such that $1 \le i < p$ and S'^i_k and S'^{i+1}_k are in S'^i_{k+2}. This implies that $p > r$, and P'_k is not a shortest k-cube path from $S_k(u)$ to $S_k(v)$.

If u is in $S^1_{k+2,10}$ or u is in $S^1_{k+2,11}$, then since (c) and (d) do not hold there are two possibilities. The first possibility is that there exists an i such that $S'^i_k \le S'^i_{k+2,10}$ if u is in $S^1_{k+2,10}$ or $S'^i_k \le S'^i_{k+2,11}$ for some i if u is in $S^1_{k+2,11}$. Then, there must be some j such that $1 \le j < p$ and S'^j_k and S'^{j+1}_k are in S'^i_{k+2}. This implies that $p > r$, and P'_k is not a shortest k-cube path from $S_k(u)$ to $S_k(v)$. The second possibility is that not all $S'^i_{k+2,10}$ have the same type or not all $S'^i_{k+2,11}$ have the same type. Then, by the definition of Q_n^X we know that not all S'^i_{k+2} have the same type. Since u is in $S^1_{k+2,10}$ or u is in $S^1_{k+2,11}$, by (iii) and (iv) we know that all S^i_{k+2}, $1 \le i \le r$, in $P_k[S_k(u), S_k(v); Q_n^X]$ are of the same type. If $p = r$, then we have two shortest k-cube paths P_k and P'_k, one is hybrid and the other is uniform, from $S_k(u)$ to $S_k(v)$ in Q_n^X. This contradicts the inductive hypothesis that the lemma holds for $m = k+2$.

Therefore, $p > r$. Summarizing these discussions, we know that the lemma holds for $m = k$. This completes the induction and the proof of the lemma.

□

We use $dim(S(u,v))$ to denote the dimension of $S(u,v)$. The following fact allow us to further simplify the process of finding a shortest path from any vertex u to any other vertex v in Q_n^X (resp. Q_n^C).

Lemma 3.3: For any two vertices u and v in Q_n^X (resp. Q_n^C), P^*[u, v; $P_m^*[S_m(u), S_m(v); S(u,v)]$] is a shortest path P^*[u, v; Q_n^X] (resp. P^*[u, v; Q_n^C]) for any even $m < dim(S(u,v))$ and any $P_m^*[S_m(u), S_m(v); S(u,v)]$.

Proof: If $dim(S(u,v))$ is even, then let $j = dim(S(u,v)) - 2$; otherwise let $j = dim(S(u,v)) - 3$. Let k be the number of j-cubes of $S(u,v)$ in any shortest j-cube path $P_j^*[S_j(u), S_j(v); S(u,v)]$. Clearly, k is equal to 2 or 3. Let k' be the number of j-cube of Q_n^X (resp. Q_n^C) in any j-cube path $P_j[S_j(u), S_j(v); Q_n^X]$ (resp. $P_j[S_j(u), S_j(v); Q_n^X]$) that contains at least one j-cube not in $S(u,v)$. It is easy to verify that $k' > k$. By lemma 3.1 (and its proof), we know that the lemma is true for $m = dim(S(u,v)) - 2$, if $dim(S(u,v))$ is even, and for $m = dim(S(u,v)) - 1$ and $m = dim(S(u,v)) - 3$, if $dim(S(u,v))$ is odd.

Suppose the lemma holds for all even m values such that $dim(S(u,v)) > m > k$, where k is an even number greater than zero, we want to prove that the lemma holds for $m = k$. Consider any shortest m-cube path, $m = k+2$, $P_{k+2}^*[S_{k+2}(u), S_{k+2}(v); S(u,v)] = (S_{k+2}^1, S_{k+2}^2, ..., S_{k+2}^r)$. Let $u_1 = u$, and u_i be the vertex in S_{k+2}^i that is con-

nected with u_{i-1} for $1 < i \leq r$; and let $v_r = v$, and v_{i-1} be the vertex in S^{i-1}_{k+2} that is connected with v_i for $1 < i \leq r$. There are three cases that need to be considered.

Case 1: $r = 1$ or $r = 2$ and type $(S^1_{k+2}) \neq$ type (S^2_{k+2}).

If $r = 1$, then either $S_k(u) = S_k(v)$ or all $P^*_k[S_k(u), S_k(v); S(u,v)]$ are hybrid. If $r = 2$ and *type* $(S^1_{k+2}) \neq$ *type* (S^2_{k+2}), then all $P^*_k[S_k(u), S_k(v); S(u,v)]$ are either hybrid or uniform. By lemma 3.1, we know that for any of these possibilities this lemma holds.

Case 2: All S^i_{k+2} of $P^*_{k+2}[S_{k+2}(u), S_{k+2}(v); S(u,v)]$ are of the same type.

By the definition of a X-hypercube and the assumption that all S^i_{k+2} of $P^*_{k+2}[S_{k+2}(u), S_{k+2}(v); S(u,v)]$ are of the same type, we know that $(u_1, u_2, ..., u_r) \mid P^*[u_r, v_r; S^r_{k+2}]$ must be a shortest path $P^*[u_1, v_r; (S^1_{k+2}, S^2_{k+2}, ..., S^r_{k+2})]$, and by case 1, we know that $P^*[u_r, v_r; P^*_k[S_k(u_r), S_k(v_r); S^r_{k+2}]]$ is a shortest path $P^*[u_r, v_r; S^r_{k+2}]$. It is easy to verify that $P^*_k[S_k(u_r), S_k(v_r); S^r_{k+2}]]$ is hybrid if and only if any k-cube path from $S_k(u_1)$ to $S_k(v_r)$ in $(S^1_{k+2}, S^2_{k+2}, ..., S^r_{k+2})$ is hybrid. Hence, the k-cube path $P'_k = (S_k(u_1), S_k(u_2), ..., S_k(u_r)) \mid P^*_k[S_k(u_r), S_k(v_r); S^r_{k+2}]$ must be a shortest k-cube path from $S_k(u_1)$ to $S_k(v_r)$ in $(S^1_{k+2}, S^2_{k+2}, ..., S^r_{k+2})$, since otherwise $(u_1, u_2, ..., u_r) \mid P^*[u_r, v_r; P^*_k[S_k(u_r), S_k(v_r); S^r_{k+2}]]$ is not a shortest path from u_1 to v_r in $(S^1_{k+2}, S^2_{k+2}, ..., S^r_{k+2})$.

Case 3: $P^*_{k+2}[S_{k+2}(u), S_{k+2}(v); S(u,v)]$ is a hybrid $(k+2)$-cube path.

First, let us consider a special case $P^*_{k+2}[S_{k+2}(u), S_{k+2}(v); S(u,v)] = (S^1_{k+2}, S^2_{k+2}, ..., S^r_{k+2})$ such that *type* $(P^*_{k+2}) = X^{r-1}C$ or $C^{r-1}X$. By the proof of lemma 3.1, $(u_1, u_2, ..., u_{r-1} \mid P^*[u_{r-1}, v_r; (S^{r-1}_{k+2}, S^r_{k+2})]$ is a $P^*[u, v; P^*_{k+2}[S_{k+2}(u), S_{k+2}(v); S(u,v)]]$.

By case 1 and arguments similar to case 2, we know that there exists a P^*[u, v; $S(u,v)$] in $(S_k(u_1), S_k(u_2), ..., S_k(u_r)) \mid P_k^*[S_k(u_{r-1}), S_k(v_r); (S_{k+2}^{r-1}, S_{k+2}^r)]$, which is a P^*[u, v; $P_k^*[S_k(u), S_k(v); S(u,v)]$]. The proof for arbitrary hybrid $(k+2)$-cube paths $P_{k+2}^*[S_{k+2}(u), S_{k+2}(v); S(u,v)] = (S_{k+2}^1, S_{k+2}^2, ..., S_{k+2}^r)$ can be reduced to the above special case as follows. Let i be an index such that subcube S_{k+2}^i and S_{k+2}^{i+1} of P_{k+2}^* have different types. Then by the argument for the special case, we know that there exists a P^*[u, v; $S(u,v)$] in $(S_k(u_1), S_k(u_2), ..., S_k(u_i)) \mid P_k^*[S_k(u_i), S_k(v_{i+1}); (S_{k+2}^i, S_{k+2}^{i+1})] \mid S_k(v_{i+2}), S_k(v_{i+3}), ..., S_k(v_r)$, which is a $P_k^*[S_k(u), S_k(v); S(u,v)]$. This completes the proof of lemma 3.3.

□

Lemma 3.3 allows us to reduce the problem of finding P^*[u, v; Q_n^X] to the problem of recursively finding a shortest m-cube path $P_m^*[S_m(u), S_m(v); Q_n^X]$ in decreasing order of even m's. Such a problem can be then further reduced to the basic problem of finding a shortest m-cube path P_m^*[u, v; Q_{m+2}^X], P_m^*[u, v; Q_{m+2}^C], P_m^*[u, v; Q_{m+3}^X], and P_m^*[u, v; Q_{m+3}^C]. For two adjacent m-cubes, S_m^1 and S_m^2 of different types in Q_n^X (or Q_n^C), where m is an even number less than n, (S_m^1, S_m^2) is isomorphic to Q_{m+1}^X and Q_{m+1}^C. Thus, we can treat (S_m^1, S_m^2) as a generalized $(m+1)$-dimensional subcube of Q_n^X (resp. Q_n^C). Consider the source vertex s and destination vertex d. We define the k-dimensional generalized subcube $S'(s,d)$ of Q_n^X as follows. If $S_k(s) = S_k(d)$, then $S'(s,d) = S_k(s)$; if $S_{k-1}(s)$ and $S_{k-1}(d)$ are adjacent, then $S'(s,d) = (S_{k-1}(s), S_{k-1}(d))$, which is a subgraph of Q_n^X induced by vertices of $S_{k-1}(s)$ and $S_{k-1}(d)$; otherwise, the k-dimensional generalized subcube $S'(s,d)$ of Q_n^X does not

exist. This notion of $S'(s,d)$ is useful in the following shortest path algorithm, which finds a shortest path by recursively finding shortest subcube paths in the smallest $S'(s,d)$.

Algorithm *PATH_1*(s,d,n)
  $P_1 := P_2 := \varnothing$;
  $k := n$; $S'(s,d) = Q_n^X$;
  repeat
    while s and d are in the same $(k-1)$-cube S_{k-1} of $S'(s,d)$ **do**
      $S'(s,d) := S_{k-1}$; $k := k-1$;
    endwhile
    if $k = 1$ **then** $P_* := P_1 \mid (s, d) \mid P_2$ and **stop**
    if k is a positive even number **then** $m := k-2$ **else** $m := k-3$;
    find a shortest m-cube path $P_m^*[S_m(s), S_m(d); S'(s,d)]$;
    case
      :m = 0:
        $P^* := P_1 \mid P_m^* \mid P_2$; **stop**.
      :$P_m^*[S_m(s), S_m(d); S'(s,d)] = (S_m^1, S_m^2)$ **and** *(type* $(P_m^*) = XX$ **or** *type* $(P_m^*) = CC$*):*
        s' := the vertex in S_m^2 that is connected to vertex s;
        $P_1 := (s,s') \mid P_1$;
        $s := s'$; $S'(s,d) := S_m^2$; $k := m$;
        if $s = d$ **then** $P^* := P_1 \mid P_2$; **stop**;
      :$P_m^*[S_m(s), S_m(d); S'(s,d)] = (S_m^1, S_m^2)$ **and** *(type* $(P_m^*) = XC$ **or** *type* $(P_m^*) = CX$*):*
        $S'(s,d) := P_m^*$; $k := m+1$;
        if $s = d$ **then** $P^* := P_1 \mid P_2$; **stop**;
      :$P_m^*[S_m(s), S_m(d); S'(s,d)] = (S_m^1, S_m^2, S_m^3)$ **and** *(type* $(P_m^*) = XXC$ **or** *type* $(P_m^*) = CCX$ **or** *type* $(P_m^*) = XCX$ *or type* $(P_m^*) = CXC$*):*
        s' := the vertex in S_m^2 that is connected to vertex s
        $P_1 := (s,s') \mid P_1$; $s := s'$; $S'(s,d) := (S_m^2, S_m^3)$; $k := m+1$;
        if $s = d$ **then** $P^* := P_1 \mid P_2$; **stop**;
      :$P_m^*[S_m(s), S_m(d); S'(s,d)] = (S_m^1, S_m^2, S_m^3)$ **and** *type* $(P_m^*) = CXX$ **or** *type* $(P_m^*) = XCC$*):*
        d' := the vertex in S_m^2 that is connected to vertex d
        $P_2 := P_2 \mid (d',d)$;
        $d := d'$; $S'(s,d) := (S_m^1, S_m^2)$; $k := m+1$;
        if $s = d$ **then** $P^* := P_1 \mid P_2$; **stop**;
    endcase
  endrepeat
end *PATH_1*

Theorem 3.1: A shortest path from any vertex s to any vertex d in Q_n^X can be computed in O(n) time.

Proof: In the algorithm, $S'(s,d)$ is treated as an k-dimensional subcube of Q_n^X. When the "subcube" $S'(s,d)$ in the algorithm is of even dimension k, then $S'(s,d)$ is indeed a real subcube of Q_n^X; but when $S'(s,d)$ in the algorithm is of odd dimension k, then $S'(s,d)$ is composed of two $(k-1)$-dimensional subcube of Q_n^X, but not necessarily a real k-dimensional subcube of Q_n^X. However, $S'(s,d)$ is isomorphic to Q_k^X or Q_k^C. This difference is not important in finding a shortest m-cube path in $S'(s,d)$. The correctness of the algorithm can be directly observed using lemma 3.1 and lemma 3.2. We may construct two tables that can be used for finding shortest m-cube paths, one for even k and the other for odd k. It is important to note that $m = k-2$ if k is even, and $m = k-3$ if k is odd. Clearly, these two tables can be constructed using $k = 2$ and $k = 3$, respectively. Each table contains a shortest m-cube paths for each of all possible $S_m(s)$ and $S_m(d)$ combinations, and the types of the m-cube on the path. Clearly, each table has no more than a constant number of entries and each entry contains a constant number of fields. Therefore, finding a shortest m-cube path between two m-dimensional subcube of Q_k^X in $S'(s,d)$ in each iteration can be done in constant time. Since the total number of iterations of the *repeat-loop* and the *while-loop* is n, the total time of this algorithm is $O(n)$.

□

Algorithm *PATH_1* computes a shortest path from s to d in optimal time. However, the vertices on the path are not generated in strict linear order. Furthermore, the

implementation of this algorithm is tedious. By applying this algorithm at most $\lceil (n+1)/2 \rceil$ times, we can construct a shortest path from s to d in strict linear order in $O(n^2)$ time. In what follows, we present an algorithm which also computes a shortest path from s to d in $O(n^2)$ time. This algorithm is much simpler and faster, considering the constant factor of its complexity. Consider an m-dimensional subcube S_m of Q_n^X (Q_n^C), where m is an even number. We use $S_{m,xy}$, where $x, y \in \{0, 1\}$, to represent the four $(m-2)$-dimensional subcubes of S_m, respectively, as shown in figure 8. For the subcube $S_m(u)$, these four $(m-2)$-dimensional subcubes are $S_{m,00}(u)$, $S_{m,01}(u)$, $S_{m,01}(u)$ and $S_{m,01}(u)$, and only one of them contains u. For an m-cube chain $C_m = (Q_m^1, Q_m^2, \ldots, Q_m^r)$, and the vertices u in Q_m^1 and v in Q_m^r, we use $S_k^1(u)$ and $S_k^r(v)$ to denote the k-cube in Q_m^1 and Q_m^r that contains u and v, respectively. The following lemma is useful in obtaining additional properties of X-hypercube stated in lemma 3.4 and lemma 3.5.

Lemma 3.4: Given any m-cube chain $C_m = (Q_m^1, Q_m^2, \ldots, Q_m^r)$ such that $r \geq 2$, for any u and v, which are m-bit binary labels, there exists a shortest path $P^*[u(Q_m^1), v(Q_m^r); C_m] = (u(Q_m^1), u(Q_m^2)) \mid P^*[u(Q_m^2), v(Q_m^r); (Q_m^2, \ldots, Q_m^r)]$, if for all even k such that $2 \leq k \leq m$ none of the following two conditions holds:

(1) $type(S_k^1(u)) = X$ and $type(S_k^r(v)) = C$, u is in $S_{k,xy}^1(u)$ and v is in $S_{k,\bar{x}\bar{y}}^r(v)$; and

(2) $type(S_k^1(u)) = C$ and $type(S_k^r(v)) = X$, u is in $S_{k,xy}^1(u)$ and v is in $S_{k,\bar{x}\bar{y}}^r(v)$.

Proof: The proof is by induction. For $m = 2$, the lemma obvious holds. Suppose that

the lemma holds for $m = 2i$, $1 \le i \le j$, consider the case when $m = 2(i+1)$, i.e. any $2(i+1)$-cube chain $C_{2(i+1)} = (Q^1_{2(i+1)}, Q^2_{2(i+1)}, \ldots, Q^r_{2(i+1)})$. If $type(Q^1_{2(i+1)}(u)) = type(Q^r_{2(i+1)}(v))$, by lemma 3.1 we know that the lemma holds. Consider the case that $type(Q^1_{2(i+1)}(u)) \neq type(Q^r_{2(i+1)}(v))$. By lemma 3.2, we know that

$$P^*[u(Q^1_{2(i+1)}), v(Q^r_{2(i+1)}); P^*_{2i}[Q^1_{2i}(u), Q^r_{2i}(v); C_{2(i+1)}]]$$

is a shortest path from $u(Q^1_{2(i+1)})$ to $v(Q^r_{2(i+1)})$ in $C_{2(i+1)}$, where $P^*_{2i}[Q^1_{2i}(u), Q^r_{2i}(v); C_{2(i+1)}]$ is any shortest $2i$-cube path from $Q^1_{2i}(u)$ to $Q^r_{2i}(v)$ in $C_{2(i+1)}$. If (1) and (2) do not hold, then it is easy to verify that

$$P = (Q^1_{2i}(u), Q^2_{2i}(u)) \mid P^*_{2i}[Q^2_{2i}(u), Q^r_{2i}(v); (Q^2_{2(i+1)}, \ldots, Q^r_{2(i+1)})]$$

is a shortest $2i$-cube path from $Q^1_{2i}(u)$ to $Q^r_{2i}(v)$ in $C_{2(i+1)}$. By lemma 3.2,

$$L(P^*[u(Q^1_{2i}), v(Q^r_{2i}); P]) = L(P^*[u(Q^1_{2(i+1)}), v(Q^r_{2(i+1)}); C_{2(i+1)}]).$$

By the induction hypothesis,

$$L(P^*[u(Q^1_{2i}), v(Q^r_{2i}); P]) = L(P^*[u(Q^1_{2i}), u(Q^2_{2i}) \mid P^*[u(Q^2_{2i}), v(Q^r_{2i}); (Q^2_{2i}, \ldots, Q^r_{2i})]]).$$

This completes the induction and the proof of the lemma.

□

Lemma 3.5: For any two distinct vertices u and v of Q^X_n, if there exists a positive even $m < dim(S(u,v))$ such that

(1) $type(S_m(u)) = X$ and $type(S_m(v)) = C$, u is in $S_{m,xy}(u)$ and v is in $S_{m,\bar{x}\bar{y}}(v)$; or

(2) $type(S_m(u)) = C$ and $type(S_m(v)) = X$, u is in $S_{m,xy}(u)$ and v is in

$S_{m,\overline{xy}}(v)$,

let m^* be the smallest such m. If (1) holds for m^*, then there exists a shortest path $P^* = (u(S_{m^*,xy}(u)), u(S_{m^*,\overline{xy}}(u))) \mid P^*[u(S_{m^*,\overline{xy}}(u))), v; S(u,v)]$. If (2) holds for m^*, then there exists a shortest path $P^* = (u(S_{m^*,xy}(u)), u(S_{m^*,\overline{xy}}(u))) \mid P^*[u(S_{m^*,\overline{xy}}(u))), v; S(u,v)]$.

Proof: Let $P_{m^*} = (S^1_{m^*}, S^2_{m^*}, ..., S^r_{m^*})$, where $S^1_{m^*} = S_{m^*}(u)$ and $S^r_{m^*} = S_{m^*}(v)$, be any shortest m^*-cube path from $S_{m^*}(u)$ to $S_{m^*}(v)$ in $S(u,v)$. If (1) holds for m^*, then $(S^1_{m^*,xy}, S^1_{m^*,\overline{xy}}, S^2_{m^*,\overline{xy}}, ..., S^r_{m^*,\overline{xy}}$ is a shortest $(m-2)$-cube path from $S_{m^*-2}(u)$ to $S_{m^*-2}(v)$ in $S(u,v)$. Since (1) and (2) do not hold for m smaller than m^*, by lemma 3.3 we know that

$$(u(S^1_{m^*,xy}), u(S^1_{m^*,\overline{xy}})) \mid P^*[u(S^1_{m^*,\overline{xy}}), v(S^r_{m^*,\overline{xy}}); (S^2_{m^*,\overline{xy}}, ..., S^r_{m^*,\overline{xy}})]$$

is a shortest path from u to v. Similar arguments can be applied to the case that (2) holds for m^*.

□

Lemma 3.6: Let $dim(S(u,v)) = k$. For any two distinct vertices u and v of Q^X_n, let $dim(S(u,v)) = k$. If none of conditions (1) and (2) given in lemma 3.4 holds for any even m such that $2 \le m \le k$, then if k is odd, $(u, u(S_{k-1}(v)) \mid P^*[u(S_{k-1}(v)), v; S_{k-1}(v)]$ is a shortest path from u to v in Q^X_n, and if k is even, $(u, u(S'_{k-2})) \mid P^*[u(S'_{k-2}), v); S_{k-1}(v)]$ is a shortest path from u to v in Q^X_n, where S'_{k-2} is the $(k-2)$-dimensional subcube of $S(u,v)$ that is adjacent to $S_{k-2}(u)$.

Proof: This lemma directly follows from lemma 3.3. The case that k is odd

corresponds to the special case of $r = 2$ in lemma 3.3, and the case that k is even corresponds to the special cases of $r = 2$ or 3 in lemma 3.3, depending on the length of shortest $(k-2)$-cube path from $S_{k-2}(u)$ to $S_{k-2}(v)$ in $S(u,v)$.

□

Combining lemma 3.6 and definition 2.2, we have the following shortest path algorithm.

```
Algorithm PATH_2(s,d,n)
    P* := ∅;
    while s ≠ d do
        compute the prefix states of s and d;
        let l and r be the index of the leftmost and rightmost distinct bit of s
        and d, respectively;
        flag := false;
        if r is odd then m* := r+1 else m* := r;
        while m* < l - 1 and flag = false do
            if prefix(n,s;m*) ≠ prefix(n,d;m*) and s_{m*} s_{m*-1} ≠ d_{m*} d_{m*-1}
            and s_{m*} ≠ d_{m*}
            then begin
                P* := (s, s_n ... s_{m*+1} d_{m*} d_{m*-1} s_{m*-2} ... s_1) | P*;
                s := s_n ... s_{m*+1} d_{m*} d_{m*-1} s_{m*-2} ... s_1;
                flag := true;
            end
            else m* := m* + 2;
        endwhile
        if flag = false then
            begin
                P* := (s, s_n ... s_{l+1} d_l s_{l-1} ... s_1) | P*;
                s := s_n ... s_{l+1} d_l s_{l-1} ... s_1;
            end
    endwhile
end PATH_2
```

Theorem 3.3: Algorithm *PATH_2* computes a shortest path from any vertex s to any vertex d in an n-dimensional X-hypercube Q_n^X in linear order in $O(n^2)$ time.

Proof: The correctness of algorithm *PATH_2* can be easily derived from lemma 3.4,

lemma 3.5, and the new definition of a X-hypercube. The time complexity of *PATH_2* can be verified as follows. Using finite-state automata A_n, prefix(n,s;m^*) and prefix(n,d;m^*) can be computed in $Q(n)$ time. Let $P^* = (s = u_1, u_2, ..., u_r = d)$ be the path computed by the algorithm. Clearly, u_1 is vertex s and u_r is vertex d. Since u_i is computed from u_{i-1} and the diameter of Q_n^X is $\lceil (n+1)/2 \rceil$, the *while*-loop iterates at most $\lceil (n+1)/2 \rceil$ times. In each iteration of the *while*-loop, $O(n)$ time is sufficient for computing the next vertex in the path. Therefore, the total time of algorithm *PATH_2* is $O(n^2)$.

□

3.3. Summary

We presented two shortest path algorithms, *PATH_1* and *PATH_2*, for X-hypercube Q_n^X. *PATH_1* computes a shortest path in non-linear order in $O(n)$ time. This algorithm can be used for centralized control of circuit switching, which establishes the connection between a source processor and a destination processor before actual data communication starts. For packet switching networks, each processor can have a copy of *PATH_2*, which can be used to find the next processor on the shortest path in $O(n)$ time, so that the total data communication time between two processors is $O(n^2)$. Both algorithms can be implemented in constant space. It remains as an open problem whether or not a shortest path between any two vertices in Q_n^T can be computed in $O(n)$ time and constant space.

Chapter 4

Average Distance between Vertices in a X-Hypercube

4.1. Introduction

The average distance between vertices of a network is an important factor in evaluating the data communication performance of the network. While the diameter of a network reflects the worst case data communication performance, the average distance between vertices characterizes the average case performance. In this chapter, we show that the average distance between vertices in Q_n^X is about 13/16 of the average distance between vertices in Q_n. This indicates that the data communication performance of X-hypercube systems is better than that of conventional hypercube in both the worst and average cases. It is known that the twisted cube, the X-hypercube and the multiply-twisted cube are isomorphic for $n \leq 3$. Comparing our results with the average vertex distance of the multiply-twisted cube and the twisted cube given in [YZ91] and [AP89] respectively, we prove that they are not isomorphic to each other for $n > 3$.

4.2. Average Distance

The distance between two vertices u and v in a connected undirected graph is the length of the shortest path from a to b. For two vertices u and v in a subgraph G of Q_n^X, let $D(u, v, G)$ denote the distance between u and v in G. The average distance between vertices in Q_n^X is equal to the sum of $D(u, v, Q_n^X)$ over all pairs of distinct vertices divided by the number of such pairs. Given any vertex u in Q_n^X, we can

assign each vertex v in Q_n^X a new n-bit binary label v' such that the new labels of all vertices satisfy the definition of Q_n^X and the new label u' of u is 00..0 (see [KZ91-3]). Therefore, the average distance between vertices in Q_n^X is equal to the sum of $D(00..0, b, Q_n^X)$ over all n-bit binary strings b such that $b \neq 00..0$ divided by the number of such b's. Let $T(n) = \sum_{b \neq 00...0} D(00...0, b, Q_n^X)$.

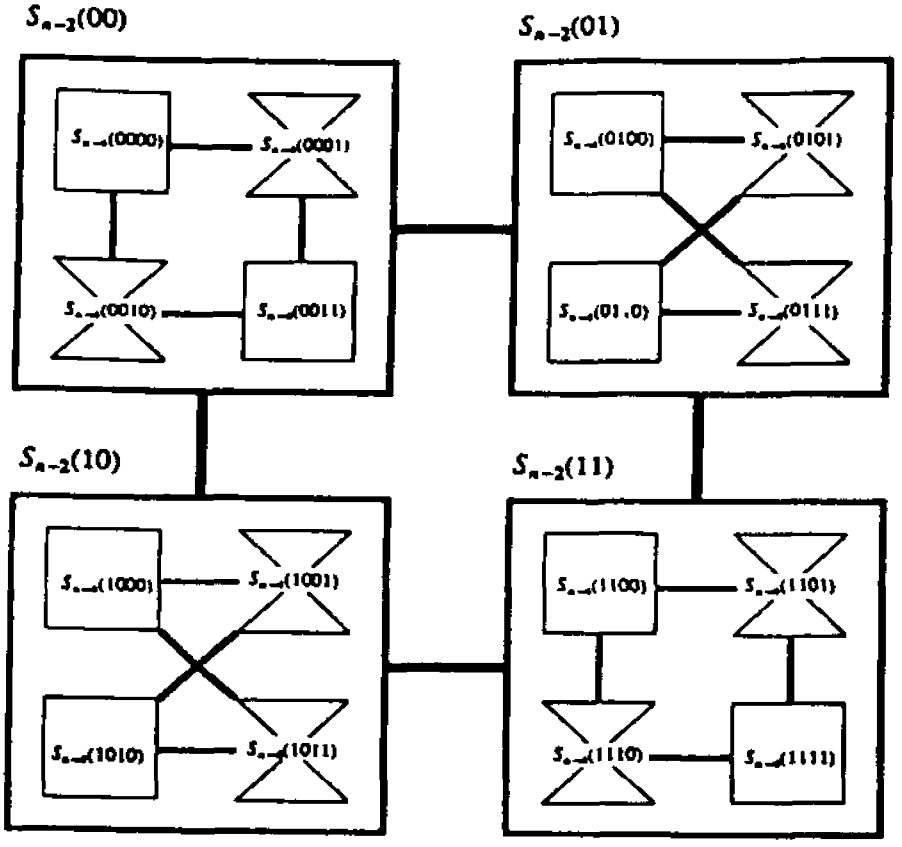


Figure 9: Partition of Q_n^X for even n

Lemma 4.1: For all even n greater than 2, $T(n) = 2^n + 13 * (n-2) * 2^{(n-5)}$; for all odd n greater than 3, $T(n) = 11 * 2^{(n-3)} + 13 * (n-3) * 2^{(n-5)}$.

Proof: First, let us evaluate $T(n)$ for all even n's. If n is greater than 4, then Q_n^X can be partitioned into four $(n-2)$-dimensional subcubes $S_{n-2}(00)$, $S_{n-2}(01)$, $S_{n-2}(10)$, and $S_{n-2}(11)$, where $S_{n-2}(ij)$ is the subgraph of Q_n^X induced by the vertices $b_n b_{n-1} \cdots b_1$ such that $b_n b_{n-1} = ij$. Similarly, each of these subcubes can be partitioned into four $(n-4)$-dimensional subcubes. For example, $S_{n-2}(00)$ is partitioned into $S_{n-4}(0000)$, $S_{n-4}(0001)$, $S_{n-4}(0010)$, and $S_{n-4}(0011)$. These partitions are shown in figure 9. Clearly,

$$T(n) = \sum_{b \in S_{n-2}(00)} D(00...0,b,Q_n^X) + \sum_{b \in S_{n-2}(01)} D(00...0,b,Q_n^X) +$$

$$\sum_{b \in S_{n-2}(10)} D(00...0,b,Q_n^X) + \sum_{b \in S_{n-2}(11)} D(00...0,b,Q_n^X).$$

This equation has four terms. We analyze these terms separately. First, it is obvious that

$$\sum_{b \in S_{n-2}(00)} D(00...0,b,Q_n^X) = \sum_{b \in Q_{n-2}^X} D(00...0,b,Q_{n-2}^X) = T(n-2).$$

For the second term, we have

$$\sum_{b \in S_{n-2}(01)} D(00...0,b,Q_n^X) = \sum_{b \in S_{n-4}(0100)} D(00...0,b,Q_n^X) +$$

$$\sum_{b \in S_{n-4}(0101)} D(00...0,b,Q_n^X) + \sum_{b \in S_{n-4}(0110)} D(00...0,b,Q_n^X) +$$

$$\sum_{b \in S_{n-4}(0111)} D(00...0,b,Q_n^X).$$

By observation, we identify the following shortest $(n-4)$-cube paths from $S_{n-4}(0000)$

to

$S_{n-4}(0100)$, $S_{n-4}(0101)$, $S_{n-4}(0110)$, and $S_{n-4}(0111)$, respectively:

$$P_{n-4}^{0} = (S_{n-4}(0000), S_{n-4}(0100)),$$

$$P_{n-4}^{1} = (S_{n-4}(0000), S_{n-4}(0100), S_{n-4}(0101)),$$

$$P_{n-4}^{2} = (S_{n-4}(0000), S_{n-4}(0010), S_{n-4}(0110)), \text{ and}$$

$$P_{n-4}^{3} = (S_{n-4}(0000), S_{n-4}(0100), S_{n-4}(0111)).$$

Furthermore, it is easy to see that

$$type\,(P_{n-4}^{0}) = XX,$$

$$type\,(P_{n-4}^{1}) = XXC,$$

$$type\,(P_{n-4}^{2}) = XXC, \text{ and}$$

$$type\,(P_{n-4}^{3}) = XCX.$$

By lemma 3.3 and lemma 3.6, we know that for any vertex b in $S_{n-4}(0100)$, there exists a shortest path from vertex 00...0 to b such that

$$P^{*}[00...0, b; Q_n^X] = P^{*}[00...0, b; P_{n-4}^{0}]$$

$$= (00...0, 010...0) \mid P^{*}[010...0, b; S_{n-4}(0100)].$$

Thus,

$$\sum_{b \in S_{n-4}(0100)} D(000...0, b, Q_n^X) = 2^{n-4} + \sum_{b \in S_{n-4}(0100)} D(0100...0, b, S_{n-2}(01)).$$

By lemma 3.3 and lemma 3.6 again, we know that for any vertex b in $S_{n-4}(0101)$, there exists a shortest path from vertex 00...0 to b such that

$$P^*[00...0,b;Q_n^X] = P^*[00...0,b;P_{n-4}^1]$$

$$= (00...0, 010...0) \mid P^*[010...0,b;(S_{n-4}(0100),S_{n-4}(0101))]$$

$$= (00...0, 010...0) \mid P^*[010...0,b;S_{n-2}(01)].$$

Therefore,

$$\sum_{b \in S_{n-4}(0101)} D(000...0,b,Q_n^X) = 2^{n-4} + \sum_{b \in S_{n-4}(0101)} D(0100...0,b,S_{n-2}(01)).$$

Similarly, considering shortest paths from 00...0 to all vertices b in $S_{n-4}(11)$ in subgraph P_{n-4}^3,

we have

$$\sum_{b \in S_{n-4}(0111)} D(000...0,b,Q_n^X) = 2^{n-4} + \sum_{b \in S_{n-4}(0111)} D(0100...0,b,S_{n-2}(01)).$$

Comparing P_{n-4}^2 with the $(n-4)$-cube path $S_{n-4}(0100),S_{n-4}(0111),S_{n-4}(0110))$, which is the shortest $(n-4)$-cube path from $(S_{n-4}(0100)$ to $S_{n-4}(0110)$ in $S_{n-2}(01)$, we know that

$$\sum_{b \in S_{n-4}(0110)} D(000...0,b,Q_n^X) = \sum_{b \in S_{n-4}(0110)} D(0100...0,b,S_{n-2}(01)).$$

Then,

$$\sum_{b \in S_{n-2}(01)} D(00...0,b,Q_n^X) = 3*2^{n-4} + \sum_{b \in S_{n-4}(0100)} D(010...0,b,S_{n-2}(01)) +$$

$$\sum_{b \in S_{n-4}(0101)} D(010...0,b,S_{n-2}(01)) +$$

$$\sum_{b \in S_{n-4}(0110)} D(010...0,b,S_{n-2}(01)) +$$

$$\sum_{b \in S_{n-4}(0111)} D(010...0,b,S_{n-2}(01)).$$

Since

$$\sum_{\substack{b \in S_{n-2}(01ij) \\ i,j \in \{0,1\}}} D(010...0,b,S_{n-2}(01)) = \sum_{b \in Q_{n-2}^X} D(00...0,b,Q_{n-2}^X),$$

we have

$$\sum_{b \in S_{n-2}(01)} D(00...0,b,Q_n^X) = 3*2^{n-4} + T(n-2).$$

The analysis of the third term is similar to that for the second term, since $S_{n-2}(10)$ and $S_{n-2}(01)$ are symmetric with respect to vertex 00...0. Hence,

$$\sum_{b \in S_{n-2}(10)} D(00...0,b,Q_n^X) = 3*2^{n-4} + T(n-2).$$

Now, consider the last term $\sum_{b \in S_{n-2}(11)} D(00...0,b,Q_n^X)$. Clearly,

$$\sum_{b \in S_{n-2}(11)} D(00...0,b,Q_n^X) = \sum_{b \in S_{n-4}(1100)} D(00...0,b,Q_n^X) +$$

$$\sum_{b \in S_{n-4}(1101)} D(00...0,b,Q_n^X) + \sum_{b \in S_{n-4}(1110)} D(00...0,b,Q_n^X) +$$

$$\sum_{b \in S_{n-4}(1111)} D(00...0,b,Q_n^X).$$

By observation, we identify the following shortest $(n-4)$-cube paths from $S_{n-4}(0000)$ to $S_{n-4}(1100)$, $S_{n-4}(1101)$, $S_{n-4}(1110)$, and $S_{n-4}(1111)$, respectively:

$$P_{n-4}^{00} = (S_{n-4}(0000),S_{n-4}(0100),S_{n-4}(1100)),$$

$$P_{n-4}^{01} = (S_{n-4}(0000),S_{n-4}(0100),S_{n-4}(1100),S_{n-4}(1101)),$$

$$P_{n-4}^{10} = (S_{n-4}(0000),S_{n-4}(0100),S_{n-4}(1100),S_{n-4}(1110)), \text{ and}$$

$$P_{n-4}^{11} = (S_{n-4}(0000),S_{n-4}(0100),S_{n-4}(0111),S_{n-4}(1111)).$$

Furthermore,

$$type\,(P_{n-4}^{00}) = XXX\,,$$

$$type\,(P_{n-4}^{01}) = XXXC\,,$$

$$type\,(P_{n-4}^{10}) = XXXC\,, \text{ and}$$

$$type\,(P_{n-4}^{11}) = XXCX\,.$$

By lemma 3.3 and lemma 3.6, and using arguments similar to that for the second term, we know that

$$\sum_{b \in S_{n-4}(1100)} D\,(000...0,b\,,Q_n^X) = 2*2^{n-4} + \sum_{b \in S_{n-4}(1100)} D\,(1100...0,b\,,S_{n-2}(11)),$$

$$\sum_{b \in S_{n-4}(1101)} D\,(000...0,b\,,Q_n^X) = 2*2^{n-4} + \sum_{b \in S_{n-4}(1101)} D\,(1100...0,b\,,S_{n-2}(11)),$$

$$\sum_{b \in S_{n-4}(1110)} D\,(000...0,b\,,Q_n^X) = 2*2^{n-4} + \sum_{b \in S_{n-4}(1110)} D\,(1100...0,b\,,S_{n-2}(11)),$$

and

$$\sum_{b \in S_{n-4}(1111)} D\,(000...0,b\,,Q_n^X) = 2^{n-4} + \sum_{b \in S_{n-4}(1111)} D\,(1100...0,b\,,S_{n-2}(11)).$$

Therefore,

$$\begin{aligned}
\sum_{b \in S_{n-2}(11)} D\,(00...0,b\,,Q_n^X) &= 7*2^{n-4} + \sum_{b \in S_{n-2}(11)} D\,(110...0,b\,,S_{n-2}(11)) \\
&= 7*2^{n-4} + \sum_{b \in S_{n-2}(11)} D\,(110...0,b\,,S_{n-2}(11)) \\
&= 7*2^{n-4} + \sum_{b \in Q_{n-2}^X} D\,(00...0,b\,,Q_{n-2}^X) \\
&= 7*2^{n-4} + T\,(n-2).
\end{aligned}$$

Summing up all four terms, we have

$$T(n) = 4*T(n-2)+13*2^{n-4}.$$

We can solve this recurrence relation by replacement using the initial condition, T(2)=4, as follows.

$$T(n) = 4*T(n-2)+13*2^{n-4}.$$

$$= 4*(4*T(n-4) + 13*2^{n-6}) + 13*2^{n-4}$$

$$= 4^2*T(n-4) + 4*13*2^{n-6} + 13*2^{n-4}$$

$$= 4^3*T(n-6) + 4^2*13*2^{n-8} + 4*13*2^{n-6} + 13*2^{n-4}$$

$$= 4^4*T(n-8) + 4^3*13*2^{n-10} + 4^2*13*2^{n-8} + 4*13*2^{n-6}$$

$$+ 13*2^{n-4}$$

.

.

.

$$= 4^{(n-2)/2}*T(2) + \sum_{i=0}^{n/2-2} 4^i*13*2^{n-2i-4}$$

$$= 4^{n/2} + \sum_{i=0}^{n/2-2} 4^i*13*2^{n-2i-4}$$

$$= 2^n + \sum_{i=0}^{n/2-2} 2^{2i}*13*2^{n-2i-4}$$

$$= 2^n + 13*\sum_{i=0}^{n/2-2} 2^{n-4}$$

$$= 2^n + 13*(n/2 - 1)*(2^{n-4})$$

$$= 2^n + 13*(n-2)*(2^{n-5})$$

Therefore, we obtain following closed form of the above recurrence relation for even n and $n > 2$.

$$T(n) = 2^n + 13 * (n-2) * 2^{(n-5)}$$

Using lemma 3.3, lemma 3.6 and similar analysis, we obtain the following recurrence relation for odd n and $n > 3$.

$$T(n) = 2*T(n-1)+3*2^{n-3}.$$

We can solve this recurrence relation by replacement, using the initial condition $T(2) = 4$ as follows:

$$\begin{aligned}
T(n) &= 2*T(n-1)+3*2^{n-3}. \\
&= 2*(4*T(n-3) + 13*2^{n-5}) + 3*2^{n-3} \\
&= 2^3*T(n-3) + 13*2^{n-4} + 3*2^{n-3} \\
&= 2^5*T(n-5) + 13*2^{n-4} + 13*2^{n-4} + 3*2^{n-3} \\
&= 2^7*T(n-7) + 13*2^{n-4} + 13*2^{n-4} + 13*2^{n-4} + 3*2^{n-3} \\
&\quad . \\
&\quad . \\
&\quad . \\
&= 2^{n-2}*T(2) + \sum_{i=1}^{(n-1)/2-1} 13*2^{n-4} + 3*2^{n-3} \\
&= 2^n + \sum_{i=1}^{(n-1)/2-1} 13*2^{n-4} + 3*2^{n-3} \\
&= 2^n + ((n-1)/2 - 1)*13*2^{n-4} + 3*2^{n-3} \\
&= 2^3 2^{n-3} + 3*2^{n-3} + 13*(n-3)*2^{n-5} \\
&= 11*2^{n-3} + 13*(n-3)*2^{n-5}
\end{aligned}$$

Therefore, we obtain following closed form of the above recurrence relation for odd n

and $n > 3$.

$$T(n) = 11 * 2^{(n-3)} + 13 * (n-3) * 2^{(n-5)}$$

This completes the proof.

□

Let $D_{ave}(n)$ denote the average distance between vertices in Q_n^X. Since $D_{ave}(n) = T(n) / (2^n - 1)$, we have following conclusion:

Theorem 4.1: For a n-dimensional X-hypercube Q_n^X, if n is even, then

$$D_{ave}(n) = 13(n-2)/32 + (13n + 6)/(32(2^n - 1)) + 1;$$

if n is odd, then

$$D_{ave}(n) = 13(n-3)/32 + (13n + 5)/(32(2^n - 1)) + 11/8.$$

□

4.3. Summary

Our analysis shows that the average distance between vertices in a X-hypercube Q_n^X is about $(13/32)n$, which is 13/16 of the average distance between vertices in a conventional hypercube Q_n. Even though the reduction in average intervertex distance is not as striking as the reduction in the diameter, this indicates that the average data communication performance of a X-hypercube network Q_n^X is better than that of a conventional hypercube Q_n. In [YZ91], it is shown that the average distance between vertices in a n-dimensional multiply-twisted cube, denoted by Q_n^{MT}, is about $(3/8)n$. According to the investigation in [AP89], the average distance between vertices in n-

dimensional twisted cube is about $(6/8)n$. Our result constitutes a proof that all of these three modified hypercubes are not isomorphic to each other.

Chapter 5

SIMD Data Communication Algorithms for a X-Hypercube

5.1. Introduction

In most problems that we wish to solve with multicomputer systems, it is required for processors to be able to communicate among themselves during execution in order to exchange data or intermediate results. The fundamental issues for data communication are considered to be a routing (one-to-one or many-to-many), broadcasting (one-to-many) and census function (many-to-one). In this chapter, we explore the effectiveness of a X-hypercube network for parallel computing by considering the one-to-one, one-to-many and many-to-one interprocessor communication problems. We present *SIMD* parallel data broadcasting and census algorithms for a X-hypercube network. In fact, the major drawback of the X-hypercube is the inherent difficulties in the routing scheme due to its complicated topology. An efficient SIMD one-to-one routing algorithm is also presented. The many-to-many routing problem will be investigated in the next chapter.

Our results indicate that X-hypercube machines are good alternatives for hypercube machines, considering that (1) the X-hypercube Q_n^X has the same structural complexity of a conventional hypercube Q_n, and consequently can be constructed with the same hardware cost; (2) the data communication cost, in general, predominates the computation cost for parallel algorithms running on a multicomputer system. That is, with little additional hardware cost, X-hypercube machines can be possibly twice as

effective as conventional hypercube machines, when communication cost in parallel computing is of major concern.

5.2. Data Broadcasting Algorithms

Broadcasting is to transfer data from one processor to all other processors in a multicomputer system. One of the simplest broadcasting algorithms is *flooding*, in which every incoming data is sent out on every outgoing link except the one it arrived on. [Tan81] While *flooding* always chooses the shortest path because it selects every possible path in parallel, it obviously generates a lot of duplicate data, i.e., a processor may receive the same message more than once. In order to avoid this redundancy, an adaptive broadcasting algorithm to base routing decision on the topology of the X-hypercube can be considered.

In this section, we first present a parallel flooding algorithm for broadcasting from P_0 to all other processors in a X-hypercube machine Q_n^X. In fact, it is not trivial to select the links that the data arrived on without hardware support. Even though those links are found and incoming data is not sent through them, *flooding* causes processors in Q_n^X to receive a redundant data. Therefore, to simplify the algorithm, our parallel flooding algorithm, *FLOODING*, sends a incoming data through all links connected to a processor. Next, we present an adaptive broadcasting algorithm by which each processor receives the message broadcasted from P_0 exactly once. Then, we show how to use this algorithm to broadcast a message from an arbitrary processor P_b to all other processors in Q_n^X. In our algorithm, we denote a variable or register owned by processor P_b by the subscript b. For example, we assume each processor P_b has a

register A_b, which will be used to contain the message received and the message to be sent to a subset of its adjacent processors. The statement **send** A_b to $B_{b'}$, where $P_{b'}$ is an element of set S_b of P_b's adjacent processors, transmits the content of register A_b to the register $B_{b'}$ of all processors in S_b in parallel. To analyze the performance of our parallel algorithms, we use two metrics: *computation step* and *communication step*. A computation step is a logical or arithmetic operation executed in all processor in parallel. A communication step is an operation to initiate interprocessor communication. In our algorithms, the statement **send** is the only communication instruction.

Broadcasting from P_0 by *flooding* can be implemented in the following algorithm.

```
Algorithm FLOODING
    for all processors P_b do in parallel
        Compute S_b, the set of all adjacent processors of P_b;
    endfor
    for all processors P_0' do in parallel
        if b is 0 then
            send A_b to A_b' such that P_b' ∈ S_b;
    endfor
    for j:= 1 to ⌈(n+1)/2⌉-1 do
        for all processor P_b do in parallel
            if P_b has received the message during previous step then
                send A_b to A_b' such that P_b' ∈ S_b;
        endfor
    endfor
end FLOODING
```

Theorem 5.1: Algorithm *FLOODING* broadcasts a message from P_0 to all other processors of Q_n^X in $O(n)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps. }i.ti +0

Proof: The correctness of algorithm *FLOODING* is obvious because Q_n^X is a con-

nected graph and its diameter is $\lceil (n+1)/2 \rceil$. The computation instruction is only to compute S_i. By definition 2.2.2, we can obtain S_b for each processor P_b in $O(n)$ time. Since algorithm *FLOODING* executes the communication instruction, **send**, $\lceil (n+1)/2 \rceil$ times, it takes $\lceil (n+1)/2 \rceil$ communication steps. This completes the proof.

□

Since algorithm *FLOODING* allows more than one processors to send the message to a processor at any communication step, special hardware may be needed for each processor to choose the input from one of its adjacent processors. Also, within one step, two processors may send the broadcasted messages to each other. This requires that the processors be connected by a full-duplex link. Based on algorithm *FLOODING*, we introduce another broadcasting algorithm which restricts each processor to receive the broadcasted message exactly once. Clearly, we need to find a spanning tree in the X-hypercube rooted at P_0 and send a message along the paths of the spanning tree. We use an adaptive technique by taking advantage of the topology of the X-hypercube to obtain paths equivalent to those in a spanning tree. To facilitate our presentation, we define a function $RBIT(b)$ as the index of the rightmost 1 of a n-bit binary string b. We assume that $RBIT(b)$ is $n+1$ when $b_n b_{n-1} \ldots b_1$ is 00...0. For example, *RBIT(00010100)* returns 3 and *RBIT* (00000000) returns 9.

Algorithm *ADAPBROAD*
(* computation part *)
for all processor P_b **do in parallel**
Compute $S_b := \{P_{b'} \mid P_{b'}$ is adjacent to P_b and $RBIT(b) > RBIT(b')\}$;
for k := 2 **to** n **by** 2 **do**
if the type of *prefix(n,b;k)* is X_e **then**
case $b_k b_{k-1}$ **of**

```
                        10 : S_b := S_b ∪ {P_b' | b'_{k-1}=1 and b'_i = b_i for
                             i ≠ k-1};
                             if b_{k+2}b_{k+1} is 00 then
                             S_b := S_b ∪ { P_b' | b'_{k+1}=1, or b'_{k+2}=1 and
                                    b'_i = b_i for i ≠ k+1 or i ≠ k+2
                                    respectively};
                        11 : S_b := ∅;
                        otherwise : do nothing;
                    endcase
                else if the type of prefix(n,b;k) is C_e then
                    case b_k b_{k-1} of
                        10 : S_b = S_b - { P_b' | b'_{k-1}=1 };
                        otherwise : do nothing;
                    endcase
                endif
            endfor
        endfor
        (* communication part *)
        send A_0 to A_0' such that P_0' ∈ S_0;
        for i = 1 to ⌈(n+1)/2⌉ - 1 do
            for all processor P_b do in parallel
                if P_b received message during the previous step then
                    send A_b to A_b' such that P_b' ∈ S_b;
            endfor
        endfor
    end ADAPBROAD
```

Theorem 5.2: Algorithm *ADAPBROAD* broadcasts a message from P_0 to all other processors of Q_n^X in $O(n)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps; furthermore, each processor P_i receives and sends the broadcasted message at most once.

Proof: The proof for the correctness of algorithm *ADAPBROAD* can be divided into two parts: (1) P_0 broadcasts message to all other processors, and each processor P_b, such that $b \neq 0$, receives message exactly once, (2) each processor P_b sends message at most once. Algorithm *ADAPBROAD* also consists of computation part for computing the subset of P_b's adjacent processors, and a communication part for sending the

broadcasted message to P_b's adjacent processors.

Part (1):

The computation part has two major steps. First, each processor P_b computes a subset of its adjacent processors, $S_b = \{ P_{b'} \mid P_{b'}$ is adjacent to P_b and $RBIT(b) > RBIT(b') \}$. Second, the **for**-loop rearranges the S_b's so that a spanning tree of Q_n^X can be obtained. To prove (1), we only need to show that every processor $P_{b'}$ except P_0 is an element of a unique S_b computed by algorithm ADAPBROAD. Consider the binary label of $P_{b'}$, $b'_n\ b'_{n-1}...b'_1$ such that $RBIT(b') = r$. In order for $P_{b'}$ to be included in S_b in the first computation step, the binary string b must satisfy one of two conditions in the definition 2.2, and $RBIT(b) > RBIT(b')$. By definition 2.2, we need to check at most two bits to determine the adjacent processors. We have the following four cases.

Case 1: The type of $prefix(n,b';r)$ is C_e for even r.

There must be some P_b which are adjacent to $P_{b'}$, such that

$(b_r b_{r-1}, b'_r b'_{r-1}) \in \{ (01,10),(11,10) \}$ and $b_i = b'_i$ for $i \neq r$ and $i \neq r-1$.

Because $RBIT(b) < RBIT(b')$, $P_{b'}$ in Case 1 can not be an element of S_b. However, the **for**-loop in the computation part forces the $P_{b'}$ to be an element of a unique set S_b. By observing the automata A_n in figure 7, we see that $prefix(n,b;r)$ is of type X_e for $b_{r+2}b_{r+1} = 00$. Since the **for**-loop in the computation part makes $P_{b'}$ to be an element of a set S_b when *prefix(n,b;r)* is of type X_e and $b_{r+2}b_{r+1} = 00$, even though $RBIT(b) = RBIT(b')$, $P_{b'}$ can be an element of a unique set S_b for $b_{r+1} \neq b'_{r+1}$ or $b_{r+2} \neq b'_{r+2}$ and $b_i = b'_i$ for i≠r+1 or i≠r+2,

respectively.

Case 2: The type of *prefix* $(n, b'; r+1)$ is C_e for odd r.

There must be some P_b which are adjacent to $P_{b'}$, such that

$(b_{r+1}b_r, b'_{r+1}b'_r) \in \{ (00,01),(10,01), (00,11),(10,11) \}$ and $b_i = b'_i$ for $i \neq r$ and $i \neq r-1$.

Because $RBIT(b) > RBIT(b')$, $P_{b'}$ in Case 2 can be an element of more than one S_b. However, the **for**-loop in the computation part deletes $P_{b'}$ from S_b if $b_{r+1}b_r = 10$ and *prefix(n,b,r+1)* is of type C_e, $P_{b'}$ is an element of a unique S_b for $b_{r+1}b_r = 00$ and $b_i = b'_i$ for $i \neq r$.

Case 3: The type of *prefix* $(n, b'; r)$ is not C_e for even r.

There must be some adjacent P_b, which are adjacent to $P_{b'}$, such that

$(b_r b_{r-1}, b'_r b'_{r-1}) \in \{ (00,10),(11,10) \}$ and $b_i = b'_i$ for $i \neq r$ and $i \neq r-1$.

If $(b_k b_{k-1}, b'_k b'_{k-1}) = (11,10)$, $P_{b'}$ can not be an element of S_b because $RBIT(b) < RBIT(b')$. Therefore, $P_{b'}$ in case 3 is an element of a unique S_b.

Case 4: The type of *prefix* $(n, b'; r)$ is not C_e for odd r.

There must be some adjacent P_b, which are adjacent to $P_{b'}$, such that

$(b_{r+1}b_r, b'_{r+1}b'_r) \in \{ (00,01),(11,01), (01,11),(10,11) \}$ for $b_i = b'_i$ for $i \neq r+1$ and $i \neq r$.

If $(b_{r+1}b_r, b'_{r+1}b'_r) \in \{ (11,01),(01,11) \}$, $P_{b'}$ can not be an element of S_b because $RBIT(b) = RBIT(b')$. If $(b_{r+1}b_r, b'_{r+1}b'_r) = (10,11)$, $P_{b'}$ can not be an element of P_b by the **for**-loop. Instead, the loop makes $P_{b'}$ an element of S_b such

that $b_{r+2}=1$ and $b'_i=b_i$ for $i \neq r+2$. Therefore, $P_{b'}$ in case 4 is also an element of a unique S_b.

This completes the proof for part (1).

Part (2):

In the communication part, each processor P_b sends the broadcasted message only if it receives the message at a previous step. Therefore, it is obvious that part (2) is true.

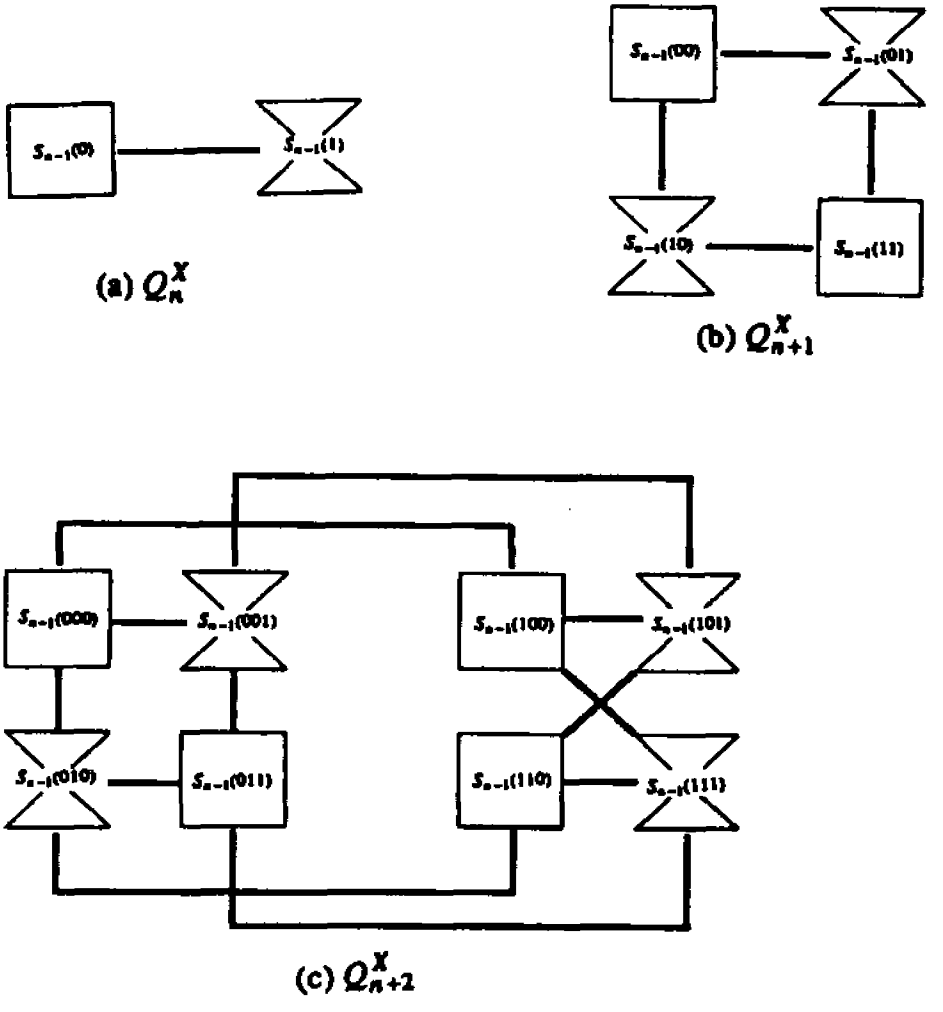


Figure 10: Q_n^X, Q_{n+1}^X and Q_{n+2}^X for odd n

Now, we consider the complexity. The computation involved is only to compute S_b. By definition 2.2, we can obtain S_b for each processor P_b in $O(n)$ time by scanning its binary label from right to left. The S_b includes all its adjacent processors $P_{b'}$ which are not an element of any $S_{b''}$ for $b \neq b'$ and $b \neq b''$ and $b' \neq b''$.

The communication complexity can be proved by induction on dimension n of a X-hypercube. It is trivial to verify that algorithm *ADAPBROAD* broadcasts a message from P_0 to any other processor $P_{b'}$ in $\lceil (n+1)/2 \rceil$ communication steps by figure 11(a) for $n = 3$. Assume that it is true for odd dimension n in Q_n^X.

Consider the even dimension of $n+1$ in figure 10(b). We have following three cases.

Case 1.1 : The processors $P_{b'}$'s are in $S_{n-1}(00)$ or $S_{n-1}(01)$.

All broadcasting paths by *ADAPBROAD* are in the subgraph $P_{n-1} = (S_{n-1}(00), S_{n-1}(01))$ induced by vertices of $S_{n-1}(00)$ and $S_{n-1}(01)$. By definition 2.1, we know that $(S_{n-1}(00), S_{n-1}(01))$ is isomorphic to Q_n^X by removing its nth bit from each vertex label. By inductive assumption, a message can be broadcasted from P_0 to any other processor $P_{b'}$ in $\lceil (n+1)/2 \rceil$ communication steps.

Case 1.2 : The processors $P_{b'}$'s are in $S_{n-1}(10)$.

All broadcasting paths by *ADAPBROAD* are in the subgraph $P_{n-1} = (S_{n-1}(00), S_{n-1}(10))$ induced by vertices of $S_{n-1}(00)$ and $S_{n-1}(10)$. By definition 2.1, we know that $(S_{n-1}(00), S_{n-1}(10))$ is isomorphic to Q_n^X by removing its $(n+1)$th bit from each vertex label. By inductive assumption, a message can be broadcasted from P_0 to any

other processor $P_{b'}$ in $\lceil (n+1)/2 \rceil$ communication steps.

Case 1.3 : The processors $P_{b'}$'s are in $S_{n-1}(11)$.

Algorithm *ADAPBROAD* lets P_0 broadcast a message first to all processors in $S_{n-1}(10)$ and then they send a message directly to their corresponding processors in $S_{n-1}(11)$. Since $(S_{n-1}(00), S_{n-1}(10))$ is isomorphic to Q_n^X, a message can be broadcasted from P_0 to any other processor $P_{b'}$ in $\lceil (n+1)/2 \rceil + 1$ communication steps.

Therefore, we can claim that a message can be broadcasted in $\lceil (n+1)/2 \rceil + 1$ communication steps. This completes the proof for even dimension.

Now consider an odd dimension of $n+2$ in figure 10(c). We have following four cases.

Case 2.1 : The processors $P_{b'}$'s are in $S_{n-1}(000)$ or $S_{n-1}(001)$ or $S_{n-1}(010)$ or $S_{n-1}(011)$.

All broadcasting paths by *ADAPBROAD* are in the subgraph $P_{n-1} = (S_{n-1}(000), S_{n-1}(001), S_{n-1}(010), S_{n-1}(011))$ induced by vertices of $S_{n-1}(000)$ and $S_{n-1}(001)$ and $S_{n-1}(010)$ and $S_{n-1}(011)$. Since $(S_{n-1}(000), S_{n-1}(001)\ S_{n-1}(010), S_{n-1}(011))$ is isomorphic to Q_{n+1}^X by removing its $(n+2)$th bit from each vertex label, we can claim that a message can be broadcasted from P_0 to any other processor $P_{b'}$ in $\lceil (n+1)/2 \rceil + 1$ communication steps.

Case 2.2 : The processors $P_{b'}$'s are in $S_{n-1}(100)$ or $S_{n-1}(101)$.

The processor P_0 first sends a message to $P_{1000..0}$ and $P_{100..0}$ broadcasts it to all processors in the subgraph $P_{n-1} = (S_{n-1}(000), S_{n-1}(101))$ induced by vertices of

$S_{n-1}(000)$ and $S_{n-1}(101)$. By definition 2.1, we know that $(S_{n-1}(100), S_{n-1}(101))$ is isomorphic to Q_n^X by removing its $(n+2)$th bit and $(n+1)$th bit from each vertex label. Therefore, a message can be broadcasted from P_0 to $P_{b'}$ in $(\lceil (n+1)/2 \rceil)+1$ communication steps.

Case 2.3 : The processors $P_{b'}$'s are in $S_{n-1}(110)$.

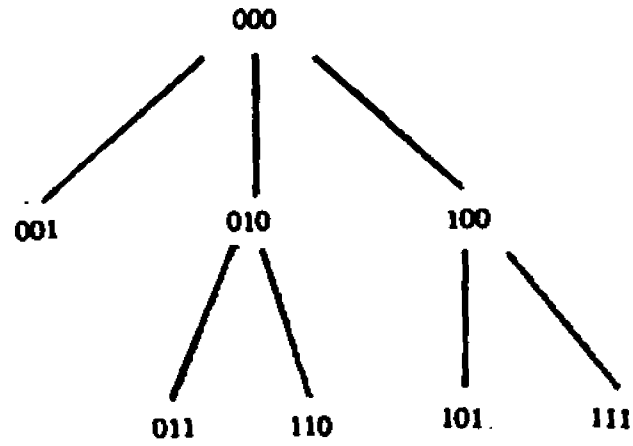
By a similar discussion to that of case 1.3, we can claim that a message can be broadcasted from P_0 to $P_{b'}$ in $(\lceil (n+1)/2 \rceil)+1$ communication steps.

Case 2.4: The processors $P_{b'}$'s are in $S_{n-1}(111)$. By the exactly same reason as in case 2.2, a message can be broadcasted from P_0 to $P_{b'}$ in $(\lceil (n+1)/2 \rceil)+1$ communication steps. This completes the proof for odd dimension.

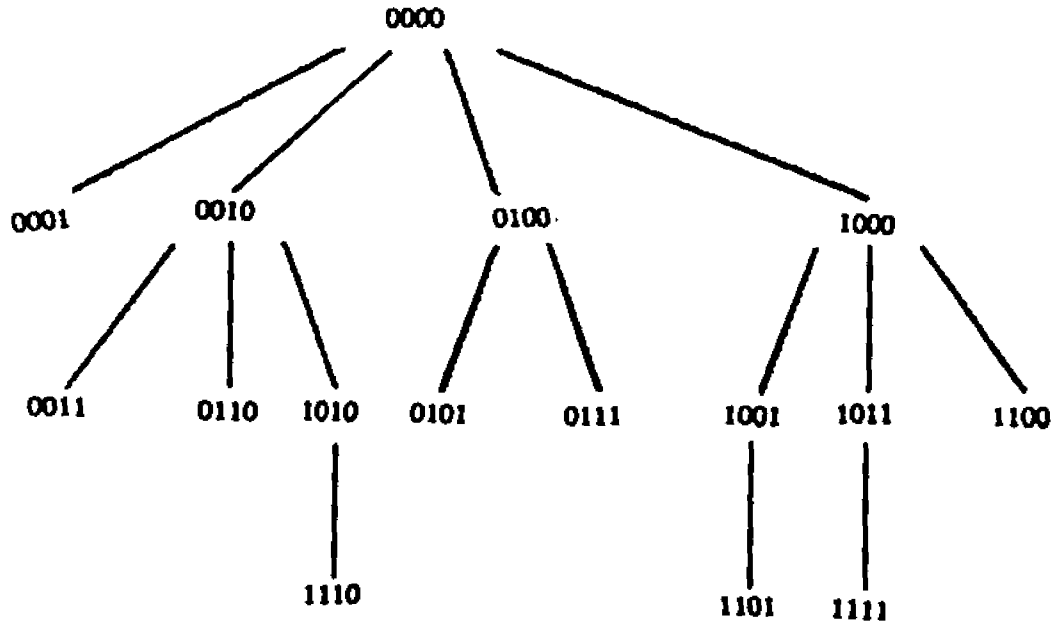
Since $(\lceil (n+1)/2 \rceil)+1$ are identical to $(\lceil (n+1+1)/2 \rceil)$ or $(\lceil (n+2+1)/2 \rceil)$ for odd n, it is true that algorithm *ADAPBROAD* broadcasts a message from P_0 to any other processor $P_{b'}$ in $(\lceil (n+1)/2 \rceil)$ communication steps in Q_n^X. This completes the proof.

□

Figure 11 shows the spanning tree of Q_3^X and Q_4^X by which algorithm *ADAPBROAD* broadcasts a message. Algorithm *ADAPBROAD* can only broadcast a message from P_o to all other processors in a X-hypercube. For efficient data communication in a multicomputer system, we may need to broadcast a message from any processor P_s to all other processors P_b.



(a) Q_3^X



(b) Q_4^X

Figure 11: Broadcasting Path

One way to achieve this goal is to provide a parallel algorithm which transforms the address of each processor P_b to a new address b' such that s'=00...0 and all new addresses b' satisfy the condition in definition 2.2. Then, algorithm *ADAPBROAD* can be used to broadcast a message from P_s using the new addresses of all processors. In what follows, we present a parallel address transformation algorithm. In this algorithm, $\oplus$ denotes the exclusive-or operation of two binary strings.

```
procedure ADDR_RELABELING(s)
    for all processors P_b do in parallel
        b' = b ⊕ s;
    endfor
    for all processors P_b do in parallel
        for k = 2 to n by 2 do
            if type of prefix(n,b;k) ≠ type of prefix (n,b';k) then
                case b'_k b'_{k-1} of
                    10 : b'_k b'_{k-1} = 11;
                    11 : b'_k b'_{k-1} = 10;
                endcase
            endif
        endfor
    endfor
end ADDR_RELABELING
```

Theorem 5.3: Procedure *ADDR_RELABELING(s)* relabels Q_n^X, where each processor P_b has address $b = b_n\ b_{n-1}...b_1$, to another Q_n^X, where a processor P_b has a new address $b' = b'_n b'_{n-1}...b'_1$ and s'=00...0. This procedure has time complexity O(n).

Proof: Since algorithm *ADDR_RELABELING(s)* applies a one-to-one mapping function, exclusive-or of s, to the label b of every processor P_b in first parallel **for**-loop and relabels $b_k b_{k-1}$ to another unique $b'_k b'_{k-1}$ for even k, s and b will be transformed to 00...0 and other unique b'. From now on, we use b' as the relabeled b by algorithm *ADDR_RELABELING(s)*. We only need to show that for every two adjacent proces-

sors P_u and P_v, $P_{u'}$ and $P_{v'}$ are also adjacent each other. In order for $P_{u'}$ and $P_{v'}$ to be adjacent each other, u' and v' must satisfy definition 2.2. Assume d is the position of first differing bit between the labels of two adjacent processors. By definition 2.2, we need to check two bits at most to determine adjacent processor. There are two cases.

Case 1: The type of *prefix(n,u;d)* is identical to the type of *prefix* $(n, u'; d)$.

In this case, only the exclusive-or function of s is applied to both label u and v. We have the following two subcases.

Subcase 1.1: The type of *prefix(n,u;d)* is C_e.

Since u and v are adjacent, $u_d u_{d-1} = \bar{v}_d \bar{v}_{d-1}$ and $u_i = v_i$ for $i \neq d$ and $i \neq d-1$. Because the exclusive-or is a one-to-one mapping function, $u'_d u'_{d-1} = \bar{v'}_d \bar{v'}_{d-1}$ and $u'_i = v'_i$ for $i \neq d$ and $i \neq d-1$. Two vertices u' and v' satisfy the condition 1 in definition 2.2. Therefore, they are adjacent to each other

Subcase 1.2: The type of *prefix(n,u;d)* is not C_e.

By definition 2.2, $u_d = \bar{v}_d$ and $u_i = v_i$ for $i \neq d$. For the same reason as in subcase 1.1, $u'_d = \bar{v'}_d$ and $u'_i = v'_i$ for $i \neq d$. Therefore, u' and v' are adjacent to each other.

Case 2: The type of *prefix(n,u;d)* is not identical to the type of *prefix* $(n, u'; d)$.

There are the following four subcases.

Subcase 2.1: The type of *prefix(n,u;d)* is C_e.

Even after applying the exclusive-or of s in the first parallel for-loop, $(u'_d u'_{d-1}, v'_d v'_{d-1}) \in \{ (00,01),(00,11),(01,00),(01,10),(10,01),(10,11),(11,00),$ $(11,10) \}$. Since *prefix* $(n, u'; d)$ is of type X_e, u' and v' are not adjacent when

$(u'_d u'_{d-1}, v'_d v'_{d-1}) \in$ { (00,11),(01,10),(10,01),(11,00) } and $u'_i = v'_i$ for $i \neq d$ and $i \neq d-1$. Since algorithm *ADDR_RELABELING(s)* changes $b'_d b'_{d-1}$ from 10 to 11 and from 11 to 10, $(u'_d u'_{d-1}, v'_d v'_{d-1}) \in$ { (00,01),(00,10),(01,00),(01,11), (11,01),(11,10),(10,00),(10,11) } and $u'_i = v'_i$ for $i \neq d$ and $i \neq d-1$. Two vertices u' and v' satisfy the condition 2 in definition 2.2. Therefore, u' and v' are adjacent each other.

Subcase 2.2: The type of *prefix(n,u;d)* is X_e.

After the first parallel **for**-loop, $(u'_d u'_{d-1}, v'_d v'_{d-1}) \in$ { (00,01),(00,10),(01,00),(01,11),(10,00),(10,11),(11,01),(11,10) }. After the second parallel **for**-loop, $(u'_d u'_{d-1}, v'_d v'_{d-1}) \in$ { (00,01),(00,11), (01,00),(01,10), (11,00),(11,10),(10,01),(10,11) }. Since *prefix(n,u';d) is* C_e, two vertices u' and v' satisfy the condition 1 in definition 2.2. Therefore, they are adjacent to each other.

Subcase 2.3: The type of *prefix(n,u;d)* is C_o.

Since k must always be odd, the second **for**-loop is not applied. Therefore $u'_d \neq v'_d$ and $u'_i = v'_i$ for $i \neq d$ when *prefix(n,u';d)* is of type T_o. Two vertices u' and v' satisfy the condition 2 in definition 2.2.

Subcase 2.4: The type of *prefix(n,u;d)* is T_o.

By a similar discussion to that of subcase 3, we also can claim that u' and v' are adjacent to each other.

We have considered all possible cases. Therefore, for any two adjacent processors P_u and P_v in Q_n^X, $P_{u'}$ and $P_{v'}$ are adjacent to each other in Q_n^X. This completes the proof.

□

By the procedure *ADDR_RELABELING* and the algorithm *ADAPBROAD*, the following generalized broadcasting algorithm can be obtained easily.

Algorithm *BROADCAST(s)*
Use procedure *ADD_RELABELING*(s) to convert in parallel the address s of P_s to $s' = 0$ and the addresses of all other P_b to b';
Use the new addresses of the processors and algorithm *ADAP-BROAD* to broadcast data from $P_{s'}$ to the rest of processors in Q_n^X
end *BROADCAST*

By theorem 5.2 and theorem 5.3, we directly obtain the following theorem.

Theorem 5.4: Algorithm BROADCAST broadcasts a message from any P_s to all other processors of Q_n^X in $O(n)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps; furthermore processor P_b receives and sends the message broadcasted at most once.

Using the algorithm *BROADCAST*, we broadcast a message from any processor to all other processors in an n-dimensional X-hypercube in O(n) computation steps and $\lceil (n+1)/2 \rceil$ communication steps. Since the diameter of a n-dimensional X-hypercube is $\lceil (n+1)/2 \rceil$, the complexity of our algorithm is optimal.

5.3. Census Algorithms

A census function is considered the opposite of broadcasting, i.e., it is to gather information about the state of network to a central location. Examples of census functions are addition, multiplication, logical and, logical or, and minimum and maximum functions. Even though the paths for the census function are the opposite as for

broadcasting, we find that it is not trivial to reverse the broadcasting paths back to obtain a census operation. In this section, we first present a simple census algorithm where the data transmission path is totally different from that of the broadcasting algorithm *ADAPBROAD*. Then, we introduce another census algorithm where the path is just the reverse of the broadcasting path. It is noticeable that the two algorithms have different computation and communication steps.

Consider the problem of finding the minimum among 2^n numbers distributed in 2^n processors in a n-dimensional X-hypercube, and storing the result in P_0. The function *min* in algorithms *MIN0* finds the smaller number between given two numbers. We assume that every processor P_b has another register Y_b to be used for storing a temporary value.

Algorithm *MIN0*
  for all processor P_b **do in parallel**
    Compute S_b, a set of all adjacent processors of P_b;
    for k := 2 **to** n **by** 2 **do**
      send A_b to $Y_{b'}$ where $b_k = 1, b'_k = 0$ and $P_{b'} \in S_b$;
      $A_b := min(A_b, Y_b)$ for $b_k = 0$;
      send A_b to $Y_{b'}$ where $b_k b_{k-1} = 01$, $b'_k b'_{k-1} = 00$, and $P_{b'} \in S_b$;
      $A_b := min(A_b, Y_b)$ for $b_k b_{k-1} = 00$;
    endfor
  endfor
  if n is odd **then**
    send A_b to $Y_{b'}$ where $b_n = 1, b'_n = 0$ and $P_{b'} \in S_b$;
    $A_b := min(A_b, Y_b)$ for $b_n = 0$;
  endif
end MIN0

Theorem 5.5: Algorithm *MIN0* finds the minimum of the 2^n numbers stored in the 2^n processors of Q_n^X and stores the result in P_0 in $O(n)$ computation steps and n communication steps.

Proof: First, consider the case for even n. Algorithm *MIN0* computes a set S_b of adjacent processors of each processor P_b. By definition 2.2, we know that there exists a unique processor $P_{b'}$ among the processors in S_b such that b_k=1 and b'_k=0 or $b_k b_{k-1}$=01 and $b'_k b'_{k-1}$=00 for even k. At the first iteration of **for**-loop, algorithm MIN0 computes the minima among numbers stored in P_b's such that their binary addresses b's are $b_n b_{n-1}...b_3 00$, $b_n b_{n-1}...b_3 01$, $b_n b_{n-1}...b_3 10$, and $b_n b_{n-1}...b_3 11$, then store them in P_b's such that b's are $b_n b_{n-1}...b_3 00$ in parallel. At second iteration, it computes the minima among numbers that are obtained at first iteration and stored in P_b's of which addresses are $b_n b_{n-1}...b_5 0000$, $b_n b_{n-1}...b_5 0100$, $b_n b_{n-1}...b_5 1000$, and $b_n b_{n-1}...b_5 1100$, then store them in P_b's of which addresses are $b_n b_{n-1}...b_5 0000$. Therefore, the minimum of 2^n numbers stored in Q_n^T is guaranteed to be obtained at P_0 after $\lceil (n+1)/2 \rceil$ iterations.

Now consider the case for odd n. The inner **for**-loop iterates $\lceil (n+1)/2 \rceil$-1 times and finds minima in the same way as in the case for even n. Finally the parallel **for**-loop finds two minima and store them in two processors $P_{000...0}$ and $P_{100...0}$. Obviously, the last **if** statement finds a minimum between them and stores it in $P_{000...0}$ which is P_0.

Since communication instruction, **send**, is executed two times for each **for**-loop iteration, it takes n communication steps. Computation needed is to compute S_b and $min(A_b, A_{b'})$. S_b can be obtained in $O(n)$ computation step using definition 2.2 and $min(A_b, A_{b'})$ takes constant time. Therefore it takes $O(n)$ computation steps. This completes proof.

□

As in the broadcasting algorithm, we may have a new census algorithm to find minimum and to store it in any processor P_s by using procedure ADDR_RELABELING(s).

Algorithm *MIN1(s)*
Use procedure *ADDR_RELABELING*(s) to convert in parallel the address s of P_s to $s' = 0$ and the addresses of all other P_i to i';
Use the new addresses of processors and algorithm *MIN0* to find and store minimum in P_0;
end *MIN1(s)*

Theorem 5.6: Algorithm *MIN1(s)* computes the minimum of the 2^n numbers stored in 2^n processors of Q_n^X and store the result in any processor P_s $O(n)$ computation steps and n communication steps.

Proof: Directly follows theorem 5.3 and theorem 5.5.

□

The communication complexity of above census algorithms are not optimal whereas they have the optimal computation complexity. In a sense that communication cost dominates computation cost in parallel computing, it is desirable to achieve optimal communication complexity. Here, we present another census algorithm to achieve the optimal communication complexity although the optimal computation complexity is sacrificed. We assume that each processor has n registers connected to each of its adjacent processors so that it can receive at most n messages at each communication step.

The basic idea of this algorithm is to reverse the broadcasting path which is a spanning tree with height of $\lceil (n+1)/2 \rceil$. The broadcasting path was constructed by obtaining the addresses of each node's *sons* in the spanning tree of Q_n^X. Reversing the broadcasting path is to find the *father* of each node. This can be obtained by modifying the computation part of the algorithm *ADAPBROAD* that created broadcasting path. Then, every *leaf* sends a message to its father. If internal nodes perform computational tasks and send its results to their fathers only after receiving the messages from all their sons, the the minimum among the numbers stored in Q_n^X can obviously be found and stored in root which is P_0 after $\lceil (n+1)/2 \rceil$ communication steps.

Algorithm *MIN2*
(* part 1 *)
Compute S_b, the subset of processor $P_{b'}$ adjacent to P_b by executing the computation part in *ADAPBROAD* ;
for all processor P_b **do in parallel**
Compute N_b, the number of elements in S_b ;
endfor
(* part 2 *)
for all processor P_b **do in parallel**
Compute $R_b := \{P_{b'} | P_{b'}$ is adjacent to P_b and $RBIT(b) < RBIT(b')\}$
for k:= 2 **to** n **by** 2 **do**
if the type of *prefix(n,b;k)* is X_e **then**
case $b_k b_{k-1}$ **of**
11: **if** $b_{k+2}b_{k+1}$ is 11 **then**
$R_b := \{P_{b'} \mid b'_{k-1}=0$ and $b'_i=b_i$ for $i \neq k-1\}$
$- \{P_{b'} \mid b'_{k-1}=0\}$;
otherwise : do nothing;
endcase
else if the type of *prefix(n,b;k)* is C_e **then**
case $b_k b_{k-1}$ **of**
10 : $R_b := R_b \cup \{P_{b'} | b'_{k+2}\, b'_{k+1}=00$ and $b'_i=b_i$
for $i \neq k+2$ or $i \neq k+1\}$;
01,11: $R_b := R_b - \{P_{b'} \mid b'_k b'_{k-1} = 10\}$;
otherwise : do nothing;
endcase

```
                endif
            endfor
        endfor
        (* part 3 *)
        for all processor P_b do in parallel
            if N_b is 0 then
                send A_b to A_b' such that P_b' ∈ R_b;
        endfor
        for i := 1 to ⌈(n+1)/2⌉-1 do
            for all processor P_b do in parallel
                if P_b received message at previous step then
                    N_b := N_b - 1;
                if N_b is 0 then
                    Compute the minimum among the numbers that P_b
                    has received and send it to A_b' such that P_b' ∈ R_b;
            endfor
        endfor
end MIN2
```

Theorem 5.7: Algorithm *MIN2* finds the minimum of the 2^n numbers stored in 2^n processors of Q_n^X and store the result in P_0 in $O(n^2)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps.

Proof: The algorithm *MIN* 2 consists of three parts. Part 1 computes the number of processors for each processor P_b to receive a message from in the census operation. This number N_b allows a processor to be identified as a leaf processor in the broadcasting spanning tree of Q_n^X and it allows internal processors to know when they send a computational results. Part 3 is to send a message from leaf processor to root processor. By the discussion given before algorithm, the correctness of part 1 and part 3 is obvious.

Now we only need to prove that part 2 reverses the broadcasting path correctly. A processor P_b ($P_{b'}$) in this algorithm is corresponding to a processor $P_{b'}$ (P_b) in computation part of algorithm *ADAPBROAD* . Therefore, it can be proved by showing that

R_b in part 2 has only one element $P_{b'}$ which is the same as P_b in computation part of algorithm *ADAPBROAD*. Part 2 has two major steps. First, each processor P_b computes a subset of its adjacent processors $R_b := \{P_{b'} \mid P_{b'}$ is adjacent to P_b and $RBIT(b) < RBIT(b')\}$. Second, the **for**-loop rearranges R_b's so that it has only one element $P_{b'}$. Consider the binary label of P_b, $b_n b_{n-1} \ldots b_1$ for $RBIT(b)=r$. In order for $P_{b'}$ to be included in R_b in the first step of part 2, the binary string b' must satisfy the definition 2.2 and $RBIT(b) < RBIT(b')$. By the definition 2.2, we need to check at most two bits to determine adjacent processors. To check if $P_{b'}$ is the same as P_b in computation part of algorithm *ADAPBROAD*, we need to see the proof of theorem 5.2. We have the following four cases as in the proof of theorem 5.2.

Case 1: The type of *prefix* $(n,b;r)$ is C_e for even r.

There must be some $P_{b'}$ which are adjacent to P_b, such that

$(b_r b_{r-1}, b'_r b'_{r-1}) \in \{(10,01),(10,11)\}$ for $b_i = b'_i$ for $i \neq r$ and $i \neq r-1$.

Because $RBIT(b) > RBIT(b')$, $P_{b'}$ in Case 1 can not be an element of R_b. However, the **for**-loop in computation part forces the $P_{b'}$ to be an element of a unique set R_b. Since the **for**-loop in computation part makes the R_b have only one element $P_{b'}$ such that $b_{r+1} \neq b'_{r+1}$ or $b_{r+2} \neq b'_{r+2}$ and $b_i = b'_i$ for $i \neq r+1$ or $i \neq r+2$ respectively when *prefix(n,b;r)* is of the type C_e and $b_{r+2} b_{r+1} = 10$ even though $RBIT(b) = RBIT(b')$. This $P_{b'}$ is exactly same as P_b in its corresponding case of proof for theorem 5.2.

Case 2: The type of *prefix* $(n,b;r+1)$ is C_e for odd r.

There must be some $P_{b'}$ which are adjacent to $P_{b'}$, such that

$(b_{r+1} b_r, b'_{r+1} b'_r) \in \{(01,00),(01,10),(11,00),(11,10)\}$ for odd r and $b_i = b'_i$ for

$i \neq r$ and $i \neq r-1$.

Because $RBIT(b) < RBIT(b')$, $P_{b'}$ in Case 2 can be an elements of more than one R_b. However, the **for**-loop in computation part subtracts $P_{b'}$ for $b'_{r+1}b'_r=10$ from R_b if $b_{r+1}b_r = 01$ or 11 and *prefix(n,b;r+1)* is of type C_e, $P_{b'}$ is the only element of a set R_b for $b_{r+1}b_r = 00$ and $b_i=b'_i$ for $i \neq r$. This $P_{b'}$ is exactly the same as P_b in the case 2 of proof for theorem 5.2.

Case 3: The type of *prefix* $(n,b;r)$ is not C_e for even r.

There must be some adjacent $P_{b'}$, which are adjacent to $P_{b'}$, such that

$(b_r b_{r-1},b'_r b'_{r-1}) \in \{ (10,00),(10,11) \}$ for $b_i=b'_i$ for $i \neq r$ and $i \neq r-1$.

If $(b_k b_{k-1},b'_k b'_{k-1}) = (10,11)$, $P_{b'}$ can not be an element of R_b because $RBIT(b) > RBIT(b')$. Therefore, $P_{b'}$ in case 3 is the only element of a set R_b. This $P_{b'}$ is also the same as P_b in the case 3 of proof for theorem 5.2.

Case 4: The type of *prefix* $(n,b;r)$ is not C_e for odd r.

There must be some adjacent $P_{b'}$, which are adjacent to $P_{b'}$, such that

$(b_{r+1}b_r,b'_{r+1}b'_r) \in \{ (01,00),(01,11), (11,01),(11,10) \}$ for $b_i=b'_i$ for $i \neq r+1$ and $i \neq r$.

If $(b_{r+1}b_r,b'_{r+1}b'_r) \in \{ (01,11),(11,01) \}$, $P_{b'}$ can not be an element of R_b because $RBIT(b) = RBIT(b')$. Therefore, $P_{b'}$ in case 4 is also the only element of a set R_b. This $P_{b'}$ is also the same as P_b in the case 4 of proof for theorem 5.2. This completes proof for the correctness of algorithm *MIN*.

Now consider the complexity. Obviously, part 1 and part 2 has $O(n)$ computation complexity. The computation time in part 3 is dominated by the $O(n)$ time to

compute minimum of the numbers in each processor. Since this computational task and **send** are executed $\lceil (n+1)/2 \rceil$ times, *MIN2* takes $O(n^2)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps. This completes the proof.

□

By combining the procedure *ADDR_RELABELING* and the algorithm *MIN2* as in *MIN1*, we can claim the following theorem directly.

Theorem 5.8: The minimum of the 2^n numbers stored in 2^n processors of Q_n^X can be computed and stored in any processor P_s using *MIN2* and *ADDR_RELABELING* in $O(n^2)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps.

5.4. SIMD One-to-One Routing Algorithm

The one-to-one routing is to transfer data from a source processor to any other destinaton processor in a shortest path. This shortest path in routing operation can be obtained by two shortest path algorithms, *PATH_1* and *PATH_2*, introduced in chapter 3. Even though two sequential shortest path algorithms, which are the basis of routing algorithm, are developed in chapter 3, an optimal SIMD routing algorithm in X-hypercubes have remained unsolved. Most SIMD algorithms mapped on multicomputer system usually requires that data be distributed over the processors' local memories. This data distribution can be achieved by broadcasting algorithm. Like other SIMD algorithms, SIMD routing algorithm can be obtained by taking advantage of broadcasting algorithm. The algorithm *PATH_1* computes a shortest path for centralized control of circuit switching system in $O(n)$ time and the algorithm *PATH_2*

computes for packet switching system in $O(n^2)$ time. Even though *PATH_2* has higher time complexity than *PATH_1*, it is much simpler and can be executed in parallel in optimal time. Therefore we use algorithm *PATH_2* to achieve a shortest path from source to destination.

We assume that each processor P_i has three more registers, x_i, y_i, and $next_i$, in addition to those in *PATH_2* and *BROADCAST*. The variable x_i holds the data to be transferred and the y_i is destination variable where the data in x_i is stored. The variable $next_i$ holds the address of processor where the data are to be sent. Routing algorithm can be obtained by combining *PATH_2* and *BROADCAST* as follows.

Algorithm *ROUTE1*(x,s,y,d);
  Use algorithm *BROADCAST* to send destination address d from source processor s to all other processors;
  for all processor P_b **do in parallel**
    compute prefix(n,b;i) and prefix(n,d;i) for all i such that $1 \le i \le n$;
    let l and r be the index of the leftmost and rightmost distinct bit of b and d, respectively;
    $flag_b$:= false;
    if r is odd **then** $m^* := r+1$ **else** $m^* := r$;
    while $m^* < l - 1$ **and** $flag_b$ = false **do**
      if prefix(n,b;m^*) $\neq$ prefix(n,d;m^*) **and** $b_{m^*} b_{m^*-1} \neq d_{m^*} d_{m^*-1}$
      and $b_{m^*} \neq d_{m^*}$
      then begin
        $next_b := b_n \ldots b_{m^*+1} d_{m^*} d_{m^*-1} b_{m^*-2} \ldots b_1$;
        $flag_b$:= true;
      end
      else $m^* := m^* + 2$;
    endwhile
    if $flag_b$ = false **then**
      $next_b := b_n \ldots b_{l+1} d_l b_{l-1} \ldots b_1$;
    endfor
  for i = 1 **to** $\lceil (n+1)/2 \rceil - 1$ **do**
    if b is s **or** P_b received a message during previous step **then**
      send x_b to y_{next_b}
    else if P_b received message **and** b is d **then**

```
                stop;
        endfor
    end ROUTE1
```

We need to review the algorithm PATH_2 briefly for the correctness of algorithm *ROUTE* 1. There exists a nested WHILE loop in PATH_2. The inner loop finds the next processor on the shortest path from any processor to destination processor. The outer loop makes the inner loop executed repeatedly until destination is obtained. The above algorithm first send the address of destination processor to every other processors using the algorithm BROADCAST and every processor obtains next processor on the shortest path by the inner loop of the algorithm PATH_2, then send data along that shortest path. Obviously, algorithm *ROUTE* 1 performs transferring data correctly in $O(n)$ computation steps and $2\lceil(n+1)/2\rceil$ communication steps at worst case. Since algorithm *ROUTE* 1 hide a complicated topology of X-Hypercubes, it will be a foundation of other algorithms to be introduced in next sections. Hence, we can claim the following theorem

Theorem 5.9: Any processor in Q_n^X can send a message to any other processor in $O(n)$ computation steps and $\lceil(n+1)/2\rceil$ communication steps.

□

5.5. Summary

We present an optimal *SIMD* data broadcasting algorithm and two efficient census algorithms for the X-hypercube. The broadcasting algorithm requires no more than $O(n)$ computation steps and $\lceil(n+1)/2\rceil$ communication steps. The census algo-

rithm *MIN0* has $O(n)$ computation steps and n communication steps, *MIN2* has $O(n^2)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps. In contrast, for a conventional hypercube, there exist *SIMD* algorithms for data broadcasting and census function taking $O(n)$ computation steps and $\Omega(n)$ communication steps. This indicates that a X-hypercube is always better than a conventional hypercube as far as broadcasting is concerned.

In the case of census function, the X-hypercube is as good as a conventional hypercube, at least if we use algorithm *MIN0*. Although the census algorithm *MIN2* takes more computation steps, almost half the communication steps also indicates that the X-hypercube is a superior multicomputer structure in the sense that communication costs may overwhelmingly dominate the cost of parallel computing. Using our optimal data brodacasting algorithm, we also construct a one-to-one routing algorithm *ROUTE1*. This algorithm, combined with *ADDR_RELABELING*, can be used to transmit a message from any processor u to any other processor v in Q_n^X in $O(n)$ computation steps and $\lceil (n+1)/2 \rceil$ communication steps. Compared with the shortest path finding algorithms given in chapter 3, this algorithm is more efficient. However, when the shortest path algorithms given in chapter 3 are converted to data routing algorithms, only the processors on the shortest path participate in the data communication process, whereas for routing a message by *ROUTE1*, processors not on the shortest path also participate the data communication. In chapter 6, we will give a many-to-many data communication algorithm.

Chapter 6

SIMD Algorithms for a X-Hypercube

6.1. Introduction

It is well known fact that all processors in multicomputer system must be able to communicate with each other for efficient parallel computation. The most fundamental issues in data communication are the routing, broadcasting and census functions. Our investigations in previous chapters indicate that the X-hypercube is superior to the conventional hypercube as far as the data communication is concerned. Since the structure of the X-hypercube is similar to a hypercube, we expect that the X-hypercube can solve any problem as efficiently as a hypercube. Indeed, the embedding results given in chapter 2 shows that the X-hypercube can be used to execute any hypercube algorithm with the same time complexity by simulating the hypercube using embedding. However, since such an embedding has dilation 2 and congestion 2, considerable communication overhead exists in this simulation. Our question is : for a given problem π, can we design an algorithm for Q_n^X without using embedding simulation such that it is as efficient as any algorithm for π on Q_n ?

It is important to realize that most parallel algorithms can be divided into several subalgorithms, each for solving a specific subproblem. For example, the data communication algorithms for Q_n^X given in chapter 3 and chapter 4 can be used as subalgorithms for different data communication problems. We have shown that these algorithms are more efficient than their counterparts for a conventional hypercube. This is because in designing these algorithms we have tried to fully explore the special topo-

logical properties of a X-hypercube. To design efficient parallel algorithms for Q_n^X, we may identify some fundamental computational problems which are the basis of a wide variety of more complicated problems. If we can obtain efficient algorithms for these basic problems, we can use these algorithms as building blocks to construct efficient algorithms for more complicated problems.

In this chapter, we consider a set of basic problems. We design SIMD parallel algorithms for these problems by exploring the combinatorial structure of the X-hypercube. Each of these algorithms takes no more communication steps than the best implementation of its counterpart on a conventional hypercube. In other words, in the design of these algorithms, simulation by embedding Q_n into Q_n^X is avoided. Our results show that for many problems, we can design efficient parallel algorithms for Q_n^X by carefully exploring the connectivities of the X-hypercube. Combined with the results given in the previous chapters, these algorithms indicate that the X-hypercube can be superior over a conventional hypercube.

In the algorithms to follow, we assume that the processors are connected by a full-duplex link, i.e., two adjacent processors can simultaneously send and receive data from each other. All intercommunication will be established by the statement **send**. We also assume that each processor P_b has enough local memory space to handle a given problem.

6.2. Many-to-Many Routing Algorithm

The many-to-many routing is to transfer data from two or more source processors to their distinct destination processors in a multicomputer system. This communi-

cation problem is equivalent to finding many paths between processors and transmitting data along these paths concurrently. It is important to note that, in general, coordinating such multi-path communication is not easy. This is because if a processor P_b is in two or more paths, it is possible that two or more processors may attempt to send messages to the processor P_b at the same communication step, or a processor P_b has to send messages to two or more processors in order to complete all communications in a predetermined number of communication steps. In such a situation, we say that a communication conflict occurs. In what follows, we give a conflict-free, many-to-many routing algorithm for a class of data communication problems. As before, we denote a local variable owned by processor P_b using subscript b. For example, a variable x in processor P_b is x_b. In each of the parallel procedures given in this chapter, a parameter, say x, is a vector of variables $(x_1, x_2,...,x_N)$, with $N = 2^n$ and where x_b is a variable associated with processor P_b.

For the many-to-many routing problem, we assume initially that each variable $dest_b$ contains the address of the destination processor P_b where we intend to send a message. If P_b does not intend to send a message to any other processor, then $dest_b$ = nil, a special value. Consider the problem of transferring messages from P_b to P_{dest_b} simultaneously for all $b \in S \subset \mathbf{N} = \{1,2,...,2^n\}$. In case of an arbitrary permutation, the transfer can be performed in $O(n^2)$ steps using bitonic sorting. Consider the following algorithm, in which $\sigma = (a_n, a_{n-1},...,a_1)$ is a permutation of $\mathbf{n}=\{1,2,...,n\}$, $dest_b(j)$ is the j-th bit of $dest_b$. We use $b//i$ to denote b' such that $b'_i \neq b_i$ and $P_{b'}$ is connected to P_b in Q_n^X. Each processor P_b has a buffer $buffer_b$, which contains the messages P_b has received but not sent out. Note that every P_b uses the same

permutation σ for guiding the data transmission.

Algorithm *ROUTE2(x,dest,σ,z)*

for all processors P_b **do in parallel**

if $dest_b \neq$ nil **then** put $(x_b, dest_b)$ in $buffer_b$;

for $j := 1$ to n **do**

if $(x_i, dest_i)$ is in $buffer_b$ **and** $dest_i(a_j) \neq b_{a_j}$ **then**

Remove $(x_i, dest_i)$ from $buffer_b$ and **send** it to $buffer_{b//a_j}$;

endfor

if $(x_b, dest_b)$ is in $buffer_b$ **then** $z_b := x_b$;

endfor

end *ROUTE2*

In general, this algorithm may cause communication conflict. When this is the case, the many-to-many routing specified by permutation σ can not be completed, i.e., some messages may not be able to reach its destination processor. However, we can avoid such conflicts by handling the permutation σ technically. In section 6.4, we consider several problems for which using *ROUTE* 2 algorithm, data communication is conflict-free.

6.3. Parallel Computation of Prefix Sum

When a sequence of n numbers $X = \{x_0, x_1, \ldots, x_{n-1}\}$ for $n \geq 1$ is given, consider the problem of computing all n partial sums $S = \{s_0, s_1, \ldots s_{n-1}\}$, where $s_i = x_0 + x_1 + \ldots + x_i$ for $0 \leq i \leq n\text{-}1$. These sums are referred to as the prefix sums of X. For exam-

ple, assuming n=8, x_0=2, x_1=1, x_2=4, x_3=6, x_4=5, x_5=0, x_6=3, and x_7=7, then s_0=2, s_1=3, s_2=7, s_3=13, s_4=18, s_5=18, s_6=21, and s_7=28. The prefix sum can be used to solve many decision and optimization problems, for example, the knapsack problem and job sequencing with deadline. [Akl89] In this chapter, we will use our prefix sum algorithm as a subalgorithm for the packing algorithm which will be introduced shortly.

Now we introduce a parallel algorithm to compute prefix sums. Each processor P_b has three variables, x_b ,$total_b$, and $temp_b$. Initially x_b has the value to be computed for the prefix sum, and when the algorithm terminates, x_b and $total_b$ contains the partial sums and total sum, respectively. That is, the total sum is computed at every processor in addition to the partial sums.

```
Algorithm PREFIX(x);
    for all processor P_b do in parallel
        total_b := x_b;
        for i:= 1 to n do
            if prefix(n,b;i) is of type C_e then
                send total_b to temp_b' such that b_i b_{i-1}=b'_i b'_{i-1} and b_k=b'_k
                for k≠i and k≠i-1;
            else
                send total_b to temp_b' such that b_i=b'_i and b_k=b'_k for k≠i;
            endif
            if b_i is 1 then
                x_b := temp_b + x_b;
                total_b := temp_b + total_b;
            else
                total_b := total_b + temp_b;
            endif
        endfor
    endfor
end PREFIX
```

The inner **for**-loop iterates n times and every iteration requires constant time. Hence the complexity of this algorithm is $O(n)$. The operations of *PREFIX* are illustrated in table-1 for $n=3$.

Table-1: Calculation of prefix sum and total sum in algorithm *PREFIX*

processor address	000	001	010	011	100	101	110	111
x_b	2	1	4	6	5	0	3	7
$total_b$	0	0	0	0	0	0	0	0
x_b after 1^{st} iteration	2	3	4	10	5	5	3	10
$total_b$ after 1^{st} iteration	3	3	10	10	5	5	10	10
x_b after 2^{nd} iteration	2	3	7	13	5	5	8	15
$total_b$ after 2^{nd} iteration	13	13	13	13	15	15	15	15
x_b after 3^{rd} iteration	2	3	7	13	18	18	21	28
$total_b$ after 3^{rd} iteration	28	28	28	28	28	28	28	28

6.4. SIMD Packing Algorithm

In the SIMD computing model, most instructions are usually executed conditionally. That is, some processors satisfying a condition may run, while other processors do not. We define the running processors as *active* processors. Packing is the task of moving data from active processors to lower numbered processors that have space in such a way that the relative order of the data among processors is unchanged. The key operations in packing are to find the destination of each active processor and to

transfer the data from source processor to its destination processor. Transferring data is implemented by algorithm *ROUTE2*. The destination processor can be found by computing the number of preceding active processors as follows. Here, we can apply the idea of the prefix sum. Since only the active processors are supposed to send data to their destination, only active processors need to compute the prefix sum of number of active processors. Therefore, we need to change the algorithm *PREFIX* as in the algorithm LOWDESTINATION. The function *BIN*(*int*) converts a given integer *int* into its equivalent binary number. That is, *BIN*(5) returns 101.

```
Procedure LOWDESTINATION(active ,dest );
    for all processor P_b do in parallel
        total_b := active_b;
        z_b := active_b;
        for i:= 1 to n do
            if prefix(n,b;i) is of type C_e then
                send total_b to temp_b' such that b_i b_{i-1}=b'_i b'_{i-1} and b_k=b'_k
                for k≠i and k≠i-1;
            else
                send total_b to temp_b' such that b_i=b'_i and b_k=b'_k for k≠i;
            endif
            if b_i is 1 then
                z_b := z_b + temp_b;
                total_b := total_b + temp_b;
            else
                total_b := total_b + temp_b;
        endfor
        if active_b = 0 then
            dest_b := nil
        else
            dest_b := BIN(z_b -1);
    endfor
end LOWDESTINATION
```

Initially, the variable $active_b$ has the value 1 if processor P_b is active, otherwise, its value is 0. After *PREFIX* is executed, the value of variable z_b computes the

number of active processors which precede P_b, including P_b itself, only when P_b is active. By technically subtracting 1 from z_b and converting it into a binary number, every active processor has its destination in the variable $dest_b$. Since the time complexity of *PREFIX* is O(n) and the parallel **for**-loop takes constant time, *LOWDESTINATION* takes O(n) time.

The packing operation can be implemented in O(n) time by combining *ROUTE* 2 and *LOWDESITINATION*, as in the following algorithm *PACK*.

Algorithm *PACK*(*active* ,*x* ,*y*);
    LOWDESTINATION(*active* ,*dest*);
    ROUTE2(*x* ,*dest* ,(*n* ,...2,1),*y*);
end *PACK*

The algorithm *PACK* can be illustrated more in detail by the example in table-2.

Table-2: Data transfer in algorithm *PACK*

processor address	000	001	010	011	100	101	110	111
x_b	2	1	4	6	3	0	3	7
$active_b$	0	1	0	1	1	0	1	0
z_b	0	1	1	2	3	3	4	4
$dest_b$ in integer	nil	0	nil	1	2	nil	3	nil
$dest_b$ in binary	nil	000	nil	001	010	nil	011	nil
y_b	1	6	3	3				

Each processor P_b has data to be transferred in the variable x_b and the value of vari-

able $active_b$ is given. The procedure *LOWDESTINATION* finds the destination processor by calculating prefix sum of *active*, and the procedure *ROUTE2* sends data to the variable y_{dest} as shown in the table-2. The initial values of $active_b$ are given in the table-2.

We will now show that there is no communication conflict in the packing operation. Suppose that there are two or more data in the same buffer at the end of the jth iteration. This means that the data have originated from the same $(j+1)$-dimensional subcube. On the other hand, their final destinations must be the same processor in different $(j+1)$-dimensional subcube. This would be a contradiction, because the distance between two or more data can not increase in the packing operation. Therefore, we know that algorithm *PACK* does not cause any communication conflicts.

Uppacking transfers data from active processors to higher numbered processors that have space in such a way that relative oreder of the data among processors is unchanged. The only difference between packing and uppacking is for each active processor to find a higher numbered destination instead of lower numbered one. The higher numbered destination can be found by subtracting the total number of active processors from and adding the total number of processors to the lower numbered destination. This can be done in O(n) time as follows.

Procedure *HIGHDESTINATION(active,dest)*
  for all processor P_b **do in parallel**
    $total_b := active_b$;
    $z_b := active_b$;
    for i:= 1 to n **do**
      if prefix(n,b;i) is of type C_e **then**
        send $total_b$ to $temp_{b'}$ such that $b_i b_{i-1}=\overline{b'}_i \overline{b'}_{i-1}$ and $b_k=b'_k$ for k≠i and k≠i-1;

```
            else
                send total_b to temp_b' such that b_i=b'_i and b_k=b'_k for k≠i;
            endif
            if b_i is 1 then
                z_b := temp_b + z_b;
                total_b :=temp_b + total_b;
            else
                total_b :=temp_b + total_b;
        endfor
        if active_b = 0 then
            dest_b := nil
        else
            dest_b := BIN(z_b +n -total_b -1);
    endfor
end HIGHDESTINATION
```

The uppacking operation can be implemented in O(n) time by the algorithm *UPPACK*, which is almost same as the algorithm *PACK*.

Table-3: Data transfer in algorithm *UPPACK*

processor address	000	001	010	011	100	101	110	111
x_b	2	1	4	6	5	0	3	7
$active_b$	0	1	0	1	1	0	1	0
z_b	0	1	1	2	3	3	4	4
$total_b$	4	4	4	4	4	4	4	4
$dest_b$ in integer	nil	4	nil	5	6	nil	7	nil
$dest_b$ in binary	nil	100	nil	101	110	nil	111	nil
y_b					1	6	5	3

Algorithm *UPPACK(active,x,y)*
    HIGHDESTINATION(active,dest);
    ROUTE2(x,dest,(n...2,1),y);
end *UPPACK*

The algorithm *UPPACK* works in the same way as *PACK*. By the same reason as in *PACK*, *UPPACK* does not cause any communication conflicts. Its detailed operations are illustrated by an example given in table-3.

6.5. Parallel Sorting Algorithms

Sorting is a communication intensive operation performed very frequently on a parallel computer. Many kinds of sorting problems and their solutions have been suggested over time. As multicomputer systems based on interconnection network became popular, parallel sorting algorithms, which take advantage of the topology of interconnection networks, have been introduced. For example, Batcher's bitonic sort fits nicely on a perfect shuffle machine or a hypercube machine. In addition, an odd-even transposition sort is suitable for linear array and two-dimensional meshes. Among these parallel sorting algorithms, Batcher's bitonic sort has been considered to have good performance on a hypercube. Whereas most sorting methods, including Batcher's bitonic sort, compares keys to rearrange them, a radix exchange sort compares bits of the binary representation of the keys. It is usually faster and simpler for small keys. The major operation in the radix exchange sort is packing and uppacking, which have been discussed in the previous section. In this section, we implement a radix exchange sort and Batcher's bitonic sort on a X-hypercube.

6.5.1. Radix Exchange Sort

The radix exchange sort makes use of the binary representation of the keys. Instead of comparing two keys with each other, it inspects individual bits of the keys to see if they are 0 or 1. Let's briefly review the sequential radix exchange sort described in [KN]:

(1-1) Sort the sequence on its most significant bit, so that all keys which have a leading 0 come before all keys which have a leading 1. This sorting is done by finding the leftmost key k_i which has the leading 1, and the rightmost key k_j with a leading 0. Then k_i and k_j are exchanged and the process is repeated until i > j.

(1-2) Let f_0 be the elements with leading 0, and let f_1 be the others. Apply the radix exchange sorting method to f_0 (starting now at the second bit from the left instead of the most significant bit), until f_0 is completely sorted; then do the same for f_1.

Since the step (1-1) exchanges the leftmost key k_i and rightmost key k_j, this data movement does not preserve the relative order of exchanged data among processors. Our interest is to utilize the packing operations on the radix sort. Therefore, we convert the above algorithm into the following one, so that we can keep the relative order of data after exchanging data.

(2-1) Sort the sequence on its least significant bit, so that all keys which have a 0 come before all keys having a 1. This step can be done in a little bit different way from the step (1-1). Find the leftmost key k_i which has the trailing 0, and the leftmost key k_j with a trailing 1. Then exchange them and repeat this until i or j is greater than the number of sort keys.

(2-2) Repeat step (2-1) on every index of the sort key. After applying to the most significant bit, the sorted list will be obtained.

We can explore inherent parallelism in this section. That is, step (2-1) can be done in parallel and it fits to the X-hypercube. Actually it is parallel packing and uppacking operations. Now we implement a parallel radix exchange sort using the procedures *PACK* and *UPPACK* as follows. The algorithm *RADIX* uses a new function called *BIT*. We assume that the function *BIT*(*int*,*j*) returns the j^{th} binary value of a given integer *int*. For example, *BIT*(5,1) is 1. We assume that each processor has two variables, low_b and $high_b$, instead of the variable $active_b$. The variables low_b and $high_b$ are used for calculating the low destination and the high destination, respectively. It is assumed that 2^n sort keys are initially evenly distributed over Q_n^X. Each local variable x_b has the key to be sorted, and the sorted key is stored in y_b. The index of the most significant bit is assumed m.

```
Algorithm RADIX
    for j := 1 to m do
        for all processors P_b do in parallel
            if BIT(x_b, j) = 0 then
                low_b := 1;
            else
                high_b := 1;
            endif
        endfor
        PACK(low,x,y);
        UPPACK(high,x,y);
    endfor
end RADIX
```

Since both *PACK* and *UPPACK* run in $O(n)$ time, the algorithm *RADIX* runs in $O(mn)$ time. The feature which makes the radix exchange sort attractive is that the

time required to sort depends on the number of bits in each key. We will illustrate this algorithm by an example. For simplicity, we assume that the sort keys are unsigned integers with 3 bits. Table-4 shows the procedure of how sort keys are rearranged.

Table-4: Sorting keys on Q_n^X by radix exchange sort

processor address	000	001	010	011	100	101	110	111
x	2	1	4	6	5	0	3	7
BIN(x)	010	001	100	110	101	000	011	111
1^{st} iteration	010	100	110	000	001	101	011	111
2^{nd} iteration	100	000	001	101	010	110	011	111
3^{rd} iteration	000	001	010	011	100	101	110	111

6.5.2. Batcher's Bitonic Sort

Batcher's bitonic sort has been the basis for sorting algorithms under several models for parallel computation models [Bat68]. This sorting method is sometimes called the bitonic merge sort since it sorts keys by merging bitonic sequence. A bitonic sequence is a sequence of numbers $k_0, k_1, ..., k_{n-1}$ with the property that

1) there exists an index i, $0 \leq i \leq$ n-1, such that k_0 through k_i is monotonically increasing and k_i through k_{n-1} is monotonically decreasing, or else

2) there exists a cyclic shift of indices so that the first condition is satisfied.

The bitonic merge produces the following two sequences from a original bitonic sequence

$\min(k_0,k_{n/2}),\min(k_1,k_{n/2+1}),\ldots,\min(k_{n/2-1},k_{n-1})$ and

$\max(k_0,k_{n/2}),\max(k_1,k_{n/2+1}),\ldots,\max(k_{n/2-1},k_{n-1})$

Since both sequences are bitonic and every element in the first sequence is smaller than every element in the second sequence, it can be easily verified that a bitonic sequence of length n is sorted in $\log n$ steps by recursively applying the min and max functions to resulting bitonic sequences in parallel. A list of n elements to be sorted can be considered as a set of n unsorted elements of length 1. We can obtain $n/2$ bitonic sequences of length 2 by merging as following:

$\min(k_0,k_1),\max(k_0,k_1),\ \min(k_2,k_3),\max(k_2,k_3),\ldots,\ \min(k_{n-2},k_{n-1}),\max(k_{n-2},k_{n-1})$

By merging successively larger and larger bitonic sequences, n elements can be sorted.

As shown above, Batcher's bitonic sort always compares elements whose indices differ in exactly one bit. Therefore it fits nicely to a hypercube, since processors in a hypercube are connected if their indices differ in exactly one bit. The Batcher's bitonic sort has been implemented on hypercube [Qui87] as follows. It sorts 2^n keys, which are distributed evenly in all processors, in ascending order. Each processor P_b is assumed to have three local variable a_b, b_b, and t_b. The key is stored initially in each a_b. After sorting, a_b has the sorted key.

```
Algorithm BITONIC-SORT-H
  for i:=1 to n do
    for j:=i downto 1 do
      for all P_b do in parallel
        if BIT(b,j) = 1 then
          send a_b to t_b' where b_j=b'_j and b_k=b'_k for k≠j;
          if BIT(b,i+1) = 0 then
```

```
            b_b := max(t_b, a_b);
            a_b := min(t_b, a_b);
          else
            b_b := min(t_b, a_b);
            a_b := max(t_b, a_b);
          endif
        endif
      if BIT(b,j) = 0 then
        send b_b to a_b' where b_j = b̄'_j and b_k = b'_k for k≠j;
      endif
    endfor
  endfor
endfor
end BITONIC-SORT-H
```

An adjacent processor with regard to a certain index position can be determined in constant time and the parallel **for**-loop takes constant time. Hence, the time complexity of *BITONIC-SORT-H* is $O(n^2)$.

Since any algorithm on a hypercube machine can be implemented on a X-hypercube machine with the same performance by theorem 2.2, Batcher's bitonic sort can also run on a n-dimensional X-hypercube in $O(n^2)$ computation steps. This has been shown already by the algorithm *BITONICMERGESORT*, mentioned in chapter 2. It is just simulating a hypercube by algorithm *EMBED1* given in chapter 2. Since Q_n is embedded into Q_n^X with dilation 2, *BITONICMERGESORT* has communication overhead whenever processors are connected by a twisted edge.

Now, consider applying Batcher's bitonic sort to a X-hypercube directly. Observing algorithm *BITONIC-SORT-H*, we notice that one of two adjacent subcubes always has a monotonically increasing sequence, while the other has a monotonically decreasing sequence at the end of each iteration of the outmost **for**-loop. This outmost **for**-loop performs the bitonic merge on a bitonic sequence. As index i

increases, it merges a larger bitonic sequence. The only difference between Q_n and Q_n^X is the twisted edge connecting the processors satisfying the condition (1) of definition 2.2. If two k-dimensional subcubes of X-hypercube are connected by twisted edges and each has a monotonically increasing or decreasing sequence for odd k, then $(k+1)$-dimensional subcube of X-hypercube actually has a bitonic sequence. Whenever prefix(n,b;k+1) is of the type C_e, two k-dimensional subcubes are connected by twisted edges. By checking the state of the prefix, we can maintain a bitonic sequences in X-hypercube during bitonic merge. Therefore we can implement Batcher's bitonic sort on a a X-hypercube without any communication overhead with same time complexity as on a hypercube by the following algorithm.

Algorithm *BITONIC-SORT-XH*
  for i:=1 **to** n **do**
    for j:=i **downto** 1 **do**
      for all P_b **do in parallel**
        if *BIT*(b,j) = 1 **then**
          if prefix(n,b;j) is of type C_e **then**
            send a_b to $t_{b'}$ where $b'_j b'_{j-1}=\bar{b}_j \bar{b}_{j-1}$ and $b_k=b'_k$ for k≠j and k≠j-1;
          else if prefix(n,b;j-1) is of type C_e and BIT(b,j-1) is 1 **then**
            send b_b to $t_{b'}$ where $b'_j=\bar{b}_j$ and $b_k=b'_k$ for k≠j;
          else
            send a_b to $t_{b'}$ where $b'_j=\bar{b}_j$ and $b_k=b'_k$ for k≠j;
          endif
          if *BIT*(b,i+1) = 0 **then**
            $b_b := max(t_b, a_b)$;
            $a_b := min(t_b, a_b)$;
          else
            if prefix(n,b;j+1) is of type C_e **and** *BIT* (b,j+1) is 1 **then**
              $b_b := max(t_b, a_b)$;
              $a_b := min(t_b, a_b)$;
            else
              if prefix(n,b;j-1) is of type T_e **and** *BIT* (b,j-1) is 1 **then**
                $b_b := max(t_b, b_b)$;
                $a_b := min(t_b, a_b)$;

```
            else
              b_b := min(t_b, a_b);
              a_b := max(t_b, a_b);
            endif
          endif
        endif
      if BIT(b,j) = 0 then
        if prefix(n,b;j) is of type C_e then
          send b_b to a_b' where b'_j b'_{j-1} = b̄_j b̄_{j-1} and b_k = b'_k for k≠j and
          k≠j-1;
        else
          send b_b to a_b' where b'_j = b̄_j and b_k = b'_k for k≠j;
        endif
      endfor
    endfor
  endfor
end BITONIC-SORT-XH
```

6.6. Summary

We present the SIMD many-to-many routing algorithm, *ROUTE* 2, for a X-hypercube Q_n^X. Using this algorithm, we develope the parallel algorithm, *PREFIX*, to compute the prefix sum. We also develope parallel algorithms, *PACK* and *UPPACK*, to perform packing and uppacking operations. All these algorithms have O(n) time complexity.

We utilize *PACK* and *UPPACK* to implement a radix exchange sort on Q_n^X. Algorithm *RADIX* sorts 2^n keys in $O(n^2)$ time in a very simple way. We also show that Batcher's bitonic sort can be implemented on Q_n^X without any overhead. We believe that for many other problems, we can design algorithms for Q_n^X by directly considering the topological structure of Q_n^X. These algorithms are as efficient as their counterparts for a conventional hypercube Q_n.

Chapter 7

Algorithms for Data Communication on a Z-Cube Interconnection Network

7.1. Introduction

A multicomputer system based on a interconnection network has been recognized as a key in the design of parallel computers. A good interconnection network, in general, should have a low vertex degree, a small diameter, large number of disjoint paths between a pair of vertices, simple routing schemes, and easy hardware implementation. These parameters are interrelated to each other. For example, a higher vertex degree gives more disjoint paths between a pair of vertices, and a smaller diameter result in less delay in data communication. A higher vertex degree guarantees more disjoint paths and smaller diameter. However, a network with higher vertex degree is more difficult to implement in hardware. According to the current electronic technology, the most decisive factor in network design is the feasibility of VLSI implementation.

So far, we emphasize the effectiveness of the X-hypercube over the conventional hypercube from the point of view of data communication. In this chapter, we consider a new hypercube-like interconnection network called the Z-cube, which has the property to be more easily implemented in VLSI [ZL91]. In a VLSI layout, the area for wires connecting processors dominates the processor area. Therefore, the higher the vertex degree that a network has, the more area it needs. An n-dimensional Z- cube, denoted by Q_n^Z, where n must be a multiple of 4, has the same number of vertices as

an n-dimensional hypercube, Q_n. But the vertex degree of Q_n^Z is 3/4 of that of Q_n. Since the Z-cube also preserves most properties of the hypercube, it can be an another good alternative for the hypercube, especially from the point of view of VLSI implementation.

In this chapter, we present two different definitions for a Z-cube and explore some topological properties of a Z-cube. We introduce algorithms for the shortest path and for data broadcasting.

7.2. Structure of a Z-cube

An n dimensional Z-cube, denoted by Q_n^Z, can be viewed as a graph of 2^n vertices, where n is multiple of 4. Each vertex is labeled by a unique n-bit binary digits, with a value 0 through n-1, letting 1 index the least significant bit. The first definition of the Z-cube is as follows.

Definition 7.1: A n-dimensional Z-cube Q_n^Z, where n is multiple of 4, is a graph of 2^n vertices, such that two vertices $u = u_n u_{n-1} \ldots u_1$ and $v = v_n v_{n-1} \ldots v_1$ are connected by an edge if and only if for some m such that $1 \le m \le n/4$, $u_k = v_k$ for $k > 4m$ and $k < 4(m-1)$, and one of the following five conditions holds:

(1) $u_{4m} u_{4m-1} = v_{4m} v_{4m-1}$, and $u_{4m-2} u_{4m-3}$ are distinct in exactly one bit;

(2) $u_{4m} u_{4m-2} = v_{4m} v_{4m-2} \in \{00,11\}$, and $u_{4m-1} u_{4m-3} = \overline{v_{4m-1}} \overline{v_{4m-3}} \in \{01,10\}$;

(3) $u_{4m} u_{4m-2} = v_{4m} v_{4m-2} \in \{01,10\}$, and $u_{4m-1} u_{4m-3} = \overline{v_{4m-1}} \overline{v_{4m-3}} \in \{00,11\}$;

(4) $u_{4m} u_{4m-2} = \overline{v_{4m}} \overline{v_{4m-2}} \in \{01,10\}$, and $u_{4m-1} u_{4m-3} = v_{4m-1} v_{4m-3} \in \{01,10\}$;

(5) $u_{4m} u_{4m-2} = \overline{v_{4m}} \overline{v_{4m-2}} \in \{00,11\}$, and $u_{4m-1} u_{4m-3} = v_{4m-1} v_{4m-3} \in \{00,11\}$.

Figure 12 shows Q_4^Z. This figure also can be used to represent Q_8^Z by treating each vertex as a Q_4^Z. It implies that Q_{4m}^Z can be constructed recursively by treating each vertex as a $Q_{4(m-1)}^Z$. We call each of the sixteen $Q_{4(m-1)}^Z$ in Q_{4m}^Z as a $4(m-1)$-dimensional subcube of Q_{4m}^Z, which is denoted by $\zeta_{4(m-1)}$. Of course, these sixteen subcube are isomorphic to each other. To facilitate our presentation, we summarize this fact in the following lemma.

Lemma 7.1: For any $m > 0$, Q_{4m}^Z consists of sixteen $\zeta_{4(m-1)}$'s which are isomorphic to $Q_{4(m-1)}^Z$ by deleting the preceding 4 bits from each vertex label.

□

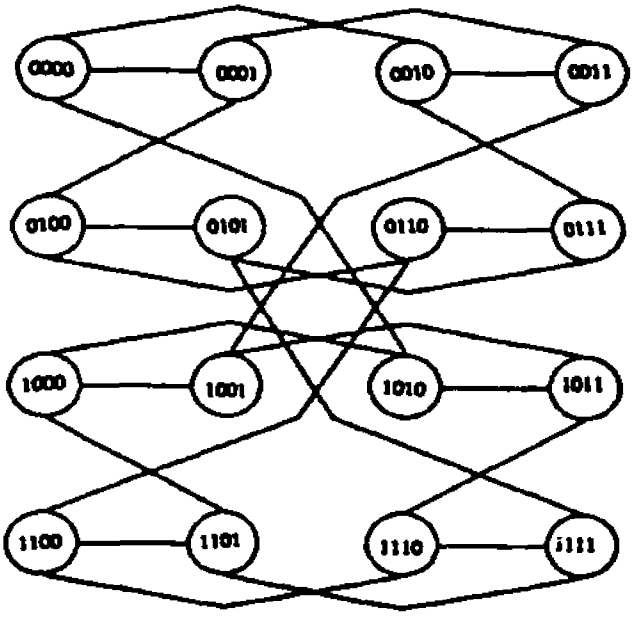


Figure 12: "Z"-shaped Q_4^Z

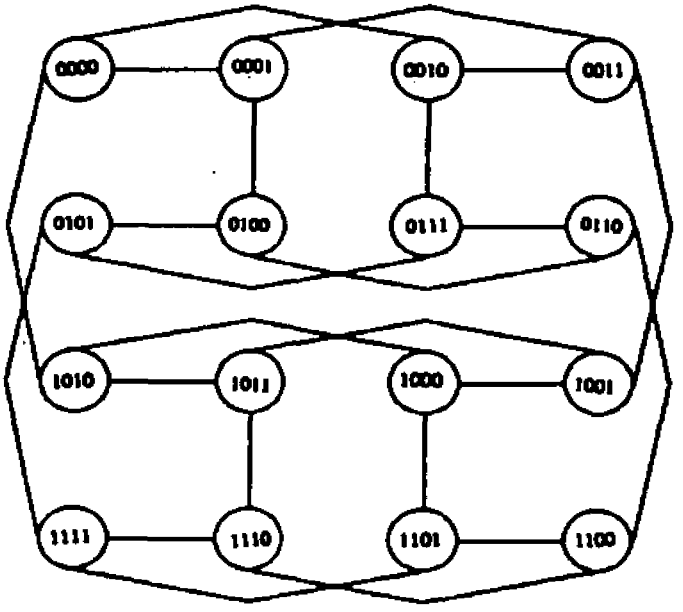


Figure 13: Hypercube-like Q_4^Z

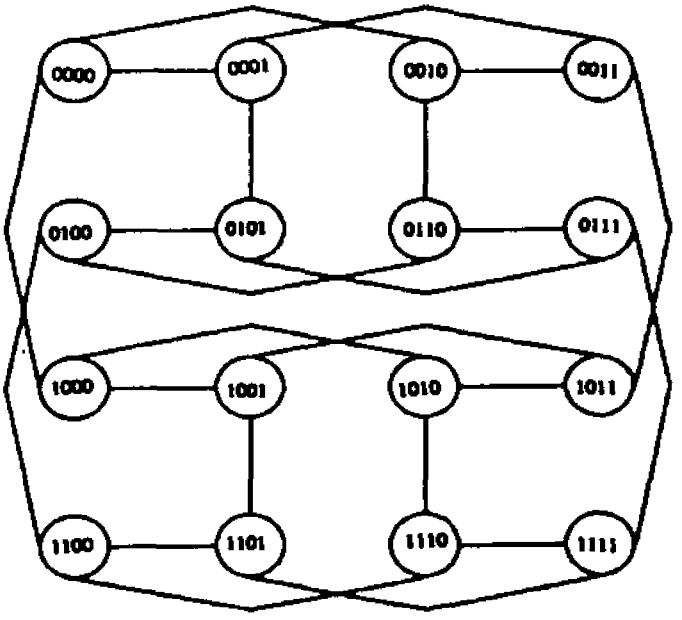


Figure 14: Relabeled Hypercube-like Q_4^Z

In figure 12, we see that the processors are connected in a "Z" shape. This is the reason why this network is called a Z-cube. This may give the impression that the Z-cube is totally different from a hypercube. If we rearrange some processors, as in figure 13, the Z-cube turns into a hypercube-like structure. Relabeling the processors in figure 13, we have another Z-cube in figure 14. Since the Z-cube in figure 14 is isomorphic to the one in figure 12, we have another definition for Q_4^Z. By lemma 7.1, we know that Q_n^Z is constructed recursively from Q_4^Z. Therefore, the following definition can be derived.

Definition 7.2: An n-dimensional Z-cube Q_n^Z, where n is multiple of 4, is a graph of 2^n vertices, such that two vertices $u = u_n u_{n-1} \ldots u_1$ and $v = v_n v_{n-1} \ldots v_1$ are connected by an edge if and only if for some m such that $1 \le m \le n/4$, $u_k = v_k$ for $k > 4m$ and $k < 4(m-1)$, and one of the following three conditions holds:

(1) $u_{4m} u_{4m-1} = v_{4m} v_{4m-1}$, and $u_{4m-2} u_{4m-3}$ are distinct in exactly one bit;

(2) $u_{4m} u_{4m-1} = \overline{v_{4m}} v_{4m-1}$ and $u_{4m-2} u_{4m-3} = v_{4m-2} v_{4m-3} \in \{00, 11\}$;

(3) $u_{4m} u_{4m-1} = v_{4m} \overline{v_{4m-1}}$ and $u_{4m-2} u_{4m-3} = v_{4m-2} v_{4m-3} \in \{01, 10\}$.

Since definition 7.2 is much simpler than definition 7.1, we will use definition 7.2 for investigating properties of the Z-cube. By definition 7.2, we know that the subgraphs induced by the vertex subset $\{b_{4m} b_{4m-1} \ldots b_1 | b_k = c_k$ for $k > 4m$ and $k < 4(m-1)\}$, where each c_k is a constant, are also isomorphic to each other and isomorphic to Q_4^Z by deleting all bits except 4 bits of $b_{4m} b_{4m-1} b_{4m-2} b_{4m-3}$. Therefore, we can claim the following lemma.

Lemma 7.2: For any $m > 0$, the subgraph which is induced by the vertex subset $\{b_{4m}b_{4m-1}...b_1 | b_k = c_k$ for $k > 4m$ and $k < 4(m-1)\}$, where each c_k is a constant, are isomorphic to Q_4^Z.

□

Now, we show that the 4-dimensional Z-cube is a vertex symmetric graph by giving the algorithm *ADDR_TRANS*, which transforms the address of each processor P_b to a new address b', such that $s' = 0000$ for a given address s and every other new address b' satisfies the condition in definition 7.2. In the algorithm to follow, $\oplus$ denotes the exclusive-or operation of two binary strings.

```
procedure ADDR_TRANS0(s)
    for all processor P_b do in parallel
        b' = b ⊕ s;
        case b_2b_1 of
            01,10: if b'_2b'_1 ≠ 01 and b'_2b'_1 ≠ 10 then
                        b'_4b'_3 := b'_3b'_4;
            00,11: if b'_2b'_1 ≠ 00 and b'_2b'_1 ≠ 11 then
                        b'_4b'_3 := b'_3b'_4;
        endcase
    endfor
end ADDR_TRANS0
```

Lemma 7.3: Procedure *ADDR_TRANS0(s)* relabels Q_4^Z in O(1) time such that if P_b has address $b = b_4b_3b_2b_1$, then P_b has a new address $b' = b'_4b'_3b'_2b'_1$ and $s' = 0000$.

Proof: Since algorithm *ADDR_TRANS0(s)* applies a one-to-one mapping function, the exclusive-or of s, to the label b of every processor P_b in first parallel **for**-loop and relabels b_4b_3 to another unique $b'_4b'_3$ in the second **for** loop, s will be transformed to $s' = 0000$ and each b will be transformed to a unique b'. From now on, we use b' as the relabeled b obtained by algorithm *ADDR_TRANS0(s)*.

We only need to show that for every two adjacent processors P_u and P_v, $P_{u'}$ and $P_{v'}$ are also adjacent to each other. In order for $P_{u'}$ and $P_{v'}$ to be adjacent to each other, u' and v' must satisfy one of the three conditions of definition 7.2. There are three cases.

Case 1: u_4u_3=v_4v_3 and u_2u_1 and v_2v_1 are distinct in exactly one bit.

Even after the exclusive-or function of s is applied in the first **for**-loop, $u'_4u'_3$=$v'_4v'_3$ and $u'_2u'_1$ and $v'_2v'_1$ are distinct in exactly one bit. In the second **for**-loop, $u'_4(v'_4)$ and $u'_3(v'_3)$ may be reversed. Even if they are reversed, $u'_3u'_4$=$v'_3v'_4$ because $u'_4u'_3$=$v'_4v'_3$. Hence, u' and v' satisfy condition 1 of definition 7.2.

Case 2: $u_4u_3 = \overline{v}_4v_3$ and $u_2u_1 = v_2v_1 \in \{00,11\}$.

After the first **for**-loop, it is obvious that $u'_4u'_3 = \overline{v'}_4v'sub3$ and $u'_2u'_1 = v'_2v'_1$. But $u'_2u'_1(v'_2v'_1)$ may be an element of either {00,11} or {01,10}. If they are an element of {00,11}, the second **for**-loop does nothing. Therefore, u' and v' satisfy condition 2 of definition 7.2. If $u'_2u'_1(v'_2v'_1)$ is an element of {01,10}, the second **for**-loop reverses $u'_4(v'_4)$ and $u'_3(v'_3)$. Hence, u' and v' satisfy condition 3 in definition 7.2.

Case 3: $u_4u_3 = v_4\overline{v}_3$ and $u_2u_1 = v_2v_1 \in \{01,10\}$.

After the first **for**-loop, it is obvious that $u'_4u'_3 = v'_4\overline{v'}_3$ and $u'_2u'_1 = v'_2v'_1$. But $u'_2u'_1(v'_2v'_1)$ may be an element of either {01,10} or {00,11}. If it is an element of {01,10}, the second **for**-loop does nothing. Therefore, u' and v' satisfy condition 3 of definition 7.2. If they are an element of {00,11}, the second **for**-loop reverses $u'_4(v'_4)$ and $u'_3(v'_3)$. Then, u' and v' satisfy condition 2 of definition 7.2.

We have considered all possible cases. Therefore, for any two adjacent processors P_u and P_v, $P_{u'}$ and $P_{v'}$ are also adjacent to each other. The procedure *ADDR_TRANS* obviously has time complexity of O(1). This completes the proof.

□

By extending the algorithm *ADDR_TRANS0* to higher dimension as in the algorithm *ADDR_TRANS1*, we obtain theorem 7.1.

Theorem 7.1: An n-dimensional Z-cube Q_n^Z can be relabeled in O(n) time such that if P_b has an address $b = b_n\ b_{n-1}...b_1$, then for any vertex s, P_b has a new address $b' = b'_n b'_{n-1}...b'_1$ and $s' = 00...0$.

Proof: For every vertex P_b, apply the algorithm *ADDR_TRANS* to the first 4 bits. Then $s_n s_{n-1}...s_1$ is change to $0000 s_{n-4} s_{n-5}...s_1$ and $b_n b_{n-1}...b_1$ is changed to $b'_n b'_{n-1} b'_{n-2} b'_{n-3} b_{n-4}\ b_{n-5}...b_1$. Repeating this procedure $n/4$ times on every other four bits of each vertex label, we can obtain relabeled Q_n^Z such that for any vertex s, P_b has a new address $b' = b'_n b'_{n-1}...b'_1$ and $s' = 00...0$. This completes the proof.

□

By algorithm *ADDR_TRANS*, we can convert many algorithms for special cases to algorithms for general cases. For example, if we have an algorithm to find a shortest path from P_0 to every other processor, this algorithm can be generalized to an algorithm for finding the shortest path from any processor P_b to every other processor.

7.3. The Shortest Path

The shortest path between two processors in Z-cube can be obtained by using a table. Each processor P_b maintains a table $T1_b$, with rows of the number of the destinations. Each row contains the shortest path to these destination. If the destination is adjacent to a source, a row has the destination itself. Otherwise, a row contains a list of intermediate processors on the shortest path to the given destination. We now consider the shortest path table between P_0 and P_b in the 4-dimensional Z-cube Q_4^Z.

Table-5: Shortest path table for P_0 in Q_4^Z

destination	1^{st} move	2^{nd} move	3^{rd} move	4^{th} move
0001	0001			
0010	0010			
0011	0010	0011		
0100	0001	0101	0100	
0101	0001	0101		
0110	0010	0110		
0111	0010	0110	0111	
1000	1000			
1001	1000	1001		
1010	1000	1010		
1011	1000	1010	1011	
1100	1000	1001	1101	1100
1101	1000	1001	1101	
1110	1000	1010	1110	
1111	1000	1010	1110	1111

Obviously, the processor P_0 need to maintain a routing table for fifteen destination processors. In addition, each processor P_b can keep the same routing table as in the processor P_0, since we have the algorithm *ADDR_TRANS* to transform Q_n^Z to another Q_n^Z such that P_b has a new address, 0000. It implies that constant space is necessary for a shortest path table in each processor.

The table for the shortest paths from P_0 to any other processors in Q_4^Z is given in table-5. By this table and the algorithm *ADDR_TRANS*, we can compute the shortest path between any two processors in Q_4^Z in O(1) time.

Theorem 7.2: The shortest path between two processors, P_u and P_v, in Q_n^Z can be computed in O(n) time.

Proof: First, consider the shortest path from P_u to $P_{u'}$ such that $u'_n u'_{n-1} u'_{n-2} u'_{n-3} = v_n v_{n-1} v_{n-2} v_{n-3}$ and $u'_{n-4} u'_{n-5} \ldots u'_1 = u_{n-4} u_{n-5} \ldots u_1$. This shortest path can be obtained by table 5 and the algorithm *ADDR_TRANS* in O(1) time. Obviously, $P_{u'}$ and P_v are in the same (n-4)-dimensional subcube ζ_{n-4}. Now this problem is reduced to finding the shortest path between two processors, $P_{u'}$ and P_v in ζ_{n-4}. By applying the same method repeatedly $n/4$ times, we can find the shortest path from P_u to P_v. This completes the proof.

□

7.4. One-to-One Routing and Broadcasting

We use the shortest path table to construct a parallel one-to-one routing algorithm. In this algorithm, we employ a procedure *ADDR_TRANS* 1(u,v). This procedure is similar to *ADDR_TRANS*. The only difference is that *ADDR_TRANS* 1 relabels for both vertex u and v, whereas *ADDR_TRANS* only relabels for one vertex. After relabeling is performed, every processor knows the new address u' and v' for u and v respectively.

```
procedure ADDR_TRANS1(u ,v )
    for all processor P_b do in parallel
        b' = b ⊕ u;
        v' = v ⊕ u;
        case b_2 b_1 of
            01,10: if b'_2 b'_1 ≠ 01 and b'_2 b'_1 ≠ 10 then
                        b'_4 b'_3 := b'_3 b'_4;
            00,11: if b'_2 b'_1 ≠ 00 and b'_2 b'_1 ≠ 11 then
                        b'_4 b'_3 := b'_3 b'_4;
        endcase
        case v_2 v_1 of
            01,10: if v'_2 v'_1 ≠ 01 and v'_2 v'_1 ≠ 10 then
                        v'_4 v'_3 := v'_3 v'_4;
            00,11: if v'_2 v'_1 ≠ 00 and v'_2 v'_1 ≠ 11 then
                        v'_4 v'_3 := v'_3 v'_4;
        endcase
    endfor
end ADDR_TRANS1
```

By combining the table 7.1 and the algorithm *ADDR_TRANS* 1, we have the following one-to-one data routing algorithm which send message from P_u to P_v.

```
Algorithm ROUTE3
    for m := n downto 4 by 4 do
        ADDR_TRANS1(u_m u_{m-1} u_{m-2} u_{m-3}, v_m v_{m-1} v_{m-2} v_{m-3});
        for all processor P_b do in parallel
            if b'_m b'_{m-1} b'_{m-2} b'_{m-3} = 0000 then
                Send a message from P_b to P_c such that b_n b_{n-1}...b_{m+1}=
                c_n c_{n-1}...c_{m+1}  and  c'_m c_{m-1} c'_{m-2} c'_{m-3}=  v'_m v'_{m-1}
                v'_{m-2} v'_{m-3}  along the path found in the table entry
                T1_u[v'_m v'_{m-1} v'_{m-2}, v'_{m-3}] ;
        endfor
    endfor
end ROUTE3
```

The algorithm *ROUTE3* finds the shortest path by the method discussed in the previous section and send a message along the shortest path. Obviously, we now have the following theorem.

Theorem 7.3: Any processor in Q_n^Z can send a message to any other processor in $O(n)$ time.

□

By the same reason as with a shortest path problem, we use a broadcasting table to broadcast a message from a processor to all other processors in Q_4^Z. Every processor P_b has the following table $T2_b$. The data broadcasting paths by this table have been depicted as the spanning tree shown in figure 15.

Table-6: Broadcasting table for P_0 in Q_4^Z

address	step	set S
0000	0	{0001,1000}
0001	1	{0011,0101}
0010	∞	∅
0011	2	{0010}
0100	3	{0110}
0101	2	{0100,0111}
0110	∞	∅
0111	∞	∅
1000	1	{1001,1010}
1001	2	{1011,1101}
1010	2	{1110}
1011	∞	∅
1100	∞	∅
1101	3	{1111}
1110	3	{1100}
1111	∞	∅

As shown in figure 15, P_0 send its message to all other processors of Q_4^Z in O(1) time by accessing the table T2. The following algorithm broadcasts a message from any processor P_u to all other processors.

Algorithm *BROAD*
  for $m := n$ **downto** 4 **by** 4 **do**
    ADDR_TRANS($u_m u_{m-1} u_{m-2} u_{m-3}$);
    for all processor $P_{b'}$ **do in parallel**
      for i :=0 to 3 **do**
        if $i = T2_{b'}$.step[$b'_m b'_{m-1} b'_{m-2} b'_{m-3}$] **and** $u_k = b'_k$ for $k \leq 4(m-1)$ **then**
          send a message to set of processors, S={$P_{b''}$| $b''_m b''_{m-1} b''_{m-2} b''_{m-3} \in T2_{b'}$.set[$b'_m b'_{m-1} b'_{m-2} b'_{m-3}$] and $u_k = b'_k$ for $k \leq 4(m-1)$} ;
      endfor
    endfor
  endfor
end *BROAD*

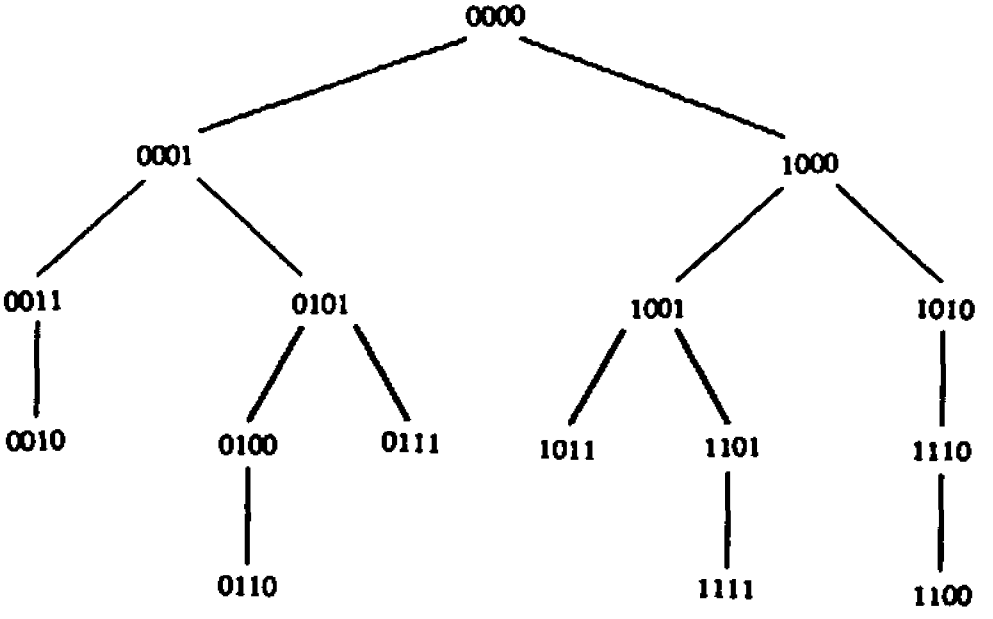


Figure 15: Broadcasting path in Q_4^Z

Theorem 7.4: Any processor P_b of Q_n^Z can broadcasts its message to all other processors in O(n) time.

Proof: First, consider a broadcasting problem to send a message from P_u to every $P_{u'}$ such that $u'_n u'_{n-1} u'_{n-2} u'_{n-3} \neq u_n u_{n-1} u_{n-2} u_{n-3}$ and $u'_{n-4} u'_{n-5} \ldots u'_1 = u_{n-4} u_{n-5} \ldots u_1$. By lemma 7.2 and using the broadcasting table and the algorithm *ADDR_TRANS*, we know this broadcasting can be achieved in O(1) time. After this, every processor $P_{b'}$, such that $b'_{n-4} b'_{n-5} \ldots b'_1 = 00\ldots0$, has a message to be sent. Next, these processors send a message inside a subcube ζ_{n-4} by the same method. Obviously, the algorithm *BROAD* broadcast a message from a processor P_b to all other processors after applying the same method repeatedly $n/4$ times. This completes the proof.

□

7.5. Summary

In this chapter, we consider a new interconnection network called the Z-cube. Since it has lower vertex degree than the conventional hypercube, it is more feasible to be implemented in hardware. We investigate some important topological properties of the Z-cube and have proved that Z-cube are vertex symmetric. By using tables, the simple algorithms for the shortest path and broadcasting have been presented.

Chapter 8

Concluding Remarks

The hypercube is the most widely adopted network structure for multicomputer systems, due to its many attractive topological properties. These properties include high connectivity, small diameter, high degree of symmetry, relatively small degree, high fault-tolerance and simple network routing algorithms. Since the first hypercube machine built at Caltech, many hypercube machines have been made available commercially. Recently, several modified hypercube network structures have been introduced to further improve the performance of conventional hypercube. Although previous, preliminary investigations on these modified hypercubes indicate that these networks may have potential to be alternatives for the conventional hypercube, the question of whether or not these modified hypercubes are indeed superior needs to be answered.

In this dissertation, we attempt to investigate two modified hypercube structures: the X-hypercube and the Z-cube. The X-hypercube is one of the variants of the hypercube that has a diameter almost the half of the diameter of the conventional hypercube without introducing any additional edges. This implies that the data communication in a X-hypercube may be more efficient than that in a hypercube. To verify the data communication performances of a X-hypercube, Q_n^X, we first give a new definition of Q_n^X. This definition is important, since it allow us to determine the connectivity of Q_n^X by examining the binary vertex labels this property. Indeed, the use of this new definition becomes the foundation of our investigation of the combinatorial

and computational aspects of Q_n^X. We show that Q_n^X can be embedded into Q_n with dilation 2 and congestion 2, and vice versa. This means that Q_n^X can be used to simulate Q_n efficiently.

It is a well known fact that communication cost dominates computation cost in many parallel algorithms whose performance depend heavily on the interconnection structures of a multicomputer system. For example, the execution time of a data communication instruction can be several orders greater than the execution time of an arithmetic or logical computation instruction. Since the main advantage of Q_n^X may be in the communication aspect, we focus our investigation on developing efficient parallel data communication algorithms. We present two algorithms for finding a shortest path between any two vertices in Q_n^X. Although both of them are in sequential form, they can be easily implemented as parallel one-to-one routing algorithms. We also present several parallel algorithms for data broadcasting and census functions. Using one of our data broadcasting algorithms as a subalgorithm, we also develop a simple parallel one-to-one routing algorithm, *ROUTE* 1. For many-to-many parallel data communication, we present a routing algorithm, *ROUTE* 2, that can complete the routing in $O(n)$ time if no conflict occurs. We show that all of one-to-one, many-to-one, and one-to many data communications in Q_n^X can be carried out in $\lceil (n+1)/2 \rceil$ communication steps, which is about two times more efficient than the corresponding hypercube operations.

We realize that the diameter of Q_n^X is $\lceil (n+1)/2 \rceil$ and indicates the lower bound for data communication performance in the worst case. Thus, we also analyze the average case data communication performance of Q_n^X. One parameter characterizing this

performance is the average distance between vertices in Q_n^X. We show that the average vertex distance of Q_n^X is about 13/16 of the average vertex distance of Q_n. This indicates that the average data communication performance of Q_n^X can be better than that of Q_n. In summary, our results on data communication in a X-hypercube show that data communication performance of Q_n^X is better than Q_n.

Comparing Q_n^X with Q_n by only considering data communication is not sufficient. We know that there are many problems that can be solved in parallel on hypercube machines. We also know that Q_n^X can be used to simulate Q_n efficiently by using an embedding method. However, such a method can cause considerable overhead due to the dilation and congestion in the embedding. Since Q_n^X is also recursively defined, we believe that most hypercube algorithms can be slightly modified to fit X-hypercube machines. Based on this belief, we consider several basic problems, such as prefix sum, packing, and sorting. We give SIMD X-hypercube algorithms for these problems and show that each of these algorithms takes the same number of communication steps and has the same computational complexity as its counterpart on the conventional hypercube machine. All of these algorithms are not based on embedding, and consequently the communication overhead of the embedding method are avoided. Our experience lead us to believe that parallel computation on a Q_n^X can be no less efficient than that on a Q_n.

Our results on a X-hypercube show that the X-hypercube is a good alternative for a conventional hypercube, especially when data communication is of major concern. However, the X-hypercube have some disadvantages when compared with a hypercube. First, the structure of Q_n^X is less regular than the structure of Q_n, and,

consequently, is hard to use. Second, the computational part of a data communication algorithm on Q_n^X can be less efficient than the computational part of the corresponding algorithm on Q_n. Our solution to this problem is to construct a set of fundamental parallel programs that can be used as building blocks. Since computation is generally less costly than communication, it is worthwhile to have communication algorithms with a little more computation time but less communication operations.

From the VLSI implementation point of view, the hypercube has some disadvantages due to its relatively high vertex degree. When the vertex degree of a network is high, its hardware implementation becomes difficult. This is because that the processor design becomes more complicated. Moreover, the VLSI layout of the network requires more area, as the number of data links per processor increases.

The Z-cube has been designed for easy VLSI implementation. The n-dimensional Z-cube, Q_n^Z, and the n-dimensional hypercube, Q_n, have the same number of vertices and the same diameter. However, Q_n^Z contains only 75 percent of the edges of Q_n, and each vertex of Q_n^Z has degree $(3/4)n$. Other attractive properties of Q_n^Z include the fact that Q_n^Z is a subgraph of Q_n, and Q_n^Z can simulate Q_n efficiently. In this dissertation, we investigate the communication aspect of Q_n^Z. We give simple algorithms for finding the shortest paths and data broadcasting. We believe that the Z-cube is another good alternatives for the conventional hypercube.

Many problems regarding parallel computations on the X-hypercube and the Z-cube remain open. We know that a hypercube can be used to simulate many useful networks by graph embedding. For example, a complete tree T_n of 2^n-1 vertices can be embedded into Q_n with dilation 2 and congestion 1. Since Q_n can be embedded

into Q_n^X with dilation 2 and congestion 2, T_n can be embedded into Q_n^X with dilation 4 and congestion 2. However, we found that for small n, T_n can be embedded into Q_n^X with dilation 1 and congestion 1, i.e., T_n is a subgraph of Q_n^X. It still remains open whether or not T_n is a subgraph of Q_n^X for all n. Similarly, it is not known yet what the best embeddings of many other useful networks into Q_n^X are. In chapter 6, we give several parallel algorithms for Q_n^X. All these algorithms utilize the connectivity of Q_n^X effectively. It remains to be answered whether we can also design X-hypercube algorithms for many other applications so that their performance is identical to hypercube algorithms. Another important issue is the fault-tolerance of Q_n^X. This issue is not addressed in this dissertation.

Since the Z-cube has been proposed very recently, the investigation of its properties is still in a preliminary stage. Many more issues of the conventional hypercube that have been investigated should be considered both for the X-hypercube and the Z-cube. The final conclusion of whether or not a X-hypercube or a Z-cube are superior over the conventional hypercube in general should be generated only after further studies are conducted, even though our results indicate that the X-hypercube and the Z-cube are good alternative for the hypercube in several aspects.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA., 1974.

[AG81] J. R. Armstrong and F. G. Gray, "Fault Diagonosis in a Boolean n-Cube of Microprocessor", IEEE Transactions on Computers, Vol. C-30, No. 8, pp. 587-590, 1981.

[AGPV88] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish, "A Tradeoff Between Information and Communication in Broadcast Protocols", Proceedings of Workshop on VLSI Architecture and Algorithms, Greece, Springer-Verlag LNCS in Computer Science, pp. 369-379, June-july 1988.

[AJ75] G. A. Anderson and E. D. Jensen, "Computer Interconnection Structures: Taxanomy, Characteristics, and Examples", Computing Surveys, Vol. 7, pp.197-213, 1975.

[Akl89] S. G. Akl, The Design and Analysis of Parallel Algorithms, Prentice-Hall, Englewood Cliffs, NJ., 1989.

[AP89] S. Abraham and K. Padmanabhan, "An Analysis of the Twisted Cube Topology", 1989 International Conference on Parallel Processing. pp. I116-I120, 1989.

[BA84] L. N. Bhuyan and D. P. Agrawal, "Generalized Hypercube and Hyperbus Structure for a Computer Network", IEEE Transaction on Computers, Vol. c-33, No. 4, pp. 323-333, 1984.

[Bat68] K. E. Batcher, "Sorting Networks and Their Applications", Proceedings of the AFIPS Spring Joint Computer Conference, Vol. 32, pp. 307-314, 1968.

[BGLY81] A. Borodin, L. Guibas, N. Lynch, and A. Yao, "Efficient Searching Using Partial Ordering", Information Processing Letters, Vol. 12, pp. 71-75, 1981.

[BH90] F. Buckley and F. Harary, Distance in Graphs, Addison-Wesley, Reading, MA., 1990.

[BM79] J. A. Bondy and U. S. R. Murty, Graph Theory with Applications, North Holland, Inc., New York, NY., 1979.

[BMS88] S. Bettayeb, Z. Miller and I. Sudborough, "Embedding Grids into Hypercubes" , Proceedings of the 3rd Aegean Workshop on Computing(AWOC 88), Corfu, Greece, 1988.

[CR79] E. Chang and R. Roberts, "An Improved Algorithm for Decentralized Extrema-finding in Circular Configurations of Processors", Communications of the ACM, Vol. 22, No. 5, pp. 281-283, 1979.

[CY85] R. Cole and C. K. Yap, "A Parallel Median Algorithm", Information Processing Letters, Vol. 20, pp. 137-139, 1985.

[DK86] R. Dechter and L. Kleinrock, "Broadcast Communications and Distributed Algorithms", IEEE Transactions on Computers, Vol. c-35, No. 3, pp. 210-218, 1986.

[EBSS88] K. Efe, P. Blackwell, T. Shiau, and W. Slough, "A Reduced Diameter Interconnection Network", Department. of Computer Science, University of Missouri, Columbia, 1988.

[Fic83] F. Fich, "New Bounds for Parallel Prefix Circuits", Proceedings of the 15th Annual ACM Symposium on Theory of Computing, pp. 114-118, 1978.

[Fly72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, Vol C-21, No.9, pp. 948-960, 1972.

[FS80] R. A. Finkel and M. Solomon, "Processor Interconnection Strategy", IEEE Transactions on Computers, Vol. C-29, No. 5, pp. 360-371, 1980.

[GR84] D. B. Gannon and J. V. Rosendale, "On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms", IEEE Transactions on Computers, Vol. C-33, pp. 1180-94, 1984.

[Har72] F. Harary, Graph Theory, Addison-Wesley, Reading, MA., 1972.

[HB84] K. Hwang and F. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, New York, NY., 1984.

[HJ86] C. T. Ho and S. L. Johnson, "Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes", Proceedings of the 1986 IEEE International Conference on arallel Processing, pp. 640-

648, 1986.

[HKS87] P. A. Hilbers, M. R. J. Koopman, and J. L. A. van de Snepscheut. "The twisted cube", Parallel Architectures and Languages Europe. Lecture Notes in Computer Science, pp.152-159, 1987.

[HMR83] J. W. Hong, K. Mehlhorn, and A. L. Rosenberg., "Cost Trade-Offs in Graph Embeddings, with applications", Journal of ACM. Vol. 30, pp. 709-728, 1983.

[HS86] W. D. Hillis and G. L. Steele, Jr., "Data Parallel Algorithms", Communications of the ACM. Vol. 29, No. 12, 1986.

[Knu73] D. E. Knuth, The Art of Computer Programming, Vol.3 Sorting and Searching. Addison-Wesley, Reading, MA., 1973.

[Kun80] H. T. Kung, The Structure of Parallel Algorithms, in M.C.Yovits(Ed.), Advances in Computers. Vol. 19, New York, Academic Press, pp. 65-112, 1980.

[KZ90] K. H. Kwon, and S. Q. Zheng, "On The Connectivity of Twisted Hypercubes", Technical Report # 90-010, Department of Computer Science, Louisiana State University, Baton Rouge, LA., 1990.

[KZ91-1] K. H. Kwon, and S. Q. Zheng, "Average Distance Between Vertices in Twisted Hypercubes", Technical Report # 91-007, Department of Computer Science, Louisiana State University, Baton Rouge, LA., 1991.

[KZ91-2] K. H. Kwon, and S. Q. Zheng, "Efficient Data Communication Algorithms for Twisted Hypercubes", Technical Report # 91-026, Department of Computer Science, Louisiana State University, Baton Rouge, LA., 1991.

[KLZ91] K. H. Kwon, S. Latifi and S. Q. Zheng, "Average Performance of Data Communication in for Twisted Hypercubes", International Conference on Finite Fields Coding Theory, and Advances in Communications and Computing, Las Vegas, Nevada, August, 1991.

[LCT89] C. Liang, Y. Chen, and W. Tsai, "Embedding of Linear and Binary Tree in Cubical Ring Connected Cycles Networks", 1989 International Conference on Parallel Processing, 1989.

[LD90] S. Lakshmivarahan and S. K. Dhall, Analysis and Design of Parallel Algorithms, McGrow-Hill, New York, NY., 1990.

[LF80] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation", Journal of the ACM, Vol. 27, pp. 831-838, 1980.

[Lie85] A. L. Liestman, "Fault-tolerant Broadcast Graphs", Networks, Vol. 15, pp. 159-171. 1985.

[NS81] D. Nassimi and S. Sahni, "Data Broadcasting in SIMD Computers", IEEE Transactions on Computers, C-30, No. 2, pp. 101-106, 1981.

[NS82] D. Nassimi and S. Sahni, "Parallel Permutation and Sorting Algorithms and a New Generalized Connecton Network', Journal of the ACM., Vol. 29, No. 3, pp. 642-667, 1982.

[Pla89] C. G. Plaxton, "Efficicient Computation on Sparse Interconnection Networks", Ph.D Thesis, Department of Computer Science, Stanford University, Stanford, CA., 1989.

[Qui87] M. J. Quinn, Designing Efficient Algorithms for Parallel Computers, McGrow-Hill, New York, NY., 1987.

[Ray88] M. Raynal, Networks and Distributed Computation: Concepts, Tools and Techniques, The MIT Press, Cambridge, MA., 1988.

[RF87] D. A. Reed and R. M. Fujimoto, "Multicomputer Networks" Scientific Computation Series, pp. 29-79, The MIT Press, Cambridge, MA., 1987.

[SS85] Y. Saad and M. H. Schultz, "Discrete parallel methods for solving banded linear systems, Technical Report YALEU/DCS/RR-387, Computer Science Department, Yale University, New Haven, CT., 1985.

[SS86] Y. Saad and M. H. Schultz, "Topological property of Hypercubes" IEEE Transactions on Computers, Vol. 37, No. 7, pp. 867-871, 1988.

[Sun88] Y. Y. Sung, "X-hupercube: A better interconnection network", Proceedings of the 26th Annual Southeast Regional ACM Conf., pp. 557-561, 1988.

[SZ85] C. L. Seitz, "The Cosmic Cube", Communications of the ACM, Vol. 28, No. 1, pp. 22-33, 1985.

[Tan81] A. S. Tanenbaum, Computer Networks, Prentice-Hall, Englewood Cliffs, NJ., 1981.

[Ull84] J. D. Ullman, Computational Aspects of VLSI, Computer Science Press, Rockville, MD., 1984.

[YZ91] G. Young and S. Q. Zheng, "Average Distance Between Vertices in Multiply-Twisted Hypercubes", Presented at 22nd Southeastern International Conference on Combinatorics Graph Theory Computing. Baton Rouge, LA., February, 1991.

[Zhe89] S. Q. Zheng, "On Embedding Between Hypercubes and Twisted Hypercubes", Technical Report #89-001, Department of Computer Science, Louisiana State University, Baton Rouge, LA., 1989.

[Zhe90] S. Q. Zheng, "SIMD Data Communication Algorithm for Multiply-twisted Hypercubes", Technical Report #90-006, Dept. of Computer Science, Louisiana State University, Baton Rouge, LA., 1990.

[ZKC91] S. Q. Zheng, K. H. Kwon, and J. H. Chen, "Finding a Shortest Path in Twisted Hypercube", Presented at "The 22nd Southeastern International Conference on Combinatorics, Graph Theory and Computing", Baton Rouge, LA., Feb.,1991. Also to appear in Congressus Numerantium.

[ZL91] S. Q. Zheng, and S. Latifi, "Z-cube Interconnection Network" Technical Report #91-022, Department of Computer Science, Louisiana State University, Baton Rouge, LA., 1991.

Vita

Kyung Hee Kwon was born in Taegu, Korea, on January 10th, 1954. He received a B.S. degree in Physics from Korea University, Seoul, Korea, in February 1976. From October 1975 to April 1979, he worked for Foxboro Korea Company, Limited. In April 1979, he was employed by Korea Institute for Industrial Economy and Technology, Computer System Laboratory, and remained as a researcher until August 1984. While working for Korea Institute for Industrial Economy and Technology, he received his M.B.A. in Electronic Data Processing from Korea Unversity in September 1981. He was sent to Yale University on a Korean Government scholarship to study at the Department of Computer Science as a special student from August 1981 to February 1982. He entered the Department of Computer Science at Old Dominion University, Virginia in August 1984 and received M.S. degree in Computer Science in December 1986. In January 1987, he joined the Ph.D. program in the Department of Computer Science at Louisiana State University where he is currently a candidate for the degree of Ph.D.

His current interests are network design, parallel algorithms and architectures, parallel compiler and computational aspects of VLSI.

He married Sung Sook Kim in July 1979 and has two sons.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Kyung Hee Kwon

Major Field: Computer Sciece

Title of Dissertation: Parallel Computation on Hypercube-like Machines.

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination: 8/30/91