INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Embeddings, communication and performance of algorithms in faulty hypercubes

Wang, Jin-Kun, Ph.D.
The Ohio State University, 1990
EMBEDDINGS, COMMUNICATION AND PERFORMANCE OF ALGORITHMS IN FAULTY HYPERCUBES

A Dissertation
Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of the Ohio State University

by
Jin-Kun WANG, B.S.E.E, M.S.E.E.

* * * * *
The Ohio State University
1990

Dissertation Committee:
Professor Fusun Ozuguner
Professor C. A. Klein
Professor D. E. Orin

Approved by:
Department of Electrical Engineering
To my grandmother, parents, my wife Fonglan, daughter Maylee, and son Synkai
ACKNOWLEDGMENTS

I would like to thank my advisor Professor Füsun Ö zgüner for her guidance, advice, encouragement, and friendship. I would also like to express my appreciation to Professor C. A. Klein for his careful review of this work. Thanks also go to Professor D. E. Orin who reviewed this research. I am most grateful to my wife, Fonglan, for her patience, understanding, and confidence in me. Without her encouragement and support, I wouldn’t be able to pursue this doctoral degree. I am also grateful to my grandmother and parents for their constant support.

Thanks also go to many friends here. They give me a lot of support and friendship. Finally, I would like to acknowledge the fellowship which I received from the Chung Shan Institute of Science and Technology, Taiwan, Republic of China.
VITA

March 8, 1955 ....................... Born—Taoyuan, Taiwan, Rep. of China

1977 ................................. B.S.E.E.
National Cheng Kung University
Tainan, Taiwan, R.O.C.

1977–1981 .......................... Research Assistant
Chung Shan Ins. Sci. & Tech.
Taiwan, R.O.C.

1982–83 ............................... M.S.E.E.
The Ohio State University
Columbus, Ohio, U.S.A.

1983–1987 .......................... Assistant Researcher
Chung Shan Ins. Sci. & Tech.
Taiwan, R.O.C.

1988–1990 ............................ Graduate Student
The Ohio State University
Columbus, Ohio, U.S.A.
PUBLICATIONS

"Test Generation and Diagnosis for the OSU Skeletal Motion Processor," Master Thesis, The Ohio State University, Columbus, Ohio, U.S.A. 1983.


FIELDS OF STUDY

Major Field: Electrical Engineering
Studies in Computer Engineering

Minor Fields:
Studies in Control Systems
Studies in Computer and Information Science
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** iii 
**VITA** iv 
**LIST OF TABLES** x 
**LIST OF FIGURES** xi 

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Organization of Dissertation</td>
<td>3</td>
</tr>
<tr>
<td>II. BACKGROUND</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Hypercube</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Graph Embedding</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Gray Code and Linear Array Embedding</td>
<td>7</td>
</tr>
<tr>
<td>2.5 Mesh Embedding</td>
<td>9</td>
</tr>
<tr>
<td>2.6 Data Communication</td>
<td>10</td>
</tr>
</tbody>
</table>
6.2.1 The Parallel CG Algorithm in a Healthy Hypercube . . . 108
6.2.2 The Parallel CG Algorithm in a Faulty Hypercube . . . 110
6.3 Application to the Parallel Sorting Algorithm ................... 114
6.3.1 The Parallel Sorting Algorithm on a 2-D Mesh Embedded
in a Healthy Hypercube .................................................... 114
6.3.2 The Parallel Sorting Algorithm on a 2-D Mesh Embedded
in a Faulty Hypercube .................................................... 119
6.4 Summary .............................................................................. 120

VII. CONCLUSION AND FUTURE RESEARCH 121
7.1 Conclusion ............................................................................. 121
7.2 Research Extensions ............................................................. 124

LIST OF REFERENCES 126
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Prohibit links of an embedding example by Algorithm \textit{LNRCSQ}</td>
</tr>
<tr>
<td>2</td>
<td>Performance results of Algorithm \textit{MESHLATIN}</td>
</tr>
<tr>
<td>3</td>
<td>The possible pairs of passing 1 node</td>
</tr>
<tr>
<td>4</td>
<td>The possible pairs of passing 2 nodes</td>
</tr>
<tr>
<td>5</td>
<td>The possible pairs of passing 3 nodes</td>
</tr>
<tr>
<td>6</td>
<td>The possible pairs of passing 4 nodes</td>
</tr>
<tr>
<td>7</td>
<td>The possible pairs of passing 5 nodes</td>
</tr>
<tr>
<td>8</td>
<td>The possible pairs of passing 6 nodes</td>
</tr>
<tr>
<td>9</td>
<td>The possible pairs of passing 7 nodes</td>
</tr>
<tr>
<td>10</td>
<td>The possible pairs of passing 8 nodes</td>
</tr>
<tr>
<td>11</td>
<td>Comparison of performance</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>FIGURES</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A 4-dimensional hypercube</td>
</tr>
<tr>
<td>2</td>
<td>A K-map illustration of embedding a 4 × 8 mesh in a 5-cube</td>
</tr>
<tr>
<td>3</td>
<td>Limitation of Algorithm LNRCUBE</td>
</tr>
<tr>
<td>4</td>
<td>A linear array of length 5 embedded in a faulty 3-cube using Algorithm LNRLATIN (a) A faulty hypercube (b) The adjacency list</td>
</tr>
<tr>
<td>5</td>
<td>A faulty 3-cube with faulty elements F₁ = 001, F₂ = 010 and L₁ = 1z1</td>
</tr>
<tr>
<td>6</td>
<td>A linear array embedded in a faulty 3-cube using the Direct-Connect Capability approach (a) The faulty hypercube (b) An initial embedding (c) The final embedding</td>
</tr>
<tr>
<td>7</td>
<td>Limitation of Algorithm MESH_CUBE</td>
</tr>
<tr>
<td>8</td>
<td>A faulty hypercube</td>
</tr>
<tr>
<td>9</td>
<td>Possible K-maps of the mesh embedding example</td>
</tr>
<tr>
<td>10</td>
<td>An example of embedding a 3 × 3 mesh</td>
</tr>
<tr>
<td>11</td>
<td>Illustration of a 3 × 3 mesh embedding using Algorithm MESH_LATIN</td>
</tr>
<tr>
<td>12</td>
<td>Some meshes found by Algorithm MESH_LATIN</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 13     | A $5 \times 5$ mesh embedded in a 6-cube using DCC approach  
(a) An initial embedding  
(b) The final embedding |
| 14     | An example of embedding a linear array using the Coordinate Sequence |
| 15     | An example of embedding a 2-D mesh in a faulty hypercube using Algorithm MESHDCC |
| 16     | Global Sum operation in a healthy 4-cube |
| 17     | An example of splitting operation |
| 18     | An example showing Global Sum operation in a faulty hypercube |
| 19     | An example showing Global Broadcast operation in a faulty hypercube |
| 20     | An example showing Global Sum and Global Broadcast with partial Exchange-Add operation in a faulty hypercube |
| 21     | Performance degradation of the parallel CG algorithm using GS-GB in a faulty hypercube |
| 22     | Performance degradation of the parallel CG algorithm using GS-GB with partial EA in a faulty hypercube |
| 23     | Merge sorting steps in the $M(4,4)$ sorting algorithm |
CHAPTER I

INTRODUCTION

1.1 Introduction

In the last years, many parallel algorithms have been developed and implemented on hypercube multiprocessors [1,2,3,4]. Hypercube multiprocessors provide a topology which is extremely regular and scalable for implementing parallel algorithms [5,6]. One of the attractive topological properties of the hypercube is that many other topologies such as a linear array, a ring, a tree, and a mesh can be embedded in the hypercube [5,7].

A study of hypercube algorithms reveals that topologies such as linear arrays and 2-D meshes are embedded in a hypercube for the computations of many parallel algorithms that require local communication while the hypercube topology is used for global communication. In [8], the linear array topology is embedded in a hypercube for performing local communication; the hypercube topology is used to perform the inner product computation and the matrix-vector multiplication in the preconditioned Conjugate Gradient method. Similarly, the parallel coarse grain Conjugate Gradient algorithm presented in [9] embeds a linear array in a hypercube to perform the local communication and uses the hypercube topology to perform the global communication. Algorithms have been developed by many researchers to embed different topologies such as linear arrays, rings, trees and meshes [5,10,11,12,13,14,15,16,17,18] and to perform data communication in a hy-
percube [19,20,21,22,23]. However, they cannot be applied to a faulty hypercube.

A faulty hypercube consists of faulty nodes and/or faulty links. The set of faulty nodes and/or links can be identified by the fault-free processors running a distributed fault-diagnosis algorithm when the number of faulty elements \( r \leq d \) [24], where \( d \) is the dimension of hypercube. In this research, it is assumed that the faulty elements are given; therefore, diagnosis of faulty elements in a hypercube will not be considered.

Most algorithms proposed for reconfiguration in a faulty hypercube find hypercubes of lower dimension and reduce the utilization by at least a half, unless hardware redundancy is incorporated [25]. In order to improve the utilization of a faulty hypercube, it is crucial to have efficient embedding and data communication algorithms for faulty hypercubes to achieve parallel processing.

The objective of this research is to develop algorithms for embedding linear arrays or 2-D meshes and performing global data communication in a faulty hypercube. The embedding algorithms developed can be applied to a faulty hypercube with any number of faulty elements. Global data communication operations developed consist of the \textit{Global Sum (GS)} operation, the \textit{Global Broadcast (GB)} operation, and \textit{GS-GB with partial Exchange-Add} [9] in a faulty hypercube. In addition, the embedding and global data communication schemes developed are applied to the parallel Conjugate Gradient algorithm and a parallel sorting algorithm. However, any parallel algorithm which requires 1-dimensional or 2-dimensional local communication and global communication operations such as \textit{Global Sum}, \textit{Exchange-Merge} [26], and \textit{Global Broadcast} can be implemented in a faulty hypercube by applying the embedding and data communication schemes developed. In addition, the schemes are also applicable to incomplete hypercubes resulting from the allocation of subcubes to different users.
1.2 Organization of Dissertation

Chapter II presents basic concepts and previous work. In Chapter III, algorithms for embedding a linear array in a faulty hypercube are described. They are: Algorithm LNRCUBE, Algorithm LNRLATIN, Algorithm PLNRLATIN, Algorithm LNRCSQ, Algorithm PLNRCSQ, and Algorithm LNRDCC. Algorithm LNRCUBE finds a maximum fault-free subcube in a faulty hypercube first, then embeds a linear array in it. Algorithm LNRLATIN uses a variation of Latin Multiplication [27] to find a linear array in a Gray code ordering. Algorithm PLNRLATIN is a parallel version of Algorithm LNRLATIN. Algorithm LNRCSQ embeds a linear array in a faulty hypercube using a variation of the Coordinate sequence approach [28]. Algorithm PLNRCSQ is a parallel version of Algorithm LNRCSQ. Algorithm LNRDCC embeds a linear array in a faulty hypercube that has Direct-Connect Capability.

In Chapter IV, algorithms for embedding an $m_1 \times m_2$ mesh in a faulty hypercube are presented. They are: Algorithm MESHQUIRE, Algorithm MESHLATIN, Algorithm PMESHLATIN1, Algorithm PMESHLATIN2, Algorithm MESHDCC, and Algorithm PMESHDCC. Algorithm MESHLATIN deletes faulty columns and/or rows in a $2^{d_1} \times 2^{d_2}$ mesh, where $d_1 \geq \lceil \log_2 m_1 \rceil$, $d_2 \geq \lceil \log_2 m_2 \rceil$ and $(d_1 + d_2) \leq d$, then pastes the healthy columns and rows in a Gray code ordering. Algorithm PMESHLATIN1 and Algorithm PMESHLATIN2 are two different parallel versions of Algorithm MESHLATIN. Algorithm MESHDCC removes faulty columns and/or rows in a $2^{d_1} \times 2^{d_2}$ mesh and forms the desired mesh from the healthy columns and rows by connecting through the Direct-Connect Hardware which is assumed to be fault-free. Algorithm PMESHDCC is a parallel version of Algorithm MESHDCC.
Algorithms developed for data communication in a faulty hypercube are described in Chapter V. These algorithms improve the utilization of a faulty hypercube and provide an efficient way to perform different types of data communication in a faulty hypercube. The Global Sum algorithm collects the partial sum in each node to a collecting node in a faulty hypercube. The Global Broadcast algorithm distributes data from one node to all the other healthy nodes in a faulty hypercube. The Global Sum and Global Broadcast with partial Exchange-Add operation in a faulty hypercube performs the accumulation of the partial sum in each node and the distribution of the sum accumulated to all nodes.

In Chapter VI, the embedding and data communication schemes developed are applied to algorithms. Both the time performance model and the performance degradation of the algorithms in faulty hypercubes are evaluated.

Finally, Chapter VII provides the summary and conclusion of this research. Some research extensions are suggested.
CHAPTER II
BACKGROUND

2.1 Introduction

Basic concepts and previous work on linear array and 2-D mesh embedding algorithms, and data communication algorithms are surveyed in this Chapter. The definition and topological properties of a hypercube are presented in Section 2.2. Graph embedding is described in Section 2.3. The techniques for embedding linear arrays and 2-D meshes in a hypercube are surveyed in Section 2.4 and Section 2.5. Note that these techniques can only be applied to a healthy hypercube and cannot be applied to a hypercube which contains faulty nodes and/or faulty links [29]. In addition, data communication techniques such as Global Sum, Global Broadcast, Exchange-Add and different schemes for communication in incomplete hypercube are surveyed in Section 2.6.

2.2 Hypercube

A $d$-dimensional hypercube ($d$-cube) consists of $2^d$ processors (nodes) labeled from 0 to $2^d-1$. Each node is represented by a $d$-tuple $(b_{d-1} \cdots b_i \cdots b_0)$, where $b_i \in \{0, 1\}$ denotes a specific coordinate in the hypercube topology [6,30]. Each node in the hypercube is linked to exactly $d$ neighbors whose binary representations differ by one and only one bit [5]. Fig. 1 shows a 4-dimensional hypercube. Several commercial hypercube multiprocessors such as Intel's iPSC, Ametek's Hypernet,
\textit{Ncube}'s machine and FPS's T-Series [31,32] are available.

A subcube in a hypercube can be represented uniquely by a \textit{d-tuple}\{0,1,x\}^d. Coordinates with values "0" and "1" will be described as \textit{bound} coordinates and those with value "x" as \textit{free} coordinates. By assigning all the combinations of "0" and "1" to the \textit{free} coordinates, all the nodes of a specific subcube are identified. For instance, subcube 10xx includes the nodes of 1000, 1001, 1011 and 1010, where $b_3$ and $b_2$ are the \textit{bound} coordinates; $b_1$ and $b_0$ are the \textit{free} coordinates. A (d-k)-dimensional subcube in a d-cube is represented by a \textit{d-tuple} with $k$ \textit{bound} coordinates and $d-k$ \textit{free} coordinates [33].

A link in a hypercube is uniquely specified by a \textit{d-tuple} \{0,1,z\}^d, which contains exactly one "z" at a particular coordinate position. Two unique processors connected via a link can be obtained by assigning "0" and "1" to the "z" coordinate. For example, link 10z1 connects node 1001 and node 1011 through the coordinate $b_1$ [33].

Some important topological properties of hypercubes are summarized as follows [5]: (1) There are $d$ different ways to tear a $d$-cube into two ($d-1$)-subcubes. (2) There are $d! \, 2^d$ different ways to label the $2^d$ nodes for conformation with hy-
percube definition. (3) Any two adjacent nodes $A$ and $B$ of a $d$-cube are such that the nodes adjacent to $A$ and those adjacent to $B$ are connected in a one-to-one fashion. (4) There are no cycles of odd length in a $d$-cube. (5) A $d$-cube is a connected graph of diameter $d$. (6) The minimum distance between two nodes $A$ and $B$ is the Hamming distance $H(A,B)$, that is the number of different bits between $A$ and $B$. (7) If $H(A,B) < d$, there are $H(A,B)$ parallel paths of length $H(A,B)$ between nodes $A$ and $B$. (8) If $H(A,B) < d$, there are $d$ parallel paths between node $A$ and $B$, each length is at most $H(A,B) + 2$. (9) A ring of length $l_r$ can be mapped into the $d$-cube when $l_r$ is even and $4 \leq l_r \leq 2^d$.

2.3 Graph Embedding

An embedding of the graph $G(V_G, E_G)$ in the graph $H(V_H, E_H)$ is to map each vertex in $G$ to a unique vertex in $H$ [16,34], where $V_G$ and $V_H$ denote the sets of nodes in $G$ and $H$ respectively, $E_G$ and $E_H$ denote the sets of edges in $G$ and $H$ respectively. There are two measures of the cost of a graph embedding [7,35]. One is the dilation cost, which is the maximum distance between adjacent nodes of $G$ in $H$. The other is the expansion cost, which is the ratio of the number of nodes of $H$ to the number of nodes of $G$. In this research, the unit dilation cost is considered in all algorithms except those that use the Direct-Connect Capability approach. In [36], it is proved that hypercube embedding is NP-complete.

2.4 Gray Code and Linear Array Embedding

Embedding a linear array of length $l_m$ in a hypercube is equivalent to finding a group of $l_m$ nodes connected in a Gray code ordering. There are two approaches of finding a Gray code. One is the Binary Reflected Gray Code; the other is the Coordinate Sequence approach.
The Binary Reflected Gray Code can be generated recursively as follows [5]: Let $G_{d-1}$ be the $(d - 1)$-bit Gray code, $G^R_{d-1}$ denote the codewords in $G_{d-1}$ in reverse order, and $0G_{d-1}$ or $1G_{d-1}$ be the Gray code obtained by prefixing a 0 or a 1 to each element of $G_{d-1}$. Then the $d$-bit Gray code can be expressed as: $G_d = \{0G_{d-1}, 1G^R_{d-1}\}$, where $d = 2, 3, \ldots$ and $G_1 = \{0, 1\}$. Each element of $G_d$ indicates a node in a $d$-cube. For instance, $G_4$ is generated as below:

$G_2 = \{0G_1, 1G^R_1\} = \{00, 01, 11, 10\}$, since $G_1 = \{0, 1\}$. Then we have $G_3 = \{0G_2, 1G^R_2\} = \{000, 001, 011, 010, 110, 111, 101, 100\}$. Finally, $G_4 = \{0G_3, 1G^R_3\} = \{0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000\}$. To embed a linear array of length $l_m$ in a hypercube is to extract $l_m$ consecutive elements from the $G_d$ obtained, where $d \geq \lceil \log_2 l_m \rceil$. For example, a linear array of length 7 in a hypercube can be embedded by extracting 7 consecutive elements from any $G_d$, where $d \geq 3$. If $d=3$, then a possible linear array embedded is 000-001-011-010-110-111-101.

The Coordinate sequence approach [28] defines a $d$-bit Gray code by a sequence of coordinates where the $i$-th element ($i = 1, \ldots, 2^d - 1$) of the sequence $S_d$ corresponds to the bit or the coordinate that is complemented to obtain codeword $i$ from codeword $(i - 1)$, with codewords numbered from 0 to $2^d - 1$. The most significant coordinate is labeled as $(d - 1)$ and the least significant coordinate labeled as 0. For example, starting with 00 the sequence of coordinate transitions $\{0, 1, 0\}$ results in the Gray code $\{00, 01, 11, 10\}$. $S_d$ can be constructed as: $\{S_{d-1}, d - 1, S_{d-1}\}$, where $d = 2, 3, \ldots$, and $S_1 = \{0\}$. A $d$-bit Gray code can be generated by selecting a start node, then complementing the start node on the bit position indicated by the first element in $S_d$ to get the second node, then complementing the second node on the bit position indicated by the second element in $S_d$ to get the third node, and so on. For example, a start node labeled as 00 with the sequence of co-
ordinate transitions \{0, 1, 0\} represents a Gray code \{00, 01, 11, 10\}. Thus, a linear array of length \(l_m\) embedded in a hypercube consists of a start node followed by \((l_m - 1)\) nodes determined uniquely by any \((l_m - 1)\) consecutive elements of \(S_d\), where \(d \geq \lceil \log_2 l_m \rceil\). For example, a linear array of length 7 in a hypercube can be embedded by assigning node 110 as the start node and extracting 6 consecutive elements of \(S_3 = \{0, 1, 0, 2, 0, 1, 0\}\) for determining the rest of the nodes. Let the 6 consecutive elements be \{1, 0, 2, 0, 1, 0\}, then a possible linear array embedded can be obtained as 110-100-101-001-000-010-011.

Finally, it is clear that the smallest hypercube which can embed a linear array of length \(l_m\) has the dimension \(d\), where \(d \geq \lceil \log_2 l_m \rceil\).

### 2.5 Mesh Embedding

Algorithms for mesh embedding in a healthy hypercube have been developed by several researchers [5,10,11,12]. An \(m_1 \times m_2\) mesh can be embedded in a \(d\)-dimensional hypercube by finding two Gray code sequences \(L_{m_1}\) and \(L_{m_2}\) for the \(m_1\) and \(m_2\) sides of the mesh respectively, and forming the required mesh by the Cartesian product \(L_{m_1} \times L_{m_2}\) [5,37]. In general, a Gray code sequence represents a linear array. Hence, \(L_{m_1}\) and \(L_{m_2}\) can be expressed as follows:

\[
L_{m_1} = \{g_1, g_2, \ldots, g_i, \ldots, g_{m_1}\}
\]

\[
L_{m_2} = \{h_1, h_2, \ldots, h_i, \ldots, h_{m_2}\}
\]

where \(g_i\) or \(h_i\) denotes a node represented by a \([\log_2 m_1]\)-tuple or a \([\log_2 m_2]\)-tuple respectively. The coordinates of the \([\log_2 m_1]\)-tuple and the coordinates of the \([\log_2 m_2]\)-tuple are selected among the coordinates of \(d\)-tuple exclusively. An example of a \(4 \times 8\) mesh embedded in a 5-dimensional hypercube is shown in Fig. 2.
The dimension of the smallest hypercube that can embed an $m_1 \times m_2$ mesh is \( \lceil \log_2 m_1 \rceil + \lceil \log_2 m_2 \rceil \) \cite{11}. This implies that embedding an $m_1 \times m_2$ mesh in a $d$-cube must be impossible if $d < ( \lceil \log_2 m_1 \rceil + \lceil \log_2 m_2 \rceil )$. This will be used initially in our algorithms to check if the mesh embedding is possible.

In \cite{5}, the following result is presented. Any $m_1 \times m_2 \cdots \times m_n$ mesh in the $n$-dimensional space $R^n$, where $m_i = 2^{d_i}$ can be mapped onto a $d$-cube where $d = d_1 + d_2 + \cdots + d_n$. The numbering of the grid points is any numbering such that each $d_i$ is in a Gray Code sequence provided that $d_i = \lceil \log_2 m_i \rceil$.

\[
\begin{array}{cccccccccc}
000 & 001 & 011 & 010 & 110 & 111 & 101 & 100 \\
\hline
00 & p & p & p & p & p & p & p \\
01 & p & p & p & p & p & p & p \\
11 & p & p & p & p & p & p & p \\
10 & p & p & p & p & p & p & p \\
\end{array}
\]

Figure 2: A K-map illustration of embedding a $4 \times 8$ mesh in a 5-cube

\subsection{2.6 Data Communication}

Data communication in a hypercube has been studied by many researchers \cite{19,21,22,23,38,39}. A standard algorithm for routing messages between arbitrary pairs of nodes in a healthy hypercube is described in \cite{38}. It is widely referred to as the $e$-cube algorithm and determines a shortest path. Four different communication schemes are presented in \cite{15}: (1) One to all broadcasting scheme which distributes common data from a single source to all other nodes. (2) One to all personalized communication scheme which sends a unique data from a single to
all other nodes. No replication of information takes place during distribution. (3) All to all broadcasting scheme which distributes common data from each node to all other nodes. (4) All to all personalized communication scheme which enables each node to send a unique piece of information to every other node. In addition, three communication graphs are also presented to represent different types of communication. They are: *n edge-disjoint spanning binomial trees*, *balanced spanning trees*, *n rotated spanning binomial trees*. In [19], several data communication algorithms including node-to-node, broadcast, total exchange algorithms are developed for a healthy hypercube and applicable to many parallel algorithms such as Gaussian Elimination, Conjugate Gradient methods, and the N-body problem. The problem of routing in faulty hypercubes has been addressed by several researchers [21,22,39]. In [39], routing and broadcasting algorithms are proposed for incomplete hypercubes with certain missing nodes provided that the hypercube network is fault-free. The routing algorithm sends a message from the source to the destination by checking the existence of links. A message is sent as follows: Let $i$ be the bit number of the first 1 in $\text{reladd}$ ($\text{reladd} = \text{source} \oplus \text{destination}$). If link $i$ exists from node source, then send the message on link $i$. The broadcast algorithm uses a travel array to forward a broadcast message in an incomplete hypercube. It does not provide the *Global Sum* operation which is required in many applications. In [21], the concepts of *faulty*, *non-faulty*, and *unsafe* are applied to routing and broadcasting in faulty hypercubes. The algorithm proposed avoids going through the unsafe subcubes according to the fault status of all neighboring nodes within a specified radius $k$ and finds the paths to forward the message. A routing method developed in [22] combines the concepts of *local information* and *randomization* to ensure high probability of successful data communication.
CHAPTER III
ALGORITHMS FOR EMBEDDING LINEAR ARRAYS

3.1 Introduction

A linear array is a very widely used topology in parallel implementations which require a 1-dimensional local communication. Algorithms for embedding a linear array in a healthy hypercube [10,11,40] cannot be applied to a faulty hypercube which contains faulty nodes and/or links [29]. Therefore, algorithms for embedding a linear array in a faulty hypercube are proposed and given in Algorithm LNRCUBE, Algorithm LNRLATIN, Algorithm PLNRLATIN, Algorithm LNRCSQ, Algorithm PLNRCSQ and Algorithm LNRDCC. First, the necessary condition for embedding a linear array in a faulty hypercube is derived in Section 3.2 as Lemma 1. Note that the necessary condition is not applicable to algorithms that use the Direct-Connect Capability approach. In Section 3.3, Algorithm LNRCUBE is presented. It finds a maximum fault-free subcube first, then embeds a linear array in the maximum fault-free subcube. Algorithm LNRLATIN is presented in Section 3.4, that finds a Gray code ordering in a faulty hypercube by a modified Latin Multiplication [27]. The parallel version of Algorithm LNRLATIN, namely PLNRLATIN, is also proposed in Section 3.5. In Section 3.6, Algorithm LNRCSQ is proposed to embed a linear array in a faulty hypercube by making use of the Coordinate sequence approach [28]. In addition, a parallel version of Algorithm LNRCSQ, namely PLNRCSQ, is developed in Section 3.7.
Finally, the Direct-Connect Capability approach is applied to Algorithm LNRDCC described in Section 3.8 for embedding a linear array in a faulty hypercube.

3.2 Necessary Condition for Embedding a Linear Array

Consider a linear array starting from a node with even parity or odd parity. The parity alternates between odd and even along the linear array as shown below:

$$L_e = even - odd - even - odd - even - odd - odd - \cdots$$

$$L_o = odd - even - odd - even - odd - even - odd - even - \cdots$$

If the length of $L_e$ or $L_o$ is an even number, then the number of even parity nodes and the number of odd parity nodes must be identical. Likewise, if the length of $L_e$ or $L_o$ is an odd number, then the absolute difference between the number of even parity nodes and the number of odd parity nodes is one.

Given a $d$-cube with $r$ faulty nodes, a linear array of length $m$ is to be embedded in it, where $m \leq (2^d - r)$. Clearly, there are $(2^d - r - m)$ extra healthy nodes available in the faulty hypercube after a possible embedding. With the property stated previously, if $m$ is an even number, a valid embedding requires that the number of even parity nodes and the number of odd parity nodes in the linear array be identical. In other words, a valid embedding requires that the number of unused nodes with even parity be equal to the number of unused nodes with odd parity since a hypercube has the same number of even and odd parity nodes. The unused nodes consist of $r$ faulty nodes and $(2^d - r - m)$ extra healthy nodes. In order to have the same number of even and odd parity unused nodes, it is required that the absolute difference between the number of faulty nodes with even parity and the number of faulty nodes with odd parity be less than or equal to the number
of extra healthy nodes. Likewise, we can derive the necessary condition for odd values of $m$. Thus, we have Lemma 1 presented as below:

**Lemma 1**: The necessary condition for embedding a linear array of length $m$ in a faulty $d$-cube with $r$ faulty nodes is:

- $|N_{\text{even}} - N_{\text{odd}}| \leq (2^d - r - m)$ when $m$ is an even number, or $|N_{\text{even}} - N_{\text{odd}}| \leq (2^d - r - m + 1)$ when $m$ is an odd number, where $N_{\text{even}}$ is the number of faulty nodes with even parity, $N_{\text{odd}}$ is the number of faulty nodes with odd parity, and $m \leq (2^d - r)$.

Lemma 1 will be applied to the embedding algorithms for checking the necessary conditions. However, Lemma 1 cannot be applied to algorithms that use the *Direct-Connect Capability* approach.

3.3 Description of Algorithm *LNRCUBE*

Algorithm *LNRCUBE* uses the algorithm proposed in [33] to find a maximum fault-free subcube, then embeds the desired linear array in it. Let $N_{\text{even}}$ be the number of faulty nodes with even parity and $N_{\text{odd}}$ be the number of faulty nodes with odd parity. The steps of Algorithm *LNRCUBE* for embedding a linear array of length $m$ in a faulty $d$-cube are listed as follows:

**Algorithm LNRCUBE**: Embedding a linear array of length $m$ in a faulty $d$-cube.

**Step 1**: If $(2^d - r) < m$, then exit.

**Step 2**: If $|N_{\text{even}} - N_{\text{odd}}| > (2^d - r - m)$ when $m$ is an even number, or $|N_{\text{even}} - N_{\text{odd}}| > (2^d - r - m + 1)$ when $m$ is an odd number, then exit.

**Step 3**: Find a maximum fault-free subcube, denoted as $M_d$-subcube.

**Step 4**: If $2^{M_d} < m$, then exit.
Step 5: Embed the desired linear array in the $M_d$-subcube, then exit.

Step 1 checks if there are sufficient number of healthy nodes for embedding the required linear array. Lemma 1 is applied in Step 2 to check the necessary condition for embedding the required linear array. Step 3 uses the algorithm proposed in [33] to find a maximum fault-free subcube, denoted as $M_d$-subcube. After finding an $M_d$-subcube, Step 4 checks if there are sufficient number of nodes in the $M_d$-subcube for embedding the required linear array. It should be noted that Algorithm $LNRCUBE$ may not find the linear array with the maximum length in most of cases. An example is shown in Fig. 3. The maximum fault-free subcube of a 3-cube with a faulty node is 2-cube; therefore, the linear array with the maximum length that can be found by Algorithm $LNRCUBE$ is 4. However, a linear array of length 7 can be found, actually. Hence, alternatives are required to overcome this limitation. The complexity of Algorithm $LNRCUBE$ for embedding a linear array in a faulty hypercube is analyzed as follows: In Step 3, the complexity of finding the $M_d$-subcube has been analyzed in [33]. It is assumed that the number of faulty nodes, $r \leq d$. In Step 5, to embed a linear array in the $M_d$-subcube requires a complexity $O(m)$ provided that the Coordinate sequence approach is used, $S_{M_d}$ is obtained by table-look-up and the $d$ inverting patterns $(00\cdots1,00\cdots10,0\cdots100,\cdots,10\cdots0)$ are stored in a table. The complexity of Step 5 can be neglected. In addition, conditional tests for a possible embedding are required in Steps 1, 2 and 4. Each of them can be done in constant time provided that the values of $d$ and $M_d$ are in a certain predefined range and the applicable values of $2^d$ and $2^{M_d}$ are pre-calculated. Therefore, an upper bound for the complexity of Algorithm $LNRCUBE$ can be expressed as:

$$O\left(\left(\left\lfloor \frac{\log_2 P}{\log_2(\log_2 P)} \right\rfloor + 1\right)(\log_2 P)(\log_2(\log_2 P))\right), \text{ when } d > 4$$
Figure 3: Limitation of Algorithm LNRCUBE

(A linear array of length 4 embedded by Algorithm LNRCUBE)

(A linear array of length 7 can be embedded actually)
\[ O \left( \left( \frac{\log P}{[1/2 \log P]} \right) (\log P)(\log(\log P)) \right), \text{ when } d \leq 4 \]

where \( P = 2^d \) and \( d \) is the dimension of hypercube.

3.4 Description of Algorithm LNRLATIN

3.4.1 Algorithm LNRLATIN

In [27], Kaufmann's Algorithm is presented to enumerate all possible linear arrays in a directed/undirected graph by making use of the Latin multiplication method. The Latin Multiplication is performed like ordinary matrix multiplication, except each entry of the matrix is a string of labels or zero. It is defined as follows:

1. Zero multiplied by anything equals zero.
2. String multiplication concatenates two strings, i.e.: \( v_1v_2v_3v_4 \times v_5v_6v_7v_8v_9v_{10} = v_1v_5v_8v_2v_3v_4 \)
3. String additions just list two strings, i.e.: \( v_1v_2v_3 + v_5v_6v_7v_8v_9v_{10} = v_1v_5v_6, v_1v_3v_4 \)
4. Any string containing a label more than once equals zero.

Kaufmann's Algorithm requires two \( 2^d \times 2^d \) matrices and performs the Latin Multiplication of the matrices \( (m - 1) \) times to obtain the linear arrays of length \( m \). Each entry of the matrices is a string which contains consecutive characters. A string indicates a linear array and a character is a label of a node in the linear array. This is a very expensive algorithm in terms of both memory and time. In order to reduce the memory complexity, the algorithm is modified as described below. Algorithm LNRLATIN embeds a linear array \( L_m \) of length \( m \) in a faulty \( d_l \)-cube with \( r_l \) faulty nodes and \( l_l \) faulty links using a modification of the Latin multiplication method [27], where \( m \leq 2^{d_l} \). The adjacency lists of all healthy nodes [27] are generated. Next, a start node is selected from the nodes which have the minimum number of adjacent nodes. This heuristic is found to speed up the search for the linear array. By appending each node in the adjacency list of the start node to the start node, all linear arrays of length 2 starting from the start node are created. To create all linear arrays of length 3...
starting from the start node, each linear array of length 2 is examined to see if the
nodes in the adjacency list of the last node can be appended to it without creating
a cycle. By repeatedly performing the appending operation and in the worst case
considering all nodes as the start node, the linear array of the required length will
be found if one exists.

Algorithm LNRLATIN:

Given a $d_l$-cube with $r_l$ faulty nodes and $l_l$ faulty links, and a linear array
$L_m$ of length $m$ is to be embedded in it, where $m \leq 2^{d_l}$. Let $N_a$ indicate the
number of linear arrays, and $l_a$ indicate the length of linear arrays.

Step 1: Apply Lemma 1 in the host computer. If $|N_{even} - N_{odd}| > (2^d - r - m)$
when $m$ is an even number, or $|N_{even} - N_{odd}| > (2^d - r - m + 1)$ when $m$
is an odd number, then exit.

Step 2: Generate the adjacency list.

Step 3: Update the adjacency list by deleting the faulty nodes and/or links.

Step 4: Select a start node. Then create all the linear arrays of $l_a=2$ and update
$N_a$. If $m = 2$, then the required linear array $L_m$ is found and exit.

Step 5: Select a linear array of length $l_a$.

Step 6: Examine the adjacency list of the last node of the linear array selected
in Step 5. If the node in the adjacency list does not appear in this array,
create a new linear array of length $l_a + 1$ by appending it to the last node.
If $l_a + 1 = m$, then the linear array is found and exit. If no linear arrays of
length $l_a + 1$ can be created, the linear array selected in Step 5 is discarded.
Repeat the appending operation until the desired linear array is found or
$N_a=0$. 
Step 7: Check if all the linear arrays of length $l_a$ have been selected in Step 5. If not, go to Step 5. If yes, check if all healthy nodes are used in Step 4. If yes, exit; else go to Step 4.

3.4.2 An Embedding Example by Algorithm LNRLATIN

An example to embed a linear array in a faulty hypercube by making use of Algorithm LNRLATIN is presented here. Fig. 4 shows a faulty 3-cube with the faulty elements $F_1 = 001$, $F_2 = 010$ and $L_1 = 1z1$. Now consider a linear array of length 5 which is required to be embedded in it by making use of Algorithm LNRLATIN. The steps to embed the desired linear array are described as follows:

Step 1: Since $F_1 = 001$, $F_2 = 010$, $r_l = 2$ and $m = 5$, we have $N_{even} = 0$, $N_{odd} = 2$, $|N_{even} - N_{odd}| = |0 - 2| = 2$, $(2^d_l - r_l - m + 1) = (2^3 - 2 - 5 + 1) = 2$ and $|N_{even} - N_{odd}| = (2^d_l - r_l - m + 1)$. According to Lemma 1, the embedding is possible.

Step 2: Generate the adjacency list for each healthy node of the faulty hypercube. The adjacency list is given in Fig. 4(b).

Step 3: Update the adjacency list for each healthy node by deleting the faulty elements 001, 010 and 1z1. We have the updated adjacency list: \{node\} \rightarrow \{adjacent nodes\}, \{000\} \rightarrow \{100\}, \{011\} \rightarrow \{111\}, \{100\} \rightarrow \{101, 110, 000\}, \{101\} \rightarrow \{100\}, \{110\} \rightarrow \{111, 100\} and \{111\} \rightarrow \{110, 011\}. The updated adjacency list is also shown in Fig. 4(b). The candidate start nodes which contain the minimum number of adjacent nodes are 000, 011 and 101.

Steps 4-7: node 000 is selected as the start node. The linear array of length 2 is 000-100; $N_a = 1$ and $l_a = 2$. Since $l_a = 2 \neq m$ and $N_a = 1 \neq 0$, the linear
Figure 4: A linear array of length 5 embedded in a faulty 3-cube using Algorithm LNRLATIN (a) A faulty hypercube (b) The adjacency list
array 000-100 is selected. The adjacent nodes of the last node (100) are 101, 110 and 000. The two new linear arrays are 000-100-101 and 000-100-110. The linear array 000-100-000 is discarded since node 000 appears twice. \( N_a = 2 \) and \( l_a = 3 \). Since \( l_a \neq m \) and \( N_a \neq 0 \), the appending operation is repeated. The adjacency list of the last node (101) consists of 100 which will create a cycle. The linear array 000-100-101 is discarded. \( N_a = 1 \) and for the linear array 000-100-110, the adjacent nodes of the last node (110) are 111 and 100. A new linear array 000-100-110-111 is obtained and the linear array 000-100-110-100 which has a cycle is discarded. \( l_a \neq m \) and \( N_a \neq 0 \), the linear array 000-100-110-111-110 which creates a cycle is discarded and the desired linear array 000-100-110-111-011 is obtained.

### 3.4.3 The Time Complexity of Algorithm LNRLATIN

The time complexity of Algorithm LNRLATIN can be analyzed as follows:

The complexity of Step 1 is \( O(r_l d_l) \) provided that \( 2^{d_l} \) is pre-calculated. The complexity of generating the adjacency list for all the nodes of a \( d_l \)-cube can be done in constant time by table-look-up. Updating of the adjacency list for all the nodes has a complexity of \( O(2^{d_l}(r_l + l_l)) \). There are \( 2^{d_l} - r_l \) healthy nodes in the faulty \( d_l \)-cube. Let \( N_{cad} \) be the number of healthy nodes. In the worst case, all healthy nodes are required to be the start node. The number of adjacent nodes of any node is less than or equal to \( d_l \). There are at most \( (d_l - 1) \) nodes in the adjacency list of the last node and each node in the adjacency list is required to be appended to a linear array in the worst case. It is clear that a linear array of length \( m \) requires \( (m - 1) \) nodes be appended to the start node one by one for forming a linear array of length \( m \). Therefore, an upper bound for the complexity
of Algorithm LNRLATIN can be expressed as:

\[ O \left( r_l d_l + 2^{d_l}(r_l + l_l) + N_{cad}d_l^m \right) \]

or

\[ O \left( 2^{d_l}(r_l + l_l) + N_{cad}d_l^m \right) \text{ since } r_l d_l \leq r_l 2^{d_l}. \]

The dominating term in the worst case complexity is \( O(N_{cad}d_l^m) \), where \( N_{cad} \) is the number of the start nodes, since there are at most \( d_l \) nodes in the adjacency list of a node and in the worst case each node in the adjacency list will be tried for each of the \((m - 1)\) appending operations. Note that this is a very pessimistic upper bound. All \( d_l^m \) choices are not evaluated because of the following reasons: The process is terminated when the first array of length \( m \) is found; every node does not have \( d_l \) adjacent nodes because of faulty nodes, and paths in the search tree are not evaluated further when a node in the adjacency list causes a cycle.

### 3.5 The Parallel Version of Algorithm LNRLATIN

Algorithm LNRLATIN is intended to execute in the host computer or the cube manager of a hypercube multiprocessor system. However, it can be parallelized in a faulty hypercube by distributing the adjacency list and the partitioned healthy nodes to each healthy node, and performing the modified Latin Multiplication in each healthy node of the faulty hypercube. Once the desired linear array is found by any healthy node, it is sent to the host computer and the processes executing in all healthy nodes are killed. If all healthy nodes cannot find the desired linear array, the embedding operation is also terminated. The parallel version of Algorithm LNRLATIN is listed as follows:
Algorithm \textit{PLNRLATIN}:

\textbf{Step 1:} Apply Lemma 1 in the host computer. If \( |N_{\text{even}} - N_{\text{odd}}| > (2^d - r - m) \) when \( m \) is an even number, or \( |N_{\text{even}} - N_{\text{odd}}| > (2^d - r - m + 1) \) when \( m \) is an odd number, then exit.

\textbf{Step 2:} Generate the adjacency list in the host computer.

\textbf{Step 3:} The host computer updates the adjacency list by deleting the faulty nodes and/or links.

\textbf{Step 4:} The host computer distributes the adjacency list and the partitioned healthy nodes to each healthy node.

\textbf{Step 5:} Each healthy node performs the following operations.

\textbf{Step 5.1:} Select a start node from the partitioned healthy nodes, then create all the linear arrays of \( l_a=2 \) and update \( N_a \). If \( m = 2 \), then the required linear array \( L_m \) is found and it is sent to the host computer.

\textbf{Step 5.2:} Select a linear array of length \( l_a \).

\textbf{Step 5.3:} Examine the adjacency list of the last node of the linear array selected in Step 5.2. If the node in the adjacency list does not appear in this array, create a new linear array of length \( l_a + 1 \) by appending it to the last node. If \( l_a + 1 = m \), then the linear array is found and it is sent to the host computer. If no linear arrays of length \( l_a + 1 \) can be created, the linear array selected in Step 5.2 is discarded. Repeat the appending operation until the desired linear array is found or \( N_a = 0 \). If the desired linear array is found, then it is sent to the host computer.
Step 5.4: Check if all the linear arrays of length $l_a$ have been selected in Step 5.2. If no, go to Step 5.2. If yes, check if all the partitioned healthy nodes have been selected in Step 5.1. If yes, inform the host computer that the desired linear array cannot be found; else go to Step 5.1.

Step 6: After receiving the desired linear array, or all healthy nodes cannot find the desired linear array, the host computer terminates the processes executing in all healthy nodes.

In the worst case, an upper bound for the complexity of Algorithm PLNR-LATIN can be expressed as:

$$O \left(2^{d_l}(r_l + l_l) + \frac{N_{cad}}{P'} d_l m \right),$$

or

$$O \left(2^{d_l}(r_l + l_l) + d_l m \right).$$

where $P'$ is the number of healthy nodes and in the worst case, $N_{cad}$ is equal to $P'$. $d_l$ is the dimension of the faulty hypercube, $r_l$ is the number of faulty nodes, $l_l$ is the number of faulty links, $m$ is the length of the linear array and $N_d$ is the maximum number of consecutive discarded elements.

3.6 Description of Algorithm LNRCSQ

3.6.1 Algorithm LNRCSQ

A faster algorithm for embedding a linear array in a faulty hypercube can be formulated by using a modification of the Coordinate sequence approach. In a fault-free $d$-cube, a linear array of length $2^d$ can be found by starting at any node and traveling dimensions specified in the coordinate sequence $S_d$. However, if one tries to do this in a faulty $d$-cube, faulty nodes and/or links will be encountered and therefore the sequence must be modified. The coordinate sequence
$S_d = \{0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, \ldots, 4, 0, \ldots\}$ forms a linear array by visiting every node in a 2-cube, then moving along dimension 2 to another 2-cube, then along dimension 3 to another 2-cube which is part of a new 3-cube and so on. Based on this observation, $S_d$ is modified as described next to construct a linear array in a faulty $d$-cube. The links connected to a faulty node and the faulty nodes are defined as prohibit links in constructing the linear array. If a prohibit link is encountered, then the corresponding element of $S_d$ is discarded. However when an element is discarded, cycles can be created. For instance, if 2 is discarded in the sequence $\{0, 1, 0, 2, 0, 1, 0\}$, the nodes in the 2-cube defined by the first $\{0, 1, 0\}$ in the sequence will be revisited in reverse order by the second $\{0, 1, 0\}$ and therefore the $\{0, 1, 0\}$ following the 2 must also be discarded. In general two consecutive identical sequences will create cycles and the second subsequence must be deleted. This can be done in the following way. Denoting the number of consecutive discarded elements as $N_{dis}$, the next element in $S_d$ is compared to the previous $N_{dis}$ elements in the sequence that were not discarded. If it is equal to any one of those elements, it is discarded. This operation is repeated until no new element is discarded. When link 0 is a prohibit link of the start node, $(d - 1)$ should be appended to $S_d$. This ensures that the linear array will include the adjacent node of start node through link 0 depending upon that the element used before last one is equal to 0. Finally, if the consecutive discarded elements include the last element in $S_d$, then the embedding operation for the selected start node is finished. In the worst case, all healthy nodes will be selected as the start node.

Algorithm \textit{LNRCSQ} may not find a linear array with the maximum length because it does not try all healthy links of a node when this node is visited for embedding. This is different from Algorithm \textit{LNRLATIN} which always finds the linear array of the maximum length. The following notation is used in Algo-
Algorithm \textit{LNRCSQ}.

\[ L_{\text{dis}} = \text{List of consecutive discarded coordinates of} \ S_d. \]

\[ L_{\text{used}} = \text{List of coordinates used for creating the linear array.} \]

\[ N_{\text{dis}} = \text{Number of the consecutive discarded elements.} \]

\[ LNR = \text{The linear array embedded.} \]

\textbf{Algorithm} \textit{LNRCSQ}: \hspace{1cm}

Given a \( d_l \)-cube with \( r_l \) faulty nodes and \( l_l \) faulty links, and a linear array \( LNR \) of length \( m \) is to be embedded in it, where \( m \leq 2^{d_l} \).

1. If \(| N_{\text{even}} - N_{\text{odd}} | > (2^{d_l} - r_l - m) \) when \( m \) is an even number, or \(| N_{\text{even}} - N_{\text{odd}} | > (2^{d_l} - r_l - m + 1) \) when \( m \) is an odd number, then exit.

2. Generate the list of prohibit links for each node of a \( d \)-cube.

3. Select a node from the healthy nodes containing the maximum number of prohibit links.

4. Clear \( L_{\text{dis}}, L_{\text{used}}, LNR, N_{\text{dis}} \) and obtain \( S_d \).

5. Let current node = start node and \( LNR = \{\text{startnode}\} \). If link 0 is the prohibit links of start node, append \((d - 1) \) to \( S_d \).

6. \textbf{if} (the corresponding element of \( S_d \) = a prohibit link of current node) \textbf{or} (the corresponding element of \( S_d \) = any one of the previous \( N_{\text{dis}} \) elements in \( L_{\text{used}} \)) \textbf{or} ((the corresponding element of \( S_d \) is the last one (\( \neq 0 \))) \textbf{and} (the previous element used \( \neq 0 \))
then {
    put the element in $L_{C_{dis}}$
    update $N_{dis}$
}
else {
    put the element in $L_{C_{used}}$
    clear $L_{C_{dis}}$ and $N_{dis}$
    append the next nodes obtained by complementing
    the bit indicated by the element to $LNR$
    if the length of $LNR = m$, then exit
}

7. if all the elements of $S_d$ are not examined
    then {
        let current node = last node of $LNR$
        go to Step 6
    }

8. if all healthy nodes are tried then exit
    else go to step 3

3.6.2 An Embedding Example by Algorithm $LNRC_{SQ}$

Fig. 5 shows a faulty 3-cube with faulty nodes $F_1 = 001$, $F_2 = 010$ and a
faulty link $l_1 = 111$. A linear array of length 5 is required to be embedded in it.
We shall apply Algorithm \textit{LNRCSQ} to embed the required linear array.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\linewidth]{figure5}
\caption{A faulty 3-cube with faulty elements $F_1 = 001$, $F_2 = 010$ and $L_1 = 1z1$}
\end{figure}

\textbf{Step 1:} Since $F_1 = 001$, $F_2 = 010$, $r_l = 2$ and $m = 5$, we have $N_{\text{even}} = 0$, $N_{\text{odd}} = 2$, $|N_{\text{even}} - N_{\text{odd}}| = |0 - 2| = 2$, $(2^d - r_l - m + 1) = (2^3 - 2 - 5 + 1) = 2$. Thus we have $|N_{\text{even}} - N_{\text{odd}}| = (2^d - r_l - m + 1)$. According to \textit{Lemma 1}, the embedding is possible.

\textbf{Step 2:} Generate the prohibit link for each node of the faulty 3-cube. From the faulty elements 001, 010 and 1z1, we have the prohibit links for each healthy node listed in Table 1 and the candidate start nodes containing the maximum number of prohibit links are 000, 011 and 101.

\textbf{Step 3:} Select node 000 from \{000, 011, 101\} as the start node.

\textbf{Step 4:} $L_{\text{Cdis}} = \{\}, L_{\text{Cused}} = \{\}, LNR = \{\}, N_{\text{dis}} = 0$ and $S_d = \{0, 1, 0, 2, 0, 1, 0\}$.

\textbf{Steps 5-7:} Let current node = 000 and $LNR = \{000\}$. Since the prohibit links of node 000 are 0 and 1, we have the following operation sequence:
Table 1: Prohibit links of an embedding example by Algorithm LNRCSQ

<table>
<thead>
<tr>
<th>node</th>
<th>prohibit links</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0,1</td>
</tr>
<tr>
<td>011</td>
<td>0,1</td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>1,2</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
</tr>
</tbody>
</table>

1. $L_{C_{dis}} = \{0\}$, $L_{C_{used}} = \{}$, $N_{dis} = 1$, $LNR = \{000\}$ and $S_d = \{1,0,2,0,1,0\}$.

2. $L_{C_{dis}} = \{0,1\}$, $L_{C_{used}} = \{}$, $N_{dis} = 2$, $LNR = \{000\}$ and $S_d = \{0,2,0,1,0\}$.

3. $L_{C_{dis}} = \{0,1,0\}$, $L_{C_{used}} = \{}$, $N_{dis} = 3$, $LNR = \{000\}$ and $S_d = \{2,0,1,0\}$.

4. $L_{C_{dis}} = \{}$, $L_{C_{used}} = \{2\}$, $N_{dis} = 0$ and $S_d = \{0,1,0\}$. We have $LNR = \{000,100\}$, where 100 is obtained by complementing 000 on the bit position 2. Let current node = 100.

5. Since node 100 has no prohibit link, we have $L_{C_{dis}} = \{}$, $L_{C_{used}} = \{2,0\}$, $N_{dis} = 0$, $S_d = \{1,0\}$ and $LNR = \{000,100,101\}$. Let current node = 101.

6. Since node 101 has prohibit links \{1,2\} and $S_d = \{1,0\}$, we have $L_{C_{dis}} = \{1\}$, $L_{C_{used}} = \{2,0\}$, $N_{dis} = 1$, $S_d = \{0\}$ and $LNR = \{000,100,101\}$. Let current node = 101.
7. Since node 101 has the prohibit links \{1, 2\}, and \(N_{dis} = 1\), and \(S_d = \emptyset\)

=1= the last element of \(L_{Cused} = \{2, 0\}\). We have \(L_{Cdis} = \{1, 0\}\),
\(L_{Cused} = \{2, 0\}\), \(N_{dis} = 2\) and \(S_d = \emptyset\). The trial for the start node

000 is finished.

Step 8: Remove node 000 from the candidate start nodes \{000, 011, 101\}. We

have the candidate start nodes \{011, 101\} and go to Step 3.

Step 3: Select node 011 from \{011, 101\} as the start node.

Step 4: \(L_{Cdis} = \emptyset, L_{Cused} = \emptyset, LNR = \emptyset, N_{dis} = 0\) and \(S_d = \{0, 1, 0, 2, 0, 1, 0\}\).

Steps 5-7: Let current node = 011 and \(LNR = \{011\}\). Since the prohibit links

of node 011 are 0 and 1. We have the following operation sequence:

1. \(L_{Cdis} = \{0\}, L_{Cused} = \emptyset, N_{dis} = 1\) \(LNR = \{011\}\) and \(S_d = \{1, 0, 2, 0, 1, 0\}\).

2. \(L_{Cdis} = \{0, 1\}, L_{Cused} = \emptyset, N_{dis} = 2, LNR = \{011\}\) and \(S_d = \{0, 2, 0, 1, 0\}\).

3. \(L_{Cdis} = \{0, 1, 0\}, L_{Cused} = \emptyset, N_{dis} = 3, LNR = \{011\}\) and \(S_d = \{2, 0, 1, 0\}\).

4. \(L_{Cdis} = \emptyset, L_{Cused} = \{2\}, N_{dis} = 0\) and \(S_d = \{0, 1, 0\}\). \(LNR = \{011, 111\}\), where 111 is obtained by complementing 011 on the bit

position 2. Let current node = 111.

5. Since node 111 has prohibit link 1, we have \(L_{Cdis} = \emptyset, L_{Cused} = \{2, 0\},

\(N_{dis} = 0\) and \(S_d = \{1, 0\}\). \(LNR = \{011, 111, 110\}\). Let current node = 110.
6. Since node 110 has prohibit link 2 and $S_d = \{1, 0\}$, we have \( L_{Cdis} = \{\} \), \( L_{Cused} = \{2, 0, 1\} \), \( N_{dis} = 0 \) and \( S_d = \{0\} \). \( LNR = \{011, 111, 110, 100\} \). Let current node = 100.

7. Since node 100 has no prohibit links and \( S_d = \{0\} \), we have \( L_{Cdis} = \{\} \), \( L_{Cused} = \{2, 0, 1, 0\} \), \( N_{dis} = 0 \), \( S_d = \{\} \) and \( LNR = \{011, 111, 110, 100, 101\} \). Since the length of \( LNR \) is 5, the embedding operation is finished.

3.6.3 The Time Complexity of Algorithm LNRCSQ

The complexity of Algorithm LNRCSQ can be analyzed as follows: the complexity of Step 1 is \( O(r_d l_l) \) provided that \( 2^d_l \) is precalculated. The complexity of generating the prohibit links for all the nodes of a \( d_l \)-cube is \( O(r_d l_l + l_l) \). Assume that \( N_m \) denotes the number of nodes containing the maximum number of prohibit links. In the worst case, the complexity of selecting a start node is \( O(N_m) \).

To obtain the coordinate sequence \( S_d \) by table-look-up can be done in constant time. Assume that the \( d \) inverting patterns \((0 \cdots 01, 0 \cdots 10, \cdots, 010 \cdots 0, 10 \cdots 0)\) are stored in a table and the maximum number of consecutive discarded elements is denoted as \( N_d \). The complexity of Step 5 is \( O(m_d l + N_d) \). Therefore, an upper bound for the complexity of Algorithm LNRCSQ can be expressed as :

\[
O (r_d l_l + r_l d_l + l_l + N_m (m_d l + N_d))
\]

or

\[
O (r_d l_l + N_m (m_d l + N_d)),
\]

where \( d_l \) is the dimension of the faulty hypercube, \( r_l \) is the number of faulty nodes, \( l_l \) is the number of faulty links, \( N_m \) is the number of the candidate start nodes.
which contain the maximum number of prohibit links, \( m \) is the length of the linear array and \( N_d \) is the maximum number of consecutive discarded elements.

### 3.7 The Parallel Version of Algorithm \( LNRCSQ \)

Algorithm \( LNRCSQ \) can also be parallelized in a faulty hypercube by distributing the partitioned candidate start nodes to each healthy node, and performing the Coordinate sequence approach with the applicable variations. Once the desired linear array is found by any healthy node, it is sent to the host computer and the processes executing in all healthy nodes are killed. If all healthy nodes cannot find the desired linear array, the embedding operation is also terminated. The parallel version of Algorithm \( LNRCSQ \) is listed as follows:

**Algorithm \( PLNRCSQ \):**

**Step 1:** Apply Lemma 1 in the host computer. If \( |N_{\text{even}} - N_{\text{odd}}| > (2^{d_l} - r_l - m) \) when \( m \) is an even number, or \( |N_{\text{even}} - N_{\text{odd}}| > (2^{d_l} - r_l - m + 1) \) when \( m \) is an odd number, then exit.

**Step 2:** The host computer generates the list of prohibit links for each node of a \( d_l \)-cube and obtains the candidate start nodes containing the maximum number of prohibit links.

**Step 3:** The host computer distributes the partitioned candidate start nodes to each healthy node.

**Step 4:** Each healthy node performs the following operations.

**Step 4.1:** Select a node from the candidate start nodes which are distributed to this node.

**Step 4.2:** Clear \( L_{C\text{dis}}, L_{C\text{used}}, LNR, N_{\text{dis}} \) and obtain \( S_d \).
Step 4.3 : Let current node = start node and $LNR = \{\text{startnode}\}$. If link 0 is the prohibit links of start node, append $(d - 1)$ to $S_d$.

Step 4.4 :

if (the corresponding element of $S_d$ = a prohibit link of current node) or (the corresponding element of $S_d$ = any one of the previous $N_{dis}$ elements in $L_{Cused}$) or ((the corresponding element of $S_d$ is the last one ($\neq 0$)) and (the previous element used $\neq 0$))

then {
    put the element in $L_{Cdis}$
    update $N_{dis}$
}

else {
    put the element in $L_{Cused}$
    clear $L_{Cdis}$ and $N_{dis}$
    append the next nodes obtained by complementing the bit indicated by the element to $LNR$
    if the length of $LNR = m$, then send $LNR$ to the host computer
}

Step 4.5 :

if all the elements of $S_d$ are not examined

then {
    let current node = last node of $LNR$
    go to Step 4.4
Step 4.6:

if all candidate start nodes are tried

then inform the host computer that

the desired linear array cannot found

else go to step 4.1

Step 5: After receiving the desired linear array, or all healthy nodes cannot find the desired linear array, the host computer terminates the processes in all healthy nodes.

Since the candidate start nodes are partitioned and distributed to each healthy node in the faulty hypercubes, The time complexity of Algorithm PLNRCSQ is better than that of Algorithm LNRCSEQ. In the worst case, Algorithm LNRCSEQ tries all healthy nodes as start node and $N_m$ is equal to $P'$. Therefore, an upper bound for the complexity of Algorithm PLNRCSQ can be expressed as

$$O \left( r_l d_l + \frac{N_m}{P'} (m d_l + N_d) \right),$$

or

$$O (r_l d_l + m d_l + N_d).$$

where $d_l$ is the dimension of the faulty hypercube, $r_l$ is the number of faulty nodes, $l_l$ is the number of faulty links, $N_m$ is the number of the candidate start nodes which contain the maximum number of prohibit links, $m$ is the length of the linear array, $P'$ is the number of healthy nodes used in the faulty hypercube and $N_d$ is the maximum number of consecutive discarded elements.
3.8 Description of Algorithm LNRDCC

3.8.1 Algorithm LNRDCC

Algorithm LNRDCC embeds a linear array in a faulty hypercube by making use of the Direct-Connect Capability. In this approach, we replace the e-cube algorithm used in [41] by an e-cube-like algorithm, namely Algorithm BYPASS, which will be described later. Each node in the hypercube with the Direct-Connect Capability has routing hardware, denoted as DCH. The DCH's of all the nodes form the hypercube network by the interconnection links. Here, we assume that the links and the DCH's are fault-free, and the faulty hypercube has faulty nodes only.

Algorithm LNRDCC first embeds a linear array of length $2^d$ in a d-cube by making use of the Coordinate sequence approach, then obtains the linear array directly by removing the faulty nodes and connecting through the DCH's of the removed faulty nodes. Fig. 6 shows an initial embedding of a linear array made first by using $S_3$ and the desired linear array is obtained by simply removing the faulty nodes. Referring to Fig. 6.(c) and according to the e-cube algorithm [38,41], the communication from node 110 to 011 will go through the DCH of node 111; therefore, the communication between node 111 and node 101 is interrupted by the communication between node 110 and node 011. Consequently, a degraded performance of local communication is resulted if Algorithm BYPASS is not used.

In a linear array embedded by Algorithm LNRDCC, Algorithm BYPASS resided in the DCH of each node uses an e-cube-like algorithm to enable the communication between any neighboring nodes exactly via a predefined path established by the DCH's of some removed faulty nodes between the source and the destination. The predefined path does not include any DCH's of the other healthy nodes so that the local communication can be done simultaneously without inter-
ruption. For example, the communication between node 000 and node 011 shown in Fig. 6 will exactly go through the DCH of node 001; similarly, the communication between node 011 and node 110 will exactly go through the DCH of node 010. Algorithm \textit{LNRDCC} can be formulated as follows:

\textbf{Algorithm LNRDCC}:

\noindent \textbf{Step 1}: If \((2^d - r) < m\), then exit.

\noindent \textbf{Step 2}: Obtain a coordinate sequence \(S_d\).

\noindent \textbf{Step 3}: Embed a linear array by making use of \(S_d\).

\noindent \textbf{Step 4}: Form the linear array by simply removing \(r\) faulty nodes.

Step 1 checks if there are sufficient number of healthy nodes to construct the linear array. Step 2 generates the required coordinate sequence \(S_d\), which can be done by table-look-up, provided that the applicable values of \(S_d\) are precalculated. Step 3 embeds a linear array by using \(S_d\). Here, we select node 0 as the start node which can be arbitrary node in the hypercube. Finally, Step 4 forms the linear array.

3.8.2 An Embedding Example by Algorithm \textit{LNRDCC}

Fig. 6 shows an example of embedding a linear array of length 6 in a faulty 3-cube with faulty nodes \(F_1 = 010\) and \(F_2 = 001\) by making use of Algorithm \textit{LNRDCC}. The steps of embedding the desired linear array are listed as follows:

\noindent \textbf{Step 1}: Since \(2^3 - 2 = 6\), the embedding must be possible. We go to Step 2.

\noindent \textbf{Step 2}: Obtain \(S_3 = \{0, 1, 0, 2, 0, 1, 0\}\) by table-look-up.
Step 3: Select node 000 as the start node and embed the linear array by making use of $S_3$.

Step 4: Remove the faulty node and obtain the linear array 000-011-110-111-101-100, where nodes 001 and 010 are faulty.

3.8.3 The Time Complexity of Algorithm LNRDCC

The time complexity of Algorithm LNRDCC for embedding a linear array in a faulty hypercube is analyzed as follows: Step 1 can be done in constant time by table-look-up, arithmetic and conditional checking operations provided that the values of $d$ are in a certain predefined range and the applicable values of $2^d$ are pre-calculated. Step 2 can be finished in constant time by assuming that the applicable values of $S_d$ are stored in a table. If $S_d$ is not stored in a table, it can be generated by the following algorithm:

Algorithm GCSQ:

Begin
$S_1 = \{0\}$;

for $i=2$ to $d$

$S_i = \{S_{i-1}, i-1, S_{i-1}\}$;

Endfor

End

The complexity of Algorithm GCSQ is $O(2^1 + 2^2 + \cdots + 2^d - d)$, or $O(2^{d+1} - 2 - d)$, since if $S = (2^1+2^2+\cdots+2^d-d)$, then $2S = (2^2+2^3+\cdots+2^d+2^{d+1}-2d)$. We have $2S - S = 2^{d+1} - 2 - d$. The table-look-up method to obtain $S_d$ is preferred since it has a constant complexity. In Step 3, the complexity of generating a linear array by using $S_d$ is $O(2^d)$ since there are $2^d-1$ nodes required to be determined after a
Figure 6: A linear array embedded in a faulty 3-cube using the Direct Connect Capability Approach (a) The faulty hypercube (b) An initial embedding (c) The final embedding
start node is selected. The determining process for each node requires a constant
time provided that the d inverting patterns (00 • 1, 00 • 10, 0 • 100, • • 10 • 0)
are stored in a table. In Step 4, forming the linear array has a complexity $O(r2^d)$. Thus, an upper bound for the time complexity of Algorithm LNRDCC can be expressed as

$$O \left( 2^d - 1 + r2^d \right) = O \left( (r + 1)2^d - 1 \right) \leq O \left( r2^d \right) \leq O \left( rP \right),$$

where $P$ is the number of processors and $r$ is the number of faulty processors.

3.9 Summary

We have presented different approaches of embedding linear arrays in a faulty hypercube. Algorithm LNRCUBE combines existing algorithms to embed linear arrays in a faulty hypercube. It may not find the maximum linear array. Algorithm LNRLATIN has been developed to overcome the limitation of Algorithm LNRCUBE. It finds a linear array in a faulty hypercube by making use of a modified approach of the Latin Multiplication [27]. The parallel version of Algorithm LNRLATIN has also been presented. Algorithm LNRLATIN shows a very pessimistic complexity in the worst-case. However, many choices are discarded because of faulty elements and cycles in actual cases. In order to improve the complexity, we have developed a faster algorithm, namely Algorithm LNRCSQ, to embed a linear array in a faulty hypercube by making use of a modification of the Coordinate sequence approach. The complexity of Algorithm LNRCSQ is better than Algorithm LNRLATIN. However, it may not find the maximum length array. When the number of faulty nodes/links in the hypercube is relatively small, Algorithm LNRCSQ will find a near maximum length array. Algorithm LNRDCC has been developed to embed a linear array in a faulty hypercube which has the
Direct-Connect Capability. In this approach, it is assumed that the links and the DCH's are fault-free, and the faulty hypercube has faulty processors only.
CHAPTER IV

ALGORITHMS FOR EMBEDDING 2-D MESHES IN FAULTY HYPERCUBES

4.1 Introduction

A 2-D mesh is a very widely used topology in many parallel implementations which usually require a 2-dimensional local communication. Algorithms for embedding 2-D meshes in a healthy hypercube have been developed by many researchers [10,11,40]. However, they cannot be applied to a faulty hypercube. Therefore, algorithms for embedding an $m_1 \times m_2$ mesh in a faulty hypercube are proposed and given in Algorithm \textsc{meshcube}, Algorithm \textsc{meshlatin}, Algorithm \textsc{pmeshlatin1}, Algorithm \textsc{pmeshlatin2}, Algorithm \textsc{meshdcc} and Algorithm \textsc{pmeshdcc}. First, the necessary condition for embedding a 2-D MESH in a faulty hypercube is derived in Section 4.2 as Lemma 2. Note that the necessary condition is not applicable to algorithms which use the Direct-Connect Capability approach. Algorithm \textsc{meshcube} first uses the algorithm proposed in [33] to find a maximum fault-free subcube, then embeds the desired mesh in the maximum fault-free subcube. Algorithm \textsc{meshlatin} first deletes the faulty columns and/or rows in a $2^{d_1} \times 2^{d_2}$ mesh, where $d_1 \geq \lceil \log_2 m_1 \rceil$, $d_2 \geq \lceil \log_2 m_2 \rceil$ and $(d_1 + d_2) \leq d$, then pastes the healthy columns and rows in a 2-D Gray code ordering. Algorithm \textsc{pmeshlatin1} and Algorithm \textsc{pmeshlatin2} are two different parallel versions of Algorithm \textsc{meshlatin}. Algorithm \textsc{meshdcc} first removes
the faulty columns and/or rows in a $2^{d_1} \times 2^{d_2}$ mesh and forms the desired mesh directly from the healthy columns and rows. It can only be applied to hypercubes with Direct-Connect Capability. Algorithm \textsc{Bypass} is executed by each routing module and it is assumed that the hypercube has faulty processors only. Algorithm \textsc{Pmeshdcc} is a parallel version of Algorithm \textsc{Meshdcc}. The time complexity of each algorithm is analyzed.

4.2 Necessary Condition for Embedding a 2-D Mesh

Consider an $m_1 \times m_2$ mesh embedded in a hypercube. The 2-D mesh has the following property: If $m_1m_2$ is an even number, then it is clear that the number of even parity nodes and the number of odd parity nodes in the mesh must be identical. If $m_1m_2$ is an odd number, then it is clear that the absolute difference between the number of odd parity nodes and the number of even parity nodes is one.

Given a $d$-cube with $r$ faulty nodes, an $m_1 \times m_2$ mesh is desired to be embedded in it, where $m_1m_2 \leq (2^d - r)$. With the property stated above, the necessary condition for embedding a 2-D mesh can be derived and expressed by the following lemma.

**Lemma 2**: The necessary condition for embedding an $m_1 \times m_2$ mesh in a faulty $d$-cube with $r$ faulty nodes is:

$$|N_{even} - N_{odd}| \leq (2^d - r - m_1m_2)$$

when $m_1m_2$ is an even number, or

$$|N_{even} - N_{odd}| \leq (2^d - r - m_1m_2 + 1)$$

when $m_1m_2$ is an odd number, where $N_{even}$ is the number of faulty nodes with even parity, $N_{odd}$ is the number of faulty nodes with odd parity, and $m_1m_2 \leq (2^d - r)$.

**Lemma 2** will be applied to the embedding algorithms for checking the neces-
sary conditions. However, Lemma 2 cannot be applied to algorithms that use the Direct-Connect Capability approach.

4.3 Description of Algorithm MESHcube

Algorithm MESHcube first uses the algorithm proposed in [33] to find a maximum fault-free subcube, then embeds the desired mesh in the maximum fault-free subcube. The steps of Algorithm MESHcube are listed as follows:

**Algorithm MESHcube: Embedding a 2-D mesh in a faulty hypercube**

**Step 1:** If \((d - d_1) < d_2\) or \((2^d - r) < m_1m_2\), then exit.

**Step 2:** If \(|N_{\text{even}} - N_{\text{odd}}| > (2^d - r - m_1m_2)\) when \(m_1m_2\) is an even number, or \(|N_{\text{even}} - N_{\text{odd}}| > (2^d - r - m_1m_2 + 1)\) when \(m_1m_2\) is an odd number, then exit.

**Step 3:** Find a maximum fault-free subcube, denoted as \(M_d\)-subcube.

**Step 4:** If \((M_d - d_1) < d_2\), then exit.

**Step 5:** Embed the desired \(m_1 \times m_2\) mesh in the \(M_d\)-subcube, then exit.

Step 1 checks if the \(m_1 \times m_2\) mesh can be embedded in a healthy \(d\)-cube and if there are sufficient number of healthy nodes in the faulty hypercube for embedding the required mesh. Lemma 2 is applied in step 2 to check if the necessary condition can be satisfied. Step 3 uses the algorithm proposed in [33] to find a maximum fault-free subcube (\(M_d\)-subcube). After finding an \(M_d\)-subcube, step 4 checks if the desired mesh can be embedded in the \(M_d\)-subcube. It should be noted that Algorithm MESHcube may not find the maximum 2-D mesh in some cases. An example is shown in Fig. 7. The maximum fault-free subcube of a 3-cube with a faulty node is 2-cube; therefore, the maximum 2-D mesh that can be found by
Algorithm $MESH\text{CUBE}$ is $2 \times 2$. However, the maximum 2-D mesh in a 3-cube with a faulty node is $2 \times 3$. Algorithm $MESH\text{LATIN}$ described in the next section overcomes this limitation.

4.4 Description of Algorithm $MESH\text{LATIN}$

4.4.1 Algorithm $MESH\text{LATIN}$

As shown in Fig. 9, the faulty elements shown in Fig. 8 are distributed at different positions in the meshes represented by Karnaugh-maps [42] obtained by assigning different coordinates to represent each side of the maps. Now we shall examine the possible embeddings of a $3 \times 3$ mesh in the different Karnaugh-maps. The strategy used in Algorithm $MESH\text{LATIN}$ is to delete columns and rows which contain faulty elements so that the remaining healthy columns and rows can be pasted as a $3 \times 3$ mesh. The pasting operation is to find a linear array for each side of the mesh without including any faulty columns or any faulty rows. In a specific Karnaugh-map illustration of a $2^d$ mesh, the group of nodes in a column or a row is a $d_1$-subcube or a $d_2$-subcube. As described in Chapter II, the group of nodes in a column or a row of a specific Karnaugh-map is represented by a unique $d$-tuple $\{0, 1, x\}^d$. In Algorithm $MESH\text{LATIN}$, we ignore all the free coordinates with the value of “x” and consider the “bound” coordinates only in deleting faulty columns/rows. By considering the “bound” coordinates only, the group of nodes in a column or a row is identified by a unique $d_1$-tuple or a unique $d_2$-tuple respectively. The faulty and healthy nodes of a column/row have the same representation when they are indicated by considering the “bound” coordinates only. Therefore, deleting any one faulty node in a column/row is equivalent to deleting the entire column/row.

Similarly, by considering the “bound” coordinates only, the links between the
Figure 7: Limitation of Algorithm $MESH\text{CUBE}$
Figure 8: A faulty hypercube
groups of nodes in two specific columns or rows in a specific Karnaugh-map are represented by a unique $d_1$-tuple $\{0, 1, z\}^{d_1}$ or a unique $d_2$-tuple $\{0, 1, z\}^{d_2}$, which contains exactly one "z" between the corresponding coordinates, where the $d_1$-tuple or the $d_2$-tuple is the "bound" coordinates for indicating the row side or the column side of the Karnaugh-map. The faulty and healthy links between the groups of nodes in two specific columns or rows have the same representation when they are identified by the "bound" coordinates only. Therefore, deleting any one faulty link between the groups of nodes in two specific columns/rows is equivalent to deleting all the links between the groups of nodes in two specific columns/rows. In addition, if the faulty links are represented by the "bound" coordinates which indicate the column/row side of the mesh, then the faulty links without a "z" in the simplified representation and the faulty links connected to the deleted columns/rows can be ignored for constructing that side of the required mesh. As shown in Fig. 8 and Fig. 9, the faulty links without a "z" in the new representation for the column/row side of the mesh cannot break any adjacent groups of that side. Therefore, they are ignored for constructing that side of the mesh. The deleted columns/rows have broken all the links connected to them; therefore, it is unnecessary to consider the faulty links connected to them.

Algorithm **MESHLATIN** first chooses $d_1$ coordinates among the $d$ coordinates to construct the $m_1$ side of the $m_1 \times m_2$ mesh, where $d_1 = \lceil \log_2 m_1 \rceil$, and then constructs the $m_2$ side of the mesh by using the remaining $(d-d_1)$ coordinates. The coordinates of the $d_1$-tuple and the coordinates of the $(d-d_1)$-tuple are selected among the coordinates of the $d$-tuple exclusively. Faulty elements are handled by considering the "bound" coordinates only. Pasting the healthy columns/rows for constructing either side of the mesh is equivalent to embedding a linear array $L_{m_1}$
Figure 9: Possible K-maps of the mesh embedding example
or $L_{m_2}$ in a faulty $d_1$-subcube or a faulty $(d-d_1)$-subcube respectively. After the linear arrays are found for both sides of the mesh, the mesh is represented by the 2-D Gray code ordering formed by $L_{m_1}$ and $L_{m_2}$. The steps of the algorithm are given as follows:

**Algorithm** *MesHLatin*:

**Step 1:** If $(d - d_1) < d_2$ or $(2^d - r) < (m_1 m_2)$, then exit.

**Step 2:** Apply Lemma 2. If the necessary condition is not satisfied, then exit.

**Step 3:** Calculate the number of extra rows $N_e = (2^{d_1} - m_1)$, that can be deleted for forming the $m_1$ side of the mesh.

**Step 4:** Choose $d_1$ coordinates among the $d$ coordinates to construct the $m_1$ side of the mesh, then represent the faulty elements by the $d_1$ coordinates only and ignore the faulty links without a “z” in the new representation, delete identical representations of the faulty nodes in the $d_1$ coordinates.

**Step 5:** If $r > N_e$ then choose $N_e$ nodes from the fault list $F$, else let $N_e = r$ and choose all of the $r$ faulty nodes. The fault list $F_{m_1}$ for constructing the $m_1$ side is generated by including the $N_e$ nodes and the faulty links obtained in Step 4; The $N_e$ nodes in $F_{m_1}$ represented in the $d_1$ coordinates correspond to the rows to be deleted.

**Step 6:** Construct $F_{m_2}$, the fault list for constructing the $m_2$ side by deleting $F_{m_1}$ in $F$, then represent the faulty elements by the $(d-d_1)$ coordinates only, and ignore the faulty links without a “z” in the new representation, and calculate the number of healthy columns $N_c = 2^{d_2} - \{\text{No. of elements in } F_{m_2}\}$. If $N_c < m_2$, then go to Step 10.
Step 7: Based on $F_{m_1}$ obtained in Step 5, apply Algorithm LNRLATIN to find $L_{m_1}$. If not found, go to Step 10.

Step 8: Based on $F_{m_2}$ obtained in Step 6, apply Algorithm LNRLATIN to find $L_{m_2}$. If not found, go to Step 10.

Step 9: Represent the required mesh by the 2-D Gray code ordering formed by the Cartesian product $L_{m_1} \times L_{m_2}$.

Step 10: Check if all the possible choices of $N_e$ nodes are used exhaustively in Step 5. If yes, then go to Step 11, else go to Step 5 for the next trial.

Step 11: Check if all the combinations of choosing $d_1$ coordinates among the $d$ coordinates are examined exhaustively. If yes, then exit, else go to Step 4 for the next trial.

Step 1 checks if the mesh can be embedded in a healthy $d$-cube and if there are sufficient number of healthy nodes to construct the required mesh. Lemma 2 is applied in Step 2. Steps 3-5 prepare the necessary representations of faulty elements and select the faulty rows and/or links to be deleted for forming the linear array of the $m_1$ side of the mesh. Similarly, Step 6 prepares the necessary representations of faulty elements and updates the fault list for forming the linear array of the $m_2$ side of the mesh. In addition, Step 6 checks if there are sufficient number of healthy columns for constructing the $m_2$ side of the mesh. Steps 7 and 8 apply Algorithm LNRLATIN to find the required linear arrays. After the linear arrays are found, the required mesh is represented by the 2-D Gray code ordering formed by $L_{m_1}$ and $L_{m_2}$. In the worst case, all the possible trials of choosing coordinates for each side of the mesh and all the possible ways of deleting the faulty elements will be performed.
4.4.2 An Embedding Example by Algorithm *MESHLATIN*

An embedding example by Algorithm *MESHLATIN* will be presented in this section. Consider a d-cube with \( r \) faulty elements (\( f_1 = 0001, f_2 = 0101, f_3 = 1100, \) and \( L_1 = 00Z1 \)) as shown in Fig. 10, and an \( m_1 \times m_2 \) mesh is desired to be embedded in it, where \( d=4, m_1=3, m_2=3, r=3 \) and 4-tuple \( = (b_3 b_2 b_1 b_0) \).

![Diagram of a d-cube with faulty elements and a 3 x 3 mesh](image)

Figure 10: An example of embedding a 3 x 3 mesh
The steps of embedding are listed as follows:

**Step 1:** If \( (d - \lfloor \log_2 m_1 \rfloor) < \lfloor \log_2 m_2 \rfloor \) or \( (2^d - r) < (m_1 \times m_2) \)?

- \( d=4, \ m_1=3, \) and \( m_2=3, \)

  then \( (d - \lfloor \log_2 m_1 \rfloor) = (4 - \lfloor \log_2 3 \rfloor) = 4 - 2 = 2 = \lfloor \log_2 m_2 \rfloor = \lfloor \log_2 3 \rfloor, \)

  and \( (2^d - r) = 2^4 - 3 = 16 - 3 = 13 > (m_1 \times m_2) = 9. \)

Therefore, the mesh can be embedded in a healthy \( d \)-cube and the number of healthy nodes is enough for the required mesh.

**Step 2:** \( m_1 m_2 = 3 \times 3 = 9 \) is an odd number. \( 2^d - r - m_1 m_2 = 16 - 3 - 9 = 4, \ f_1=0001 \) (odd parity), \( f_2=0101 \) (even parity), \( f_3=1100 \) (even parity).

\( |N_{\text{even}} - N_{\text{odd}}| = 1 < (2^d - r - m_1 m_2). \)

Therefore, it is possible to have a valid embedding.

**Step 3:** The number of extra rows which can be deleted for constructing the \( m_1 \) side of the mesh. \( N_e = (2^d_1 - m_1) = 4 - 3 = 1. \)

**Step 4:** \( m_1=3, \) and \( d_1 = \lfloor \log_2 m_1 \rfloor = 2. \) Choose 2 coordinates among \( b_3 b_2 b_1 b_0. \)

There are \( \binom{4}{2} \) ways to choose them. Let’s choose \( b_3 b_2 \) first and represent the faulty elements by considering \( b_3 b_2 \) only. The operation is listed as below:

<table>
<thead>
<tr>
<th>Fault List:</th>
<th>( b_3 \ b_2 \ b_1 \ b_0 )</th>
<th>( \rightarrow \ b_3 \ b_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1 )</td>
<td>0 0 0 1</td>
<td>( \rightarrow \ 0 0 )</td>
</tr>
<tr>
<td>( f_2 )</td>
<td>0 1 0 1</td>
<td>( \rightarrow \ 0 1 )</td>
</tr>
<tr>
<td>( f_3 )</td>
<td>1 1 0 0</td>
<td>( \rightarrow \ 1 1 )</td>
</tr>
<tr>
<td>( L_1 )</td>
<td>0 0 z 1</td>
<td>( \rightarrow \ 0 0 )</td>
</tr>
</tbody>
</table>

Since faulty link \( L_1 \) has no “z” in the new representation, it is neglected. It means that the faulty link does not affect the construction of the \( m_1 \) side of the mesh when \( b_3 b_2 \) coordinates are selected.
Step 5: Since there are 3 faulty nodes and one extra node can be deleted, we have \( \binom{3}{1} \) ways to do it. Let's try to delete \( f_1 \) first and generate the fault list for constructing the \( m_1 \) side, denoted as \( F_{m_1} = \{ f_1, L_1 \} = \{00,00 \} = \{00 \}. \) Since \( L_1 = f_1, L_1 \) is deleted when \( f_1 \) is deleted.

Step 6: Update the fault list for constructing the \( m_2 \) side by deleting the fault elements in \( F_{m_1} \), then represent the fault list in \( b_1b_0 \), denoted as \( F_{m_2} = \{01, 00 \}. \) The operation is illustrated as follows:

\[
\begin{array}{cccccc}
\text{Fault List:} & b_3 & b_2 & b_1 & b_0 & b_1 & b_0 \\
f_1 = 0 & 0 & 0 & 1 & & \leftarrow \text{Deleted} \\
f_2 = 0 & 1 & 0 & 1 & \rightarrow & 0 & 1 \\
f_3 = 1 & 1 & 0 & 0 & \rightarrow & 0 & 0 \\
L_1 = 0 & 0 & z & 1 & \leftarrow \text{Deleted} \\
\end{array}
\]

The number of healthy columns = 2 < \( m_2 \). We should go back to step 5 and try to delete \( f_2 \) instead of \( f_1 \).

Step 5(trial 2): Delete \( f_2 \) and generate the fault list \( F_{m_1} = \{ f_2 \} = \{01 \}. \)

Step 6(trial 2): Update the fault list for constructing the \( m_2 \) side by deleting the fault elements in \( F_{m_1} \), then represent the fault list in \( b_1b_0 \), denoted as \( F_{m_2} = \{01, 00, z1 \}. \) The operation is illustrated as follows:

\[
\begin{array}{cccccc}
\text{Fault List:} & b_3 & b_2 & b_1 & b_0 & b_1 & b_0 \\
f_1 = 0 & 0 & 0 & 1 & \rightarrow & 0 & 1 \\
f_2 = 0 & 1 & 0 & 1 & \leftarrow \text{Deleted} \\
f_3 = 1 & 1 & 0 & 0 & \rightarrow & 0 & 0 \\
L_1 = 0 & 0 & z & 1 & \rightarrow & z & 1 \\
\end{array}
\]

The number of healthy columns = 2 < \( m_2 \). We should go back to step 5 and
try to delete $f_3$ instead of $f_1$ or $f_2$.

**Step 5 (trial 3):** Delete $f_3$ and generate the fault list $F_{m_1} = \{f_3\} = \{11\}$.

**Step 6 (trial 3):** Update the fault list for constructing the $m_2$ side by deleting the fault elements in $F_{m_1}$, then represent the fault list in $b_1b_0$, denoted as $F_{m_2}$. The operation is illustrated as follows:

<table>
<thead>
<tr>
<th>Fault List:</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 = 0 \quad 0 \quad 0 \quad 1 \quad \rightarrow \quad 0 \quad 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_2 = 0 \quad 1 \quad 0 \quad 1 \quad \rightarrow \quad 0 \quad 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_3 = 1 \quad 1 \quad 0 \quad 0 \quad \leftarrow \quad $ Deleted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_1 = 0 \quad 0 \quad z \quad 1 \quad \rightarrow \quad z \quad 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The number of healthy columns = 3 = $m_2$ and $F_{m_2} = \{01, z1\}$.

**Step 7:** Apply Algorithm $LNRLATIN$ to embed a linear array in a faulty 2-cube with $F_{m_1} = \{11\}$. The procedure is listed as below.

1. $m_1 = 3$ is an odd number, $|N_{even} - N_{odd}| = 1$, $(2^d - r - m_1 + 1) = 2^2 - 1 - 3 + 1 = 1$. Since $|N_{even} - N_{odd}| \leq (2^d - r - m_1 + 1)$, it is possible to have a valid embedding.

2. Adjacency list: $00$-{01,10}, 01-{00,11}, 10-{00,11}, 11-{01,10}.

3. Update the adjacency list by deleting $F_{m_1}$. The new adjacency list: $00$-{01,10}, 01-{00}, 10-{00}.

4. Let the start node = 00 and create the linear arrays of length 2. They are 00 - 01 and 00 - 10. The number of arrays = 2.

5. Examine each node in the adjacency list of the last node of $L_{m_1}(tem1)$. The last node is 01 and its adjacency list is {00}. Since 00 is a node
of $L_{m_1}(tem1)$, we cannot append 00 after 01 to create a linear array of length 3. Similarly, $L_{m_1}(tem2)$ cannot be appended to any nodes for creating a linear array of length 3. Therefore, we should use the other node as the start node since the number of arrays $= 0$.

6. Let the start node $= 01$ and create the linear arrays of length 2.
$L_{m_1}(tem1) = \{01, 00\}$. The number of arrays $= 1$.

7. Examine each node in the adjacency list of the last node of $L_{m_1}(tem1)$.
The last node is 00 and its adjacency list is $\{01, 10\}$. Since 01 is a node of $L_{m_1}(tem1)$, we cannot append 01 after 00 to create a linear array of length 3. However, 10 can be appended after 01 to create a linear array of length 3. Hence, the linear array found is $L_{m_1} = \{01, 00, 10\}$.

**Step 8:** Apply Algorithm $LNRLATIN$ to embed a linear array in an injured 2-cube with $F_{m_2} = \{01, z1\}$. The procedure is listed as below.

1. $m_2 = 3$ is an odd number, $|N_{even} - N_{odd}| = 1$, $(2^d - r - m_2 + 1) = 2^2 - 1 - 3 + 1 = 1$. Since $|N_{even} - N_{odd}| \leq (2^d - r - m_2 + 1)$, it is possible to have a valid embedding.

2. Adjacency list: 00-$\{01, 10\}$, 01-$\{00, 11\}$, 10-$\{00, 11\}$, 11-$\{01, 10\}$.

3. Update the adjacency list by deleting $F_{m_2}$. The new adjacency list:
00-$\{10\}$, 10-$\{00, 11\}$, 11-$\{10\}$.

4. Let the start node $= 00$ and create the linear arrays of length 2.
$L_{m_2}(tem1) = \{00, 10\}$. The number of arrays $= 1$.

5. Examine each node in the adjacency list of the last node of $L_{m_2}(tem1)$.
The last node is 10 and its adjacency list is $\{00, 11\}$. Since 00 is a node of $L_{m_2}(tem1)$, we cannot append 00 after 10 to create a linear array of
length 3. However, 11 can be appended after 10 to create a linear array of length 3. Hence, the linear array found is $L_{m_2} = \{00, 10, 11\}$.

**Step 9:** Represent the required mesh by the 2-D Gray code ordering formed by $L_{m_1}$ and $L_{m_2}$.

The mesh embedding process stated above is illustrated in Fig. 11.

4.4.3 The Time Complexity of Algorithm *MESHLATIN*

The complexity of Algorithm *MESHLATIN* is analyzed as follows. Step 1 can be finished in constant time by table-look-up, arithmetic and conditional checking operations. Since the complexity of determining the parity of a node is $O(d)$, the complexity of Step 2 is $O(rd)$ provided that $2^d$ is pre-calculated. Step 3 can be done in constant time provided that the applicable values of $2^{d_1}$ are pre-calculated. In the worst case, the complexity in Step 4 is $O\left(\binom{d}{d_1}\right)$ [43]. In Step 4, the complexity of representing the faulty elements by the $d_1$-coordinates is $O(r + l)$ because it takes a constant time for each element. In the worst case, the complexity of selecting $N_e$ rows in Step 5 is $O\left(\binom{r}{N_e}\right)$. Generation of the fault list in Step 5 has a complexity of $O(N_e + l)$. The complexity is $O((r + l)N_e)$ for deleting the faulty elements in the fault list obtained in Step 5. In Step 6, the complexity of representing the faulty elements by the $(d-d_1)$-coordinates is $O(r + l - N_e)$. Let the complexity of Step 7 be $Lnr_{m_1}$ and the complexity of Step 8 be $Lnr_{m_2}$. According to the analysis of Algorithm *LNRLATIN*,

$$Lnr_{m_1} = O\left(2^{d_1}(N_e + l) + N_{cad_1}d_1 \cdot m_1\right)$$

and

$$Lnr_{m_2} = O\left(2^{d_2}(r + l - N_e) + N_{cad_2}d_2 \cdot m_2\right),$$
Illustration of Trial 1:

Illustration of Trial 2:

Illustration of Trial 3:

Figure 11: Illustration of a $3 \times 3$ mesh embedding using Algorithm *MESHLATIN*
where \( N_{cad1} \) and \( N_{cad2} \) are the number of healthy nodes as the start node for constructing \( L_{m1} \) and \( L_{m2} \) respectively. In Step 9, the complexity of constructing the required mesh is \( O(m_1 m_2) \). In addition, Steps 10 and 11 can be done in constant time. Thus, an upper bound for the time complexity of Algorithm \textit{MESHLATIN} can be expressed as:

\[
O \left( \left( \binom{d}{d_1} (r + l + \binom{r}{N_e} (N_e + l + (r + l)N_e + (r + l - N_e) + L_{nrm1} + L_{nrm2}) + m_1 m_2 \right) \right).
\]

Since \( L_{nrm1} > (N_e + l) \), \( L_{nrm2} > (r + l - N_e) \), \( (L_{nrm1} + L_{nrm2}) > (r + l)N_e \), \( L_{nrm2} > (r + l) \), \( \binom{d}{d_1} < 2^d \) and \( m_1 m_2 < 2^d \), the above complexity can be expressed as:

\[
O \left( P \binom{r}{N_e} (L_{nrm1} + L_{nrm2}) \right),
\]

where \( P \) is the number of processors. The above analysis presents a worst-case complexity which may be overly pessimistic. It should be noted that the complexity stated above is one of the upper bounds and a smaller upper bound is expected to be found since the term \( d_l^m \) of the complexity is overestimated.

4.4.4 Implementation and Performance of Algorithm \textit{MESHLATIN}

The performance results of Algorithm \textit{MESHLATIN} implemented in the host computer of the \textit{iPSC/2} hypercube is presented here. The overall performance results for a set of examples are presented in Table 2. The longest and the average execution time are measured for the set of examples. The examples cover the dimensions of the hypercube from \( d = 3 \) to \( d=9 \). They include some possible dimensions of 2-D meshes. It is assumed that the maximum number of faulty nodes is equal to the dimension of the hypercube [24]. All the possible combinations of \( d \) faulty nodes are tested, where \( d < 6 \). For \( d \geq 6 \), the combinations of all possible \( d \) faulty nodes is too large to test in a multiuser environment. Therefore, only 2000
samples are taken for each set of examples when \( d \geq 6 \). From the measurements, it can be seen that Algorithm \( MESHLATIN \) is an efficient algorithm. The worst case we have so far requires 1.16 seconds to embed a \( 15 \times 30 \) mesh in a 9-cube with a particular set of 9 faulty elements. By examining Table 2 carefully, it is clear that the execution time depends on the dimension of hypercube, the mesh size, and the distribution of the faulty nodes. In general, the larger the dimension of hypercube is, the longer the execution time of embedding is. Similarly, if the maximum dimension of the required mesh is larger, the execution time is longer. For example, for \( d = 5 \), the longest execution time for embedding a \( 3 \times 7 \) mesh is 0.03 seconds; while the longest execution time for embedding a \( 2 \times 13 \) mesh is 0.68 seconds.

The performance results shown in Table 2 reveal that the average embedding performance would be acceptable in most applications. Some meshes found by Algorithm \( MESHLATIN \) are presented in Fig. 12. It is believed that Algorithm \( MESHLATIN \) is an efficient approach of mesh embedding for achieving reconfiguration in a faulty hypercube.

4.5 The Parallel Versions of Algorithm \( MESHLATIN \)

Algorithm \( MESHLATIN \) can be parallelized in a faulty hypercube by two approaches. The first approach, namely Algorithm \( PMESHLATIN1 \), distributes the fault list and uses Algorithm \( PLNRLATIN \) in each healthy node to construct both sides of the desired 2-D mesh. The second approach, namely Algorithm \( PMESHLATIN2 \), distributes the partitioned combinations of choosing \( d_1 \) coordinates among the \( d \) coordinates for constructing the \( m_1 \) side of the mesh to each healthy node and uses Algorithm \( MESHLATIN \) in each healthy node. In the first approach, once the desired linear array is found by any healthy node, it is
### Table 1: Performance Results of Algorithm *MESHLATIN*

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Mesh Size</th>
<th>Faulty Elements</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Longest</td>
</tr>
<tr>
<td>d=3</td>
<td>2 x 3</td>
<td>Any 2 faults</td>
<td>0.01 sec</td>
</tr>
<tr>
<td>d=4</td>
<td>2 x 5</td>
<td>Any 4 faults</td>
<td>0.01 sec</td>
</tr>
<tr>
<td>d=4</td>
<td>3 x 3</td>
<td>Any 4 faults</td>
<td>0.02 sec</td>
</tr>
<tr>
<td>d=4</td>
<td>3 x 4</td>
<td>Any 4 faults</td>
<td>0.02 sec</td>
</tr>
<tr>
<td>d=5</td>
<td>2 x 13</td>
<td>Any 5 faults</td>
<td>0.68 sec</td>
</tr>
<tr>
<td>d=5</td>
<td>3 x 7</td>
<td>Any 5 faults</td>
<td>0.03 sec</td>
</tr>
<tr>
<td>d=5</td>
<td>4 x 4</td>
<td>Any 5 faults</td>
<td>0.03 sec</td>
</tr>
<tr>
<td>d=6</td>
<td>7 x 7</td>
<td>Any 6 faults*</td>
<td>0.06 sec</td>
</tr>
<tr>
<td>d=6</td>
<td>4 x 14</td>
<td>Any 6 faults*</td>
<td>0.08 sec</td>
</tr>
<tr>
<td>d=7</td>
<td>7 x 15</td>
<td>Any 7 faults*</td>
<td>0.16 sec</td>
</tr>
<tr>
<td>d=8</td>
<td>15 x 15</td>
<td>Any 8 faults*</td>
<td>0.34 sec</td>
</tr>
<tr>
<td>d=9</td>
<td>15 x 30</td>
<td>Any 9 faults*</td>
<td>1.16 sec</td>
</tr>
</tbody>
</table>

Note: * 2000 samples are taken for the test.
Note: \( F \) = fault list, \( d \) = dimension of hypercube.

Figure 12: Some meshes found by Algorithm \textit{MESHLATIN}
sent to the host computer and the processes executing in all the healthy nodes are killed. If all the healthy nodes cannot find the desired linear array, the linear array embedding operation is also terminated. The parallel version of embedding linear array is applied to construct both sides of the desired 2-D mesh. In the second approach, once the desired 2-D mesh is found by any healthy node, it is sent to the host computer and the processes executing in all the healthy nodes are killed. If all the healthy nodes cannot find the desired 2-D mesh, the 2-D mesh embedding operation is also terminated.

Algorithm *PMESHLATIN1*:

**Steps 1-6:** Steps 1-6 of Algorithm *MESHLATIN* are executed on the host computer.

**Step 7:** Based on the fault list obtained in Step 5, distribute the fault list and apply Algorithm *PLNRLATIN* in each healthy node to find \( L_{m_1} \). If not found, go to Step 10.

**Step 8:** Based on the fault list obtained in Step 6, distribute the fault list and apply Algorithm *PLNRLATIN* in each healthy node to find \( L_{m_2} \). If not found, go to Step 10.

**Steps 9-11:** Steps 9-11 of Algorithm *MESHLATIN* are executed on the host computer.

Algorithm *PLNRLATIN* is applied in each healthy node to construct both sides of the desired 2-D mesh. Consequently, the time complexity of Algorithm *PMESHLATIN1* is better than that of Algorithm *MESHLATIN*. Let \( L_{nrp_{m_1}} \) indicate the complexity of Step 7 and \( L_{nrp_{m_2}} \) indicate the complexity of Step 8.
According to the analysis of Algorithm PLNRLATIN, we have

\[ L_{nrp_m1} = O \left( 2^{d_1}(N_e + l) + d_1^{m_1} \right) \]

and

\[ L_{nrp_m2} = O \left( 2^{d_2}(r + l - N_e) + d_2^{m_2} \right) . \]

Thus, an upper bound for the time complexity of Algorithm PMESHLATIN1 can be expressed as:

\[ O \left( P \left( \frac{r}{N_e} \right) (L_{nrp_m1} + L_{nrp_m2}) \right) . \]

Algorithm PMESHLATIN2:

**Steps 1-3:** Steps 1-3 of Algorithm MESHLATIN are executed on the host computer.

**Step 4:** The host computer generates all the combinations of choosing \( d_1 \) coordinates among the \( d \) coordinates to construct the \( m_1 \) side of the mesh, then distributes the partitioned combinations and \( N_e \) to each healthy node.

**Step 5:** Each healthy node performs the following steps.

**Step 5.1:** Choose \( d_1 \) coordinates among the \( d \) coordinates from the partitioned combinations to construct the \( m_1 \) side of the mesh, then represent the faulty elements by the \( d_1 \) coordinates only and ignore the faulty links without a "z" in the new representation.

**Step 5.2:** If \( r > N_e \) then choose \( N_e \) nodes from the fault list, else let \( N_e = r \) and choose all the \( r \) faulty nodes. The fault list for constructing the \( m_1 \) side is generated by including the \( N_e \) nodes and the faulty links obtained in Step 5.1, but deleting the faulty links with the same new representations as any selected \( N_e \) nodes.
Step 5.3: Update the fault list for constructing the $m_2$ side by deleting the faulty elements in the fault list obtained in Step 5.2, then represent the faulty elements by the $(d-d_1)$ coordinates only, and ignore the faulty links without a "z" in the new representation, and calculate the number of healthy columns $N_c$. If $N_c < m_2$, then go to Step 5.7.

Step 5.4: Based on the fault list obtained in Step 5.2, apply Algorithm LNRLATIN in each healthy node to find $L_{m_1}$ and send it to the host computer. If not found, go to Step 5.6.

Step 5.5: Based on the fault list obtained in Step 5.3, apply Algorithm LNRLATIN in each healthy node to find $L_{m_2}$ and send it to the host computer. If not found, go to Step 5.6.

Step 5.6: Check if all the possible choices of $N_c$ nodes are used exhaustively in Step 5.2. If no, go to Step 5.2 for the next trial.

Step 5.7: Check if all the partitioned combinations of choosing $d_1$ coordinates among the $d$ coordinates are examined exhaustively. If yes, then inform the host computer that the desired linear arrays cannot be found, else go to Step 5.1 for the next trial.

Step 6: After receiving the desired linear arrays, the host computer represents the required mesh by the 2-D Gray code ordering formed by the Cartesian product $L_{m_1} \times L_{m_2}$ and kills the processes executing in each healthy node. Additionally, if all the healthy nodes cannot find the desired linear arrays, the host computer also kills the processes executing in each healthy node.

In Algorithm $PMESHALTIN2$, all the combinations of choosing $d_1$ coordinates among the $d$ coordinates to construct the $m_1$ side of the mesh are partitioned and
distributed to each healthy nodes, then each healthy node performs the partitioned task to find both sides of the desired 2-D mesh. Consequently, the time complexity of Algorithm PMESHLATIN2 is better than that of Algorithm MESHLATIN. Let $P$ indicate the number of healthy nodes used in the faulty hypercube, $Lnr_{m_1}$ indicate the complexity of Step 5.4; and $Lnr_{m_2}$ indicate the complexity of Step 5.5. According to the analysis of Algorithm PLNRLATIN, we have

$$Lnr_{m_1} = O\left(2^{d_1}(N_e + l) + d_1^{m_1}\right)$$

and

$$Lnr_{m_2} = O\left(2^{d_2}(r + l - N_e) + d_2^{m_2}\right).$$

Thus, an upper bound for the time complexity of Algorithm PMESHLATIN2 can be expressed as:

$$O\left(\frac{d}{P^r} \left(\binom{r}{N_e}(Lnr_{m_1} + Lnr_{m_2})\right)\right),$$

or

$$O\left(\frac{P}{P^r} \left(\binom{r}{N_e}(Lnr_{m_1} + Lnr_{m_2})\right)\right)$$

since $P > \binom{d}{d_1}$.

4.6 Description of Algorithm MESHDCC

A more efficient algorithm for embedding 2-D meshes in a faulty hypercube with Direct-Connect Capability is presented in this section. In Algorithm MESHDCC, it is assumed that the links and the Direct-Connect Hardware (DCH's) of the hypercube that has the Direct-Connect Capability are fault-free, and the faulty hypercube has faulty processors only. In addition, Algorithm BY-PASS executed by the DCH's of each node uses an e-cube-like algorithm to enable the local communication between columns and/or rows in the embedded mesh exactly via the DCH's of the removed faulty columns/rows.
Algorithm \textit{MESHDCC} first embeds a $2^{d_1} \times 2^{d_2}$ mesh represented by a 2-D Gray code ordering in a faulty hypercube, where $d_1 \geq \lceil \log_2 m_1 \rceil$, $d_2 \geq \lceil \log_2 m_2 \rceil$, and $(d_1 + d_2) \leq d$, then removes faulty columns and/or rows. Finally, it forms the desired $m_1 \times m_2$ mesh from the healthy columns and rows directly. Since the $m_1$ side and the $m_2$ side of the mesh are formed directly without considering if they form a 2-D Gray code, the mesh embedding is much easier than the other methods which try to find a 2-D Gray code ordering in a faulty hypercube.

An example of embedding a $5 \times 5$ mesh in a faulty 6-cube is illustrated in Fig. 13. An initial embedding of an $8 \times 8$ mesh shown in Fig. 13.(a) is made by simply generating a 2-D Gray code for the coordinates $656463$ and $626160$. Finally, a possible final embedding of a $5 \times 5$ mesh is obtained by removing the faulty columns and/or rows as shown in Fig. 13.(b). According to the e-cube algorithm \cite{38,41}, we know that communication from the nodes of subcube $xxx001$ (column 001) to the nodes of subcube $xxx110$ (column 110) will go through the DCH's of subcube $xxx000$ and subcube $xxx010$; similarly, communication from the nodes of subcube $xxx110$ (column 110) to the nodes of subcube $xxx001$ (column 001) will go through the DCH's of subcube $xxx111$ and subcube $xxx101$. Consequently, the local communication associated with subcube $xxx111$ (column 111) or subcube $xxx000$ (column 000) is interrupted.

Algorithm \textit{BYPASS} uses the properties of the \textit{Coordinate sequence} and the identifications of the source and the destination to determine a unique predefined path. The path will only include some DCH's of the nodes between the source and the destination in a linear array embedded by making use of the \textit{Coordinate sequence} approach. In our example shown in Fig. 13(b), Algorithm \textit{BYPASS} ensures the communication between the nodes of subcube $xxx001$ (column 001) and the
Figure 13: A 5×5 mesh embedded in a 6-cube using DCC approach  
(a) An initial embedding (b) The final embedding
nodes of subcube \textit{xxx}110 (\textit{column} 110) exactly via the DCH's of subcube \textit{xxxx}011 and subcube \textit{xxxx}010. The required parameters used in Algorithm \textit{BYPASS} for determining the communication path are the nearest $k$ coordinate transitions around the source node and the (source $\oplus$ destination), where $k$ is a specified range determined by how many consecutive failed nodes to be passed and $\oplus$ indicates the logic Exclusive-OR operation. In this research, $k$ is specified as 4 in order to bypass 1-8 consecutive failed nodes between any nodes in a linear array. When more consecutive nodes are required to be bypassed, the required range of $k$ can be determined by a similar analysis.

\subsection*{4.6.1 Algorithm \textit{BYPASS}}

Algorithm \textit{BYPASS} executed in the DCH of each node uses a special \textit{e-cube}-like routing algorithm to replace the \textit{e-cube} algorithm used in [41]. It provides a better performance of local communication in a linear array or a 2-D mesh embedded in a faulty hypercube than the \textit{e-cube} algorithm does. Algorithm \textit{BYPASS} uses the properties of the \textit{Coordinate sequence} and the identifications of the source and the destination to determine a unique predefined path. The path will only include some DCH's of the nodes between the source and the destination in a linear array embedded by making use of the \textit{Coordinate sequence} approach. The required parameters used in Algorithm \textit{BYPASS} for determining the communication path are the nearest $k$ coordinate transitions around the source node and (source $\oplus$ destination), where $k$ is a specified range determined by the number of consecutive failed nodes to be bypassed and $\oplus$ indicates the logic Exclusive-OR operation.

\subsection*{4.6.2 Properties of Coordinate Sequence}

Consider a coordinate sequence $S_d$ generated by Algorithm \textit{GCSQ}, where \[ S_d = \{0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,5,\ldots \} \]
0, 1, 0, 2, 0, 0, 2, 0, 1, 0}. By examining \( S_d \), the properties of the coordinate sequence associated with the linear array embedded can be obtained as follows:

**Property 1:** A coordinate transition \( i \) of a coordinate sequence \( S_d \) repeats every \( 2^{(i+1)} - 1 \) elements, where \( i = 0, 1, \cdots, d - 1 \).

For example, 0 repeats every other element, 1 repeats every 3 elements, 2 repeats every 7 elements, 3 repeats every 15 elements, and so on.

**Property 2:** Any 2 consecutive coordinate transitions of a sequence \( S_d \) always contain a 0 and an integer between 1 and \( d - 1 \).

**Property 3:** A coordinate transition \( i \) of a coordinate sequence \( S_d \) is greater than the previous and the next \( 2^i - 1 \) elements, where \( i = 0, 1, \cdots, d - 1 \).

For example, 1 is greater than the previous and the next 1 element, 2 is greater than the previous and the next 3 elements, 3 is greater than the previous and the next 7 elements, and so on.

**Property 4:** Any \( n \) consecutive elements of a coordinate sequence \( S_d \) contain \( \lfloor \frac{n}{2^i+1} \rfloor \) or \( \lceil \frac{n}{2^i+1} \rceil \) elements of value \( i \), where \( n > 1 \) and \( i = 0, 1, \cdots, d - 1 \).

For example, \( n = 2 \) contains \( \lfloor \frac{2}{2^0+1} \rfloor = 1 = \lfloor \frac{2}{2^0+1} \rfloor \) element of value 0, and contains \( \lceil \frac{2}{2^1+1} \rceil = 1 \) or \( \lfloor \frac{2}{2^1+1} \rfloor = 0 \) element of value \( x \geq 1 \). \( n = 3 \) contains \( \lfloor \frac{3}{2^0+1} \rfloor = 2 \) or \( \lceil \frac{3}{2^0+1} \rceil = 1 \) elements of value 0, and contains \( \lfloor \frac{3}{2^2+1} \rfloor = 1 \) or \( \lfloor \frac{3}{2^2+1} \rfloor = 0 \) element of value \( x \geq 1 \).

**Property 5:** Any \( n \) consecutive coordinate transitions of a coordinate sequence \( S_d \) at least contain an element of value \( \geq i \), where \( i \) is the minimum value which makes \( \lfloor \frac{n}{2^i+1} \rfloor = 0 \).

Since \( i \) is the minimum value which makes \( \lfloor \frac{n}{2^i+1} \rfloor = 0 \), we have \( 2^i \leq n < 2^{i+1} \). Assuming \( i - 1 \) is the maximum value \( \lfloor \frac{n}{2^{i-1}+1} \rfloor = 0 \), we have \( 2^i \leq n < 2^{i+1} \).
transitions, we can prove that \( n \) is out of range. If \( i - 1 \) is the maximum value in the \( n \) consecutive coordinate transitions, then the maximum value of \( n \) is \((2^{i-1} - 1 + 2^{i-1} - 1 + 1)\) by using the fact of Property 3. Thus, we have \( n < 2^i - 1 \). This is a contradiction since it cannot be that \( 2^i < n < 2^i - 1 \).

**Property 6**: (Mirror property) Consider the communication path between two nodes located \( n - 1 \) nodes apart in a linear array embedded by making use of the Coordinate sequence. The \( n \) consecutive coordinate transitions of a sequence \( S_d \) for the communication path can be reduced to one transition \( c \) if \( n \) is an odd number, \( c \) is the \( (\frac{n}{2} + 1) \)-th element of the \( n \) transitions, and \( c \) is the maximum element.

For example, for \( n = 3 \), 1 is the middle element of \( \{0, 1, 0\} \) and 1 is the maximum one, thus \( \{0, 1, 0\} \) can be reduced to \( \{1\} \). For \( n = 7 \), 2 is the middle element of \( \{0, 1, 0, 2, 0, 1, 0\} \) and 2 is the maximum one, thus, \( \{0, 1, 0, 2, 0, 1, 0\} \) can be reduced to \( \{2\} \). As long as a mirror reflection happens on an element of an \( n \) consecutive coordinate transitions, the whole mirror reflection can be removed.

### 4.6.3 Derivation of Algorithm **BYPASS**

We shall apply the properties of the Coordinate sequence to analyze the relationship between the nodes in a linear array embedded in a hypercube. The result of analysis will be used in Algorithm **BYPASS** to force the predefined communication path between the nodes in a linear array include only the DCH's of some nodes between them. The following variables are defined to be used in describing Algorithm **BYPASS**.

**SRC, DST** = Binary representation of the source, and the destination.
PROBE = SRC ⊕ DST, where ⊕ indicates the logic Exclusive-OR operation.

\[ P = [p_0, p_1, \ldots, p_{p_0}] \], where \( p_0 \) is the number of 1's in PROBE, and \( p_1, \ldots, p_{p_0} \) are the coordinates with 1's of PROBE in the descending order.

\[ D = [d_1, d_2, d_3, d_4] \], where \( d_1 \) is the non-zero coordinate of the coordinate transitions which transfer the source node to the first nearest neighbors, \( d_2 \) is the non-zero coordinate of the coordinate transitions which transfer the first nearest neighbors to the second nearest neighbors, \( d_3 \) is the non-zero coordinate of the coordinate transitions which transfer the second nearest neighbors to the third nearest neighbors, and \( d_4 \) is the non-zero coordinate of the coordinate transitions which transfer the third nearest neighbors to the fourth nearest neighbors. If the non-zero coordinate does not exist, it is ignored. For example, Fig. 14 shows that \( D \) is \([2, 1, 1, 3]\) for the source node 0110 and \( D \) is \([-1, 1, -1, 2]\) for the source node 0000.

\[ T = [t_1, \ldots, t_{p_0}] \], the coordinate sequence which enables the message routing in a linear array by passing through the DCH's of some nodes between the source and the destination. It is similar to the coordinate sequence obtained by the e-cube algorithm except that the sequence is reconstructed.

---

Figure 14: An example of embedding a linear array using the Coordinate Sequence

We have analyzed the cases that cover 2-9 transitions normally required to pass 1-8 nodes in a linear array embedded by making use of the Coordinate sequence.
approach. In each analysis, both the left destination and the right destination around a source in a linear array are considered. After the analysis of each case, a rule is obtained to define the desired $T$ by the parameters of $P$ and $D$. Each analysis is presented as follows:

**Case 1:**

To pass 1 node in a linear array requires 2 consecutive transitions normally. According to the property of a coordinate sequence, all the possible pairs of transitions and their associated $D$, $P$, and $T$ for passing a node from a source to the left destination and the right destination in a linear array are listed in Table 3.

<table>
<thead>
<tr>
<th>Pair</th>
<th>Transitions to $DST_{left,right}$</th>
<th>$t_1, t_2$ $left,right$</th>
<th>$d_1, d_2$</th>
<th>$p_0, p_1, p_2$ $left,right$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,1, $n_1$, 0</td>
<td>0,1, $n_1$, 0</td>
<td>$n_1$, 1</td>
<td>2,1,0, $2, n_1$, 0</td>
</tr>
<tr>
<td>2</td>
<td>1,0, 0, $n_2$</td>
<td>1,0, 0, $n_2$</td>
<td>$n_2$</td>
<td>2,1,0, $2, n_2$, 0</td>
</tr>
</tbody>
</table>

where $n_1, n_2 > 1$.

Table 3: The possible pairs of passing 1 node

The left transition ([01] or [10]) of each pair passes a left node from the source to the destination; the right transition ($[n_10]$ or $[0n_2]$) of each pair passes a right node from the source to the destination. Now we shall investigate how $T$ can be determined by using the parameters $d_1, d_2, p_0, p_1$ and $p_2$. The investigation is to find the answers to the following four questions.

1. How to get $[t_1, t_2] = [0, 1]$ based on $d_1 = n_1$, $d_2 = 1$, $p_0 = 2$, $p_1 = 1$ and $p_2 = 0$ ?
2. How to get $[t_1, t_2] = [n_1, 0]$ based on $d_1 = n_1$, $d_2 = 1$, $p_0 = 2$, $p_1 = n_1$ and $p_2 = 0$ ?
3. How to get $[t_1, t_2] = [1, 0]$ based on $d_1 = 1$, $d_2 = n_2$, $p_0 = 2$, $p_1 = 1$ and $p_2 = 0$ ?
4. How to get $[t_1, t_2] = [0, n_2]$ based on $d_1 = 1$, $d_2 = n_2$, $p_0 = 2$, $p_1 = n_2$ and $p_2 = 0$ ?
It is clear that the answers to the above four questions can be combined and summarized as below:

If \( p_0 = 2 \) then if \( p_2 = 0 \) and \( p_1 = d_1 \) then \( T = [p_1, p_2] \)

else \( T = [p_2, p_1] \)

Since it is necessary to consider the other cases of bypassing different numbers of nodes, both \( p_0 \) and \( p_2 \) are required to be checked here. For the following cases, we will only list the possible pairs of transitions and their associated \( D, P, \) and \( T \) for passing nodes from a source to the left destination and the right destination in a linear array. In addition, by carefully investigating how \( T \) can be determined by using the parameters \( d_1, d_2, p_0, p_1, p_2 \) and \( p_3 \), we have a rule for each case.

Case 2:

The possible pairs of passing 2 nodes in a linear array are shown in Table 4:

<table>
<thead>
<tr>
<th>Pair</th>
<th>Transitions to ( DST_{\text{left, right}} )</th>
<th>( T_{\text{left, right}} )</th>
<th>( d_1, d_2 )</th>
<th>( p_0, p_1, \ldots, p_3, \text{left, right} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0, 1, 0, [n_1, 0, 1]</td>
<td>1, p_1 = \text{max}(d_1, d_2)</td>
<td>1, 0</td>
<td>[0, 1, 0, 1, 0]</td>
</tr>
<tr>
<td>2</td>
<td>1, 0, 2, [0, n_2, 0]</td>
<td>1, 0, 2, n_2</td>
<td>1, n_2</td>
<td>[3, 1, 0, 1, n_2]</td>
</tr>
<tr>
<td>3</td>
<td>0, 2, 0, [1, 0, n_3]</td>
<td>2, 1, 0, n_3</td>
<td>1, 2</td>
<td>[1, 2, 0, n_3, 1, 0]</td>
</tr>
</tbody>
</table>

where \( n_1 > 1 \) and \( n_2, n_3 > 2 \).

Table 4: The possible pairs of passing 2 nodes

The rule is:

if \( p_0 = 1 \) then \( T = [p_1] \)

if \( p_0 = 3 \) then if \( p_2 = 1 \) and \( p_1 = \text{max}(d_1, d_2) \) then \( T = [p_1, p_3, p_2] \)

else \( T = [p_2, p_3, p_1] \)

Case 3:

The possible pairs of passing 3 nodes in a linear array are shown in Table 5.
Table 5: The possible pairs of passing 3 nodes

The rule is:

\[
\text{if } (p_0 = 2) \text{ then if } (p_2 = 1 \text{ and } p_1 = \max(d_1, d_2)) \text{ then } T = [p_1, p_2] \\
\text{else } T = [p_2, p_1]
\]

Case 4:

The possible pairs of passing 4 nodes in a linear array are shown in Table 6.

Table 6: The possible pairs of passing 4 nodes

The rule is:

\[
\text{if } (p_0 = 1) \text{ then } T = [p_1] \\
\text{if } (p_0 = 3) \text{ then if } (p_2 = 1 \text{ and } p_1 = \max(d_1, d_2)) \text{ or } (p_2 > 1 \text{ and } p_1 = \max(d_1, d_2, d_3, d_4)) \text{ then } T = [p_1, p_3, p_2] \\
\text{else } T = [p_2, p_3, p_1]
\]

Case 5:

The possible pairs of passing 5 nodes in a linear array are listed in Table 7.
Table 7: The possible pairs of passing 5 nodes

The rule is:

\[
\text{if} \ (p_0 = 2) \ \text{then} \ if \ (p_2 = 0 \ \text{and} \ p_1 \neq 1 \ \text{and} \ p_1 = d_3) \ \text{then} \ T = [p_1, p_2]
\]

\[
\text{else} \ T = [p_2, p_1]
\]

\[
\text{if} \ (p_0 = 4 \ \text{and} \ p_1 = \text{max}(d_1, d_2)) \ \text{then} \ T = [p_1, p_4, p_3, p_2]
\]

\[
\text{else} \ T = [p_2, p_3, p_4, p_1]
\]

**Case 6:**

The possible pairs of passing 6 nodes in a linear array are listed in Table 8.

Table 8: The possible pairs of passing 6 nodes

The rule is:

\[
\text{if} \ (p_0 = 1) \ \text{then} \ T = [p_1]
\]

\[
\text{if} \ (p_0 = 3) \ \text{then} \ if \ (p_2 > 1 \ \text{and} \ p_1 = \text{max}(d_1, d_2, d_3, d_4)) \ \text{then} \ T = [p_1, p_3, p_2]
\]

\[
\text{else} \ T = [p_2, p_3, p_1]
\]

**Case 7:**

The possible pairs of passing 7 nodes in a linear array are listed in Table 9.
Table 9: The possible pairs of passing 7 nodes

The rule is:

\[
\text{if } (p_0 = 2) \text{ then if } (p_2 = 2 \text{ and } p_1 = \max(d_1, d_2, d_3, d_4)) \text{ then } T = [p_1, p_2] \\
\text{else } T = [p_2, p_1]
\]

Case 8:

The possible pairs of passing 8 nodes in a linear array are listed in Table 10.

<table>
<thead>
<tr>
<th>Pair</th>
<th>Transitions to (DST_{\text{left, right}})</th>
<th>(I_{\text{left, right}})</th>
<th>(d_1, d_2, d_3, d_4)</th>
<th>(P_0, P_1, \ldots, P_8)_{\text{left, right}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,1,0,2,0,1,0,3,0,1,0,2,0,1,0</td>
<td>2,0,3,0,1,2,3</td>
<td>n_1,1,1,2</td>
<td>3,3,2,0,1,3,3,2</td>
</tr>
<tr>
<td>2</td>
<td>1,0,2,0,1,0,3,0,0,2,0,1,0</td>
<td>2,1,3,0,2,3</td>
<td>n_2,1,2,1</td>
<td>3,3,2,1,3,3,2</td>
</tr>
<tr>
<td>3</td>
<td>0,2,0,1,0,3,0,1,1,0,2,0,1,0</td>
<td>2,0,3,0,1,2,3</td>
<td>n_3,1,2,1</td>
<td>3,3,2,0,1,3,3,2</td>
</tr>
<tr>
<td>4</td>
<td>2,0,1,0,3,0,1,0,2,0,1,0</td>
<td>3,0,2,0,2,3</td>
<td>n_4,1,2,1</td>
<td>3,3,2,0,1,3,3,2</td>
</tr>
<tr>
<td>5</td>
<td>0,1,0,2,0,1,0,3,0,0,2,0,1,0</td>
<td>3,1,2,0,2,3</td>
<td>n_5,1,2,1</td>
<td>3,3,2,0,1,3,3,2</td>
</tr>
<tr>
<td>6</td>
<td>1,0,2,0,1,0,3,0,0,2,0,1,0</td>
<td>3,0,2,0,2,3</td>
<td>n_6,1,2,1</td>
<td>3,3,2,0,1,3,3,2</td>
</tr>
<tr>
<td>7</td>
<td>0,1,0,2,0,1,0,3,0,0,2,0,1,0</td>
<td>3,0,2,0,2,3</td>
<td>n_7,1,2,1</td>
<td>3,3,2,0,1,3,3,2</td>
</tr>
<tr>
<td>8</td>
<td>0,1,0,2,0,1,0,3,0,0,2,0,1,0</td>
<td>3,0,2,0,2,3</td>
<td>n_8,1,2,1</td>
<td>3,3,2,0,1,3,3,2</td>
</tr>
<tr>
<td>9</td>
<td>0,1,0,2,0,1,0,3,0,0,2,0,1,0</td>
<td>3,0,2,0,2,3</td>
<td>n_9,1,2,1</td>
<td>3,3,2,0,1,3,3,2</td>
</tr>
</tbody>
</table>

where \(n_1, \ldots, n_8 \geq 3\) and \(n_9 \geq 4\).

Table 10: The possible pairs of passing 8 nodes

The rule is:

\[
\text{if } (p_0 = 1) \text{ then } T = [p_1] \\
\text{if } (p_0 = 3) \text{ then if } (p_2 > 1 \text{ and } p_1 = \max(d_1, d_2, d_3, d_4)) \text{ then } T = [p_1, p_3, p_2] \\
\text{else } T = [p_2, p_3, p_1]
\]
The above cases cover 2-9 transitions which are normally required to pass 1-8 nodes in a linear array embedded by using the Coordinate sequence approach. We have generated a rule for each case. These rules are combined as follows:

Algorithm \textit{BYPASS}:

\begin{verbatim}
if \(p_0 = 1\) then \(T = [p_1]\)
if \(p_0 = 2\) then if \((p_2 = 0 \text{ and } (p_1 = d_1 \text{ or } (p_1 \neq 1 \text{ and } p_1 = d_3)))\) or 
  \((p_2 = 1 \text{ and } p_1 = \max(d_1, d_2))\) or 
  \((p_2 = 2 \text{ and } p_1 = \max(d_1, d_2, d_3, d_4))\) then \(T = [p_1, p_2]\)
  else \(T = [p_2, p_1]\)
if \(p_0 = 3\) then if \((p_2 = 1 \text{ and } p_1 = \max(d_1, d_2))\) or 
  \((p_2 > 1 \text{ and } p_1 = \max(d_1, d_2, d_3, d_4))\) then \(T = [p_1, p_3, p_2]\)
  else \(T = [p_2, p_3, p_1]\)
if \((p_0 = 4 \text{ and } p_1 = \max(d_1, d_2))\) then \(T = [p_1, p_4, p_3, p_2]\)
  else \(T = [p_2, p_3, p_4, p_1]\)
if \(p_0 > 4\) then \(T = [p_2, p_3, \ldots, p_{p_0}, p_1]\)
\end{verbatim}

The last rule of Algorithm \textit{BYPASS} is added to the rules generated in the previous cases. It ensures the proper inter-node communication in a healthy \(d\)-cube, where \(d > 4\). Note that Algorithm \textit{BYPASS} provides a special \(e\)-cube-like routing approach with different channel sequence (coordinate sequence) used in communication. It guarantees a proper operation of bypassing any 1-8 consecutive failed nodes between the source and the destination.

The operation of Algorithm \textit{BYPASS} can be easily understood by going through the following examples. Consider a linear array, shown in Fig. 14, embedded in a hypercube equipped with Algorithm \textit{BYPASS} by using the Coordinate sequence approach.

\textbf{Example 1:} Referring to Fig. 14, let \(\text{SRC}=0110\), we have \([d_1, d_2, d_3, d_4] = [2, 1, 1, 3]\). Suppose that the faulty node is 0111 and the DST is 0101. We shall now show how Algorithm \textit{BYPASS} can determine the path from
the SRC to the DST. First, PROBE is obtained by the logic Exclusive-OR operation, SRC ⊕ DST=0110⊕0101=0011. Then, by definition, we have P=[p₀, p₁, p₂] = [2, 1, 0]. According to Algorithm BYPASS, since p₀ = 2, p₂ = 0, p₁ ≠ d₁, and p₂ = 1, we have T = [p₂, p₁] = [0, 1]. Thus the path is 0110-0111(dch)-0101, where 0111 is the faulty node between the source and the destination.

Example 2: The reverse path of Example 1. Referring to Fig. 14, let SRC=0101, we have [d₁, d₂, d₃, d₄] = [1, 3, 2, 1]. Suppose that the faulty node is 0111 and the DST is 0110. We shall now show how Algorithm BYPASS can determine the path from the SRC to the DST. First, PROBE is obtained by the logic Exclusive-OR operation, SRC ⊕ DST= 0101⊕0110=0011. Then, by definition, we have P=[p₀, p₁, p₂] = [2, 1, 0]. According to Algorithm BYPASS, since p₀ = 2, p₂ = 0, and p₁ = d₁ = 1, we have T = [p₁, p₂] = [1, 0]. Thus the path is 0101-0111(dch)-0110, where 0111 is the faulty node between the source and the destination.

Example 3: Referring to Fig. 14, let SRC=0100, we have [d₁, d₂, d₃, d₄] = [3, 1, 1, 2]. Suppose that the faulty nodes are 1100, 1101, 1111, 1110 and 1010, and the DST is 1011. We shall now show how Algorithm BYPASS can determine the path from the SRC to the DST. First, PROBE is obtained by the logic Exclusive-OR operation, SRC ⊕ DST=0100⊕1011=1111. Then, by definition, we have P=[p₀, p₁, p₂, p₃, p₄] = [4, 3, 2, 1, 0]. According to Algorithm BYPASS, since p₀ = 4 and p₁ = max(d₁, d₂) = 3, we have T = [p₁, p₄, p₃, p₂] = [3, 0, 1, 2]. Thus the path is 0100-1100(dch)-1101(dch)-1111(dch)-1011, where 1100, 1101 and 1111 are the faulty nodes between the source and the destination.
Now, the bypassing operation is briefly described as follows:

1. Initially each node has its own $d_1, d_2, d_3$ and $d_4$.

2. When a message-passing is requested, the source node computes $\text{PROBE} = \text{SRC} \oplus \text{DST}$, where $\oplus$ indicates the logic Exclusive-OR operation.

3. The DCH of the source node uses Algorithm $\text{BYPASS}$ to determine the correct coordinate transition sequence $T$.

4. DCH operations:
   
   (a) According to $T$, establish a path including the DCH's of some failed nodes between the source and the destination.
   
   (b) If the path is established, acknowledgment will be received by the source.
   
   (c) Message transmission.
   
   (d) Release the connection of DCH’s.

4.6.4 Algorithm $\text{MESHDCC}$

Algorithm $\text{MESHDCC}$ simply constructs the desired mesh without examining if it still has the feature of a 2-D Gray code ordering after the faulty columns and/or rows are removed.

Algorithm $\text{MESHDCC}$  
$\text{Embedding a 2-D mesh in a faulty hypercube}$

Given a $d$-cube with $r$ faulty nodes, and an $m_1 \times m_2$ mesh is required to be embedded in it. Let $d_1 = \lceil \log_2 m_1 \rceil$, $d_2 = \lceil \log_2 m_2 \rceil$, $F_{m_1}$ and $F_{m_2}$ are the lists of faulty nodes represented by the bound coordinates only for the $m_1$ side and the $m_2$ side respectively.
Step 1: If \((d - d_1) < d_2\) or \((2^d - r) < (m_1 \times m_2)\), then exit.

Step 2: Calculate the number of extra rows \(N_e = (2^{d_1} - m_1)\), that can be removed for constructing the \(m_1\) side of the mesh.

Step 3: Choose \(d_1\) coordinates among \(d\) coordinates to construct the \(m_1\) side of the mesh, then represent the faulty nodes in the \(d_1\) coordinates only.

Step 4: If \(r > N_e\) then choose \(N_e\) nodes from the fault list, else let \(N_e = r\) and choose all the \(r\) faulty nodes. Then let \(F_{m_1} = \{\text{the chosen nodes}\}\).

Step 5: Obtain \(F_{m_2}\) by deleting the \(N_e\) nodes from the \(r\) faulty nodes, then represent the faulty nodes of \(F_{m_2}\) in the \((d-d_1)\) coordinates only and calculate the number of healthy columns, \(N_c = 2^{d-d_1} - \text{(No. of elements in } F_{m_2})\).

If \(N_c < m_2\), then check if all the possible choices of \(N_e\) nodes are used exhaustively. If yes, then go to Step 9, else go to Step 4 for the next trial.

Step 6: Obtain \(S_{d_1}\) and generate \(LT_{m_1}\) of length \(2^{d_1}\). Then form \(L_{m_1}\) by deleting the \(N_e\) nodes listed in \(F_{m_1}\) from \(LT_{m_1}\).

Step 7: Obtain \(S_{d-d_1}\) and generate \(LT_{m_2}\) of length \(2^{d-d_1}\). Then form \(L_{m_2}\) by deleting the faulty nodes listed in \(F_{m_2}\) and extracting the \(m_2\) healthy nodes from the remaining healthy nodes of \(LT_{m_2}\).

Step 8: Represent the required mesh by the Cartesian product \(L_{m_1} \times L_{m_2}\).

Step 9: If all the combinations of choosing \(d_1\) coordinates among \(d\) coordinates are examined exhaustively, then exit, else go to Step 3 for the next trial.

The steps of Algorithm \textit{MESHDC}\textit{C} for embedding a mesh in a hypercube with \textit{Direct-Connect Capability} are the same as Algorithm \textit{MESH}, except that Lemma
2 is not applied and Algorithm LNRDCC is applied to construct each side of the mesh and it is not necessary to find a 2-D Gray code ordering.

4.6.5 An Embedding Example by Algorithm MESHDCC

An example which uses Algorithm MESHDCC to embed a 2-D mesh in an faulty hypercube is presented here. Consider a d-cube with r faulty nodes \((f_1=0001, f_2=0101, \text{and } f_3=1100)\) as shown in Fig. 15. Now an \(m_1 \times m_2\) mesh is desired to be embedded in it, where \(d=4, m_1=3, m_2=3, r=3\) and 4-tuple \(=(b_3 b_2 b_1 b_0)\). The solution steps are described as follows:

**Step 1:** Since \(d=4, m_1=3, \text{and } m_2=3\), we have \((d - \lfloor \log_2 m_1 \rfloor) = (d - \lfloor \log_2 m_2 \rfloor) = 2\), \(\lfloor \log_2 m_2 \rfloor = \lfloor \log_2 m_1 \rfloor = 2\), and \((2^d - r) = 2^d - 3 = 16 - 3 = 13 > (m_1 \times m_2) = 9\). Therefore, the mesh can be embedded in a healthy 4-cube and the number of healthy nodes is sufficient for the required mesh.

**Step 2:** The number of extra rows which can be deleted for constructing the \(m_1\) side of the mesh. \(N_e = (2^d - m_1) = 4 - 3 = 1\).

**Step 3:** Since \(m_1=3\), we have \(d_1 = \lfloor \log_2 m_1 \rfloor = 2\). Choose 2 coordinates among \(b_3 b_2 b_1 b_0\). There are \(\binom{4}{2}\) ways to do it. Let us choose \(b_3 b_2\) first and represent the faulty nodes by considering \(b_3 b_2\) only as below:

<table>
<thead>
<tr>
<th>Fault List:</th>
<th>(b_3)</th>
<th>(b_2)</th>
<th>(b_1)</th>
<th>(b_0)</th>
<th>(b_3)</th>
<th>(b_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(f_2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(f_3)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Step 4:** Since there are 3 faulty nodes and one extra node can be deleted, we have \(\binom{3}{1}\) ways to do it. Let us try to delete \(f_1\) first and generate the fault list for constructing the \(m_1\) side, denoted as \(F_{m_1} = \{f_1\} = \{00\}\).
Step 5: Generate the fault list for constructing the $m_2$ side by deleting the fault nodes in $F_{m_1}$, then represent the fault list in $b_1b_0$, denoted as $F_{m_2} = \{01, 00\}$. The operation is illustrated as follows:

Fault List:

<table>
<thead>
<tr>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$ = 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>← Deleted</td>
<td></td>
</tr>
<tr>
<td>$f_2$ = 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>→</td>
<td>0</td>
</tr>
<tr>
<td>$f_3$ = 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>→</td>
<td>0</td>
</tr>
</tbody>
</table>

The number of healthy columns = 2 < $m_2$. We should go back to step 4 and try to delete $f_2$ instead of $f_1$.

Step 4(trial 2): Delete $f_2$ and generate the fault list $F_{m_1} = \{f_2\} = \{01\}$.

Step 5(trial 2): Generate the fault list for constructing the $m_2$ side. The fault list in $b_1b_0$ is $F_{m_2} = \{01, 00\}$. The operation is illustrated as follows:

Fault List:

<table>
<thead>
<tr>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$ = 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>→</td>
<td>0</td>
</tr>
<tr>
<td>$f_2$ = 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>← Deleted</td>
<td></td>
</tr>
<tr>
<td>$f_3$ = 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>→</td>
<td>0</td>
</tr>
</tbody>
</table>

The number of healthy columns = 2 < $m_2$. We should go back to step 4 and try to delete $f_3$ instead of $f_1$ or $f_2$.

Step 4(trial 3): Delete $f_3$ and generate the fault list, $F_{m_1} = \{f_3\} = \{11\}$.

Step 5(trial 3): Generate the fault list $F_{m_2}$. The operation is illustrated as follows:
Fault List:

<table>
<thead>
<tr>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$ = 0 0 0 1</td>
<td>$\rightarrow$</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_2$ = 0 1 0 1</td>
<td>$\rightarrow$</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_3$ = 1 1 0 0</td>
<td>Deleted</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The number of healthy columns = 3 = $m_2$ and $F_{m_2} = \{01\}$.

**Step 6:** Obtain $S_2$ by table-look-up, where $S_2 = \{0, 1, 0\}$, and produce a linear array $LT_{m_1} = \{00, 01, 11, 10\}$ according to $S_2$. Then generate $L_{m_1}$ by removing $F_{m_1} = \{11\}$. Finally, we have $L_{m_1} = \{00, 01, 10\}$.

**Step 7:** Obtain $S_2$ by table-look-up, where $S_2 = \{0, 1, 0\}$, and produce a linear array $LT_{m_2} = \{00, 01, 11, 10\}$ according to $S_2$. Then generate $L_{m_2}$ by removing $F_{m_2} = \{01\}$. Finally, we have $L_{m_1} = \{00, 11, 10\}$.

**Step 8:** Represent the required mesh by the Cartesian product $L_{m_1} \times L_{m_2}$.

### 4.6.6 The Time Complexity of Algorithm MESHDC

The time complexity of Algorithm MESHDC is analyzed and expressed as follows: In the worst case $\binom{d}{d_1}$ combinations of coordinates will be tried to construct the $m_1$ side of the mesh and for each choice $\binom{r}{N_e}$ ways of deleting rows will be evaluated. Step 5 will require $rN_e$ operations. Therefore the worst case time complexity is $O[P\binom{r}{N_e}rN_e]$, where $P = 2^d$.

### 4.7 The Parallel Version of Algorithm MESHDC

Algorithm MESHDC can be parallelized by distributing the partitioned combinations of choosing $d_1$ coordinates among the $d$ coordinates for constructing the $m_1$ side of the mesh to each healthy node and making use of Algorithm LNRDCC.
Figure 15: An example of embedding a 2-D mesh in a faulty hypercube using Algorithm MESHDCC
in each healthy node to find both sides of the desired 2-D mesh. Once the desired linear arrays are found by any healthy node, they are sent to the host computer and the processes executing in all the healthy nodes are killed. If all the healthy nodes cannot find the desired linear arrays, the embedding operation is also terminated. The parallel approach of embedding 2-D mesh in a faulty hypercube is listed as follows:

Algorithm \textit{PMESHDCC}:

\textbf{Steps 1-3:} Steps 1-3 of Algorithm \textit{MESHDCC} are executed on the host computer.

\textbf{Step 4:} Each healthy node performs the following operations:

\textbf{Step 4.1:} Choose \(d_1\) coordinates among \(d\) coordinates from the partitioned combinations to construct the \(m_1\) side of the mesh, then represent the faulty nodes in the \(d_1\) coordinates only.

\textbf{Step 4.2:} If \(r > N_e\) then choose \(N_e\) nodes from the fault list, else let \(N_e = r\) and choose all the \(r\) faulty nodes. Then let \(F_{m_1} = \{\text{the chosen nodes}\}\).

\textbf{Step 4.3:} Obtain \(F_{m_2}\) by deleting the \(N_e\) nodes from the \(r\) faulty nodes, then represent the faulty nodes of \(F_{m_2}\) in the \((d-d_1)\) coordinates only and calculate the number of healthy columns, \(N_c = 2^{d-d_1} - \text{(No. of elements in } F_{m_2})\). If \(N_c < m_2\), then check if all the possible choices of \(N_e\) nodes are used exhaustively. If yes, then go to Step 4.7, else go to Step 4.2 for the next trial.

\textbf{Step 4.4:} Obtain \(S_{d_1}\) and generate \(LT_{m_1}\) of length \(2^{d_1}\). Then form \(L_{m_1}\) by deleting the \(N_e\) nodes listed in \(F_{m_1}\) from \(LT_{m_1}\).
Step 4.5: Obtain $S_{d-d_1}$ and generate $LT_{m_2}$ of length $2^{d-d_1}$. Then form $L_{m_2}$ by deleting the faulty nodes listed in $F_{m_2}$ and extracting the $m_2$ healthy nodes from the remaining healthy nodes of $LT_{m_2}$.

Step 4.6: Send $L_{m_1}$ and $L_{m_2}$ to the host computer and exit.

Step 4.7: If all the partitioned combinations of choosing $d_1$ coordinates among $d$ coordinates are examined exhaustively, then inform the host computer that the desired linear arrays cannot be found, else go to Step 4.1 for the next trial.

Step 5: After receiving the desired linear arrays, the host computer represents the required mesh by the 2-D Gray code ordering formed by the Cartesian product $L_{m_1} \times L_{m_2}$ and kills the processes executing in each healthy node. Additionally, if all the healthy nodes cannot find the desired linear arrays, the host computer also kills the processes executing in each healthy node.

In Algorithm PMeshDCC, all the combinations of choosing $d_1$ coordinates among the $d$ coordinates to construct the $m_1$ side of the mesh are partitioned and distributed to each healthy node, then each healthy node performs the partitioned task to find both sides of the desired 2-D mesh. Consequently, the time complexity of Algorithm PMeshDCC is better than that of Algorithm MeshDCC. An upper bound for the time complexity of Algorithm PMeshDCC can be expressed as:

$$O\left(\frac{P}{P'} \left( \frac{r}{N_e} \right)^{rN_e} \right),$$

where $P'$ is the number of healthy nodes.

4.8 Summary

We have presented different approaches of embedding 2-D meshes in a faulty hypercube. Algorithm MeshCube combines existing algorithms to embed 2-
D meshes in a faulty hypercube. It may not find the maximum 2-D meshes. Algorithm \textit{MESHLATIN} has been developed to overcome the limitation of Algorithm \textit{MESHCUBE}. It finds a 2-D mesh in a faulty hypercube by deleting faulty rows/columns and pasting the healthy rows and columns in a 2-D Gray code sequence. The parallel versions of Algorithm \textit{MESHLATIN} have also been developed. Algorithm \textit{MESHLATIN} shows a very pessimistic complexity in the worst-case. An experimental result of 2-D mesh embedding simulation reveals that the average performance is acceptable for $d \leq 9$. It is clear that the execution time depends on the dimension of hypercube, the mesh size, and the distribution of the faulty nodes. In general, the larger the dimension of the desired 2-D mesh is, the longer the execution time of embedding is. In order to improve the complexity, we have developed a faster algorithm, namely Algorithm \textit{MESHDCC}, to embed a linear array in a faulty hypercube that has the \textit{Direct-Connect Capability}. It is assumed that the links and the DCH's of the hypercube are fault-free, and the faulty hypercube has faulty processors only. In addition, Algorithm \textit{BYPASS} has been developed to incorporate in the DCH's of each node. It is an e-cube-like algorithm for enabling the local communication between columns and/or rows in the embedded mesh exactly via the DCH’s of some removed faulty columns/rows.
CHAPTER V

Algorithms for Data Communication in Faulty Hypercubes

5.1 Introduction

Algorithms for data communication in healthy/faulty hypercubes have been developed by several researchers [19,21,22,29,39,44]. However, these data communication algorithms developed are not the types widely used in most of parallel applications. In many parallel applications, global data communication operations such as the Global Sum operation and the Global Broadcast operation are required. Therefore, different schemes of performing global data communication in a faulty hypercube are developed and described in this chapter.

5.2 Global Sum in Faulty Hypercubes

The Global Sum (GS) operation is performed in many algorithms to add or collect data residing in each node to one node. The $d$ communication steps are shown in Fig. 16 for a communication dimension sequence of $\{0,1,2,3\}$. The GS can also be collected by performing a partial Global Sum operation in $2^i$ disjoint $(d-i)$-subcubes in parallel and collecting the partial global sum in nodes that form an $i$-subcube. For example in a 5-cube, for $i = 2$ each of the four disjoint 3-cubes (00xxx, 01xxx, 11xxx and 10xxx) can perform a partial Global Sum operation in 3 steps and collect the partial global sum in nodes 00000, 01000, 11000 and 10000 respectively, which are nodes of the 2-cube xx000. Then these $2^i = 4$
nodes of the 2-cube \(xx000\) perform a Global Sum (GS) operation in 2 steps. Note that the dimension sequences in the four 3-cubes can be different. The algorithm presented next for finding the GS in a faulty hypercube is based on the latter scheme described. Fig. 18 shows a 5-cube with 5 faulty nodes. It is assumed that the data to be summed is resided in the 27 healthy nodes. The algorithm first tries to split the 5-cube into two 4-cubes. One contains all healthy nodes; the other contains all faulty nodes. If such a split can be found, then the partial sum in each healthy node in the faulty cube can be sent to the adjacent nodes in the good cube in one step and the GS can be performed in the good cube in \((d-1)\) communication steps. In order to find a \((d-1)\)-cube containing all of the faulty nodes, a commutative and associative intersection operation \(I_d = \{I\}^d\) is defined as follows: 010=0, 111=1, 011=q, q10=q11=q=q. The coordinates that have a "0" or "1" value in the intersection of all faulty processors will be referred to as the fixed coordinates \(c_f\), and the number of the fixed coordinates is denoted as \(n_f\).

We shall define a subroutine \textit{Intersection} to perform the intersection \(I_{1,2,3,\ldots,r} = f_1I_f2I_f3I\ldots I_f_r\) for a given set of faulty nodes \((f_1, f_2, \ldots, f_r)\). The subroutine will return \(c_f\) and \(n_f\). If \(n_f \geq 1\), then the faulty nodes belong to a subcube of dimension \((d - k)\) with \(k = n_f\). In Fig. 17, \(I_{1,2,3,4,5} = qqqqq\) and therefore a good-faulty subcube split cannot be found. Then the 5-cube is split along any dimension into 2 faulty 4-cubes. Fig. 17 shows the first split along dimension 4 and leads to two faulty 4-cubes \((0xxx\) and \(1xxx\)). Subcube \(0xx\) contains \(f_1, f_2,\) and \(f_3\); subcube \(1xxx\) contains \(f_4\) and \(f_5\). We also have \(I_{1,2,3} = 0qqqq\) and \(I_{4,5} = 1qqq0\). Next, \(1xxx\) can be split into a good-faulty pair of 3-cubes, namely \(1xxx1\) and \(1xxx0\). GS in subcube \(1xxx\) can then be performed as follows: First the partial sum in each healthy node of subcube \(1xxx0\) is sent to neighbors in subcube \(1xxx1\) and then a GS is performed in subcube \(1xxx1\). \(0xxx\) has to be
split again into a faulty-faulty subcube pair. One way to split is $0x1xx(f_3)$ and $0x0xx(f_1, f_2)$ as shown in Fig. 17. Since $I_3 = f_3 = 01101$ and $I_{1,2} = 0q01q$, both subcubes can be split into a good-faulty subcube pair. The partial GS's for the subcubes $0x1xx$ and $0x0xx$ are accumulated in one node in each of the collecting subcubes $001xx$ and $0x00x$ respectively. The computation of the partial GS's in subcubes $0x1xx$ and $0x0xx$ are done in parallel in 3 steps. In order to collect the 2 partial GS's in one node in the 4th step, the nodes collecting the partial GS's in each 3-cube must be adjacent. The partial GS in $1xxx$ will be performed in parallel to the above partial sum operations. The collecting nodes for each of the partial GS's must be in the same 2-cube so that the final GS can be collected in 2 steps. Such collecting nodes can be determined by performing a consistency operation on the healthy (collecting) subcubes found in the last split. For the 3 collecting cubes $1xxx1, 001xx$ and $0x00x$ in Fig. 17, the collecting nodes can be found by performing a consistency operation $\&_d = \{\&\}^d$ defined as follows: $0\&0 = 0\&x = 0, 1\&1 = 1\&x = 1, 0\&1 = -\&0 = -\&1 = -\&x = -, \ x\&x = x$. The candidate collecting nodes for each collecting cube are obtained by replacing the "-" coordinates of the result with the corresponding coordinate values in the collecting cube. For this example: $1xxx1\&001xx\&0x00x = 000101$. The candidate collecting nodes are $10x01, 00101$, and $00001$, which are all nodes of subcube $x0x01$.

The following variables are used in describing the Global Sum operation in the faulty hypercube.

$SENDSWITCH$: Control each node to send once only.

$DIMENSIONSEQ$: Sequence of coordinates to perform the message-passing operation.
Step 1 (dimension 0)

Step 2 (dimension 1)

Step 3 (dimension 2)

Step 4 (dimension 3)

Figure 16: Global Sum operation in a healthy 4-cube
**MYNODE**: Binary representation of the current local node which executes the algorithm.

**NOTRECNODE**: Nodes which should not do a specific receiving operation.

**DIRECTION**: Direction to send along a specific link.

**CRECV**: Synchronized receiving operation.

**CSEND**: Synchronized sending operation.

**s**: Buffer for sending partial sum in each node, or collecting global sum at the collecting node.

**r**: Buffer for receiving operation.

The steps of the splitting operation are listed as follows:

**Algorithm em SPLIT:**

**Step 1**: Given a set of faulty nodes \( F = \{f_1, f_2, \ldots, f_r\} \), apply Intersection to obtain \( c_f \) and \( n_f \). If \( n_f \geq 1 \), then select a coordinate among \( c_f \) to split and determine the following parameters and exit. \( DIMENSIONSEQ_i = \{c_{11}, \text{the rest of the coordinates in any order}\} \), where \( c_{11} \) is the first dimension to send, which can be any one of \( c_f \). **NOTRECNODE** = the nodes obtained by complementing coordinate \( c_{11} \) of faulty nodes. Specify the transfer direction (1 to 0 or 0 to 1) for each dimension.

**Step 2**: Let \( F_{G_i} \) be the set of faulty nodes belonging to a cube \( G_i \) to be split into two faulty subcubes and \( L_{f-f} \) be the list where each entry \( i \) consists of the cube to be split \( (G_i) \), the faulty nodes \( (F_{G_i}) \) in cube \( G_i \), and the candidate coordinates \( (C_{d_i}) \) for splitting \( G_i \) into two faulty subcubes. Initially, put \( F_{G_i} = F \), \( G_i = \text{the d-cube} \) and \( C_{d_i} = \{(d-1), \ldots, 0\} \) in \( L_{f-f} \).
Step 3: Select a coordinate among $C_{d_i}$ to split $G_i$ into two subcubes $G_{i_1}$ (containing faulty nodes $F_{G_{i_1}}$) and $G_{i_2}$ (containing faulty nodes $F_{G_{i_2}}$). Then remove the split coordinate $C_{split}$ from $C_{d_i}$ and put $(G_{i_1}, F_{G_{i_1}}, C_{d_i})$ and $(G_{i_2}, F_{G_{i_2}}, C_{d_i})$ in the list $L_{f-f}$.

Step 4: Select a $(G_{i}, F_{G_{i}}, C_{d_i})$ from $L_{f-f}$. Then apply Intersection and consider $C_{d_i}$ coordinates only. If $n_f \geq 1$ then select a coordinate among $c_f$ to split and obtain a faulty-good pair of subcubes $G_{i_1}$ and $G_{i_2}$. Then put them in a list $L_{f-g}$ containing the faulty-good pairs. If $n_f < 1$, go to step 3.

Step 5: If $L_{f-f}$ is not empty, go to Step 4.

Step 6: Determine the collecting node by performing a consistency operation on the healthy subcubes in $L_{f-g}$. If the collecting node cannot be found, all possible ways of splitting should be tried. The trial starts from the last split where a choice was made. In the worst case, all different splitting coordinates will be tried. If all trials have been performed, then exit.

Step 7: Determine the following parameters for each group of subcubes.

1. Let $C_l$ be the split coordinates used in the path of the tree from the last split to the root. $DIMENSIONSEQ_i = \{ C_{lastsplit}, \text{(the rest of the coordinates-}C_l) \text{ in any order, } C_l \}$.

2. $NOTRECNODE = \{ \text{nodes obtained by inverting all faulty nodes on the bit position indicated by the first element in } DIMENSIONSEQ_i \}$.

3. $DIRECTION_i = \{ \text{the corresponding bit value in each dimension of the collecting node} \}$. 
For the example in Fig. 17, Step 7 determines the following parameters for each group of nodes in each subcube. For subcube lxxxx, we have $\text{DIMENSIONSEQ}_i = \{0, 1, 2, 3, 4\}$, and $\text{NOTRECNODE} = \{10101, 11011\}$ obtained by inverting faulty nodes 10100 and 11010 on bit 0. For subcube 0x1xx, we have $\text{DIMENSIONSEQ}_i = \{3, 0, 1, 2, 4\}$ and $\text{NOTRECNODE} = \{00101\}$. For subcube 0x0xx, we have $\text{DIMENSIONSEQ}_i = \{1, 0, 3, 2, 4\}$, $\text{NOTRECNODE} = \{00000, 01001\}$. $\text{DIRECTION}_i$ for subcube lxxxx is \{10001\} and $\text{DIRECTION}_i$ for subcube 0xxxx is \{00001\}. The Global Sum operation which uses the parameters $\text{DIMENSIONSEQ}_i$, $\text{NOTRECNODE}$ and $\text{DIRECTION}_i$ is shown in Fig. 18.

In an application program, the splitting operation is executed initially to obtain the required parameters, then the parameters are distributed to each healthy node. Whenever a Global Sum operation is required in the application program, Algorithm $\text{GSFnode}$ will perform the Global Sum in the faulty hypercube in $d$ steps.

Referring to Algorithm $\text{GSFnode}$ listed as below, initially each node holds its own partial sum in buffer $s$. It is assumed that both send and receive operations in Algorithm $\text{GSFnode}$ are synchronous. Each healthy node executing Algorithm $\text{GSFnode}$ first sets the flag $\text{SENDSWITCH}$ which controls the node to send the message once only for performing the Global Sum operation correctly. After a sending operation is made, $\text{SENDSWITCH}$ is reset to prevent a second sending operation which will cause an incorrect result. Starting from the first split coordinate, the healthy nodes in the faulty subcube send their partial sum in $s$ to their corresponding nodes in the healthy subcube. The sender and receiver are distinguished by $\text{NOTRECNODE}$ and $\text{DIMENSIONSEQ}_i$. If the current node is belong to $\text{NOTRECNODE}$, then it will not receive the message. The
reason is that, in a commercial hypercube system, the message-passing primitive for sending a message requires the receiving node be specified explicitly. Each receiving node adds its own partial sum to the partial sum received for obtaining a new partial sum. The operation is repeated for the rest coordinates indicated by $DIMENSIONSEQ_i$. Finally, the global sum is accumulated in the collecting node.

### Algorithm GSFnode:

/* Initially each node holds its own partial sum in $s$ */
/* Both send and receive operations are synchronous */

`SENDSWITCH = 1; /* each node to send once only */`

`i = 0;`

`j = DIMENSIONSEQ_i; /* the first split coordinate */`

if (`SENDSWITCH = 1`) then

if (`MYNODE $\neq$ NOTRECNODE`) and (coordinate $j$ of `MYNODE` $= DIRECTION_i$) then { CRECV partial sum $r$ from link $j$;

`$s = s + r;$`

}

else { Send $s$ to neighbor via link $j$;

`SENDSWITCH = 0;`

}

for $i = 1$ to $d-1$

{ `j = DIMENSIONSEQ_i;`

if (`SENDSWITCH = 1`) then

if (coordinate $j$ of `MYNODE` $= DIRECTION_i$) then{ Receive $r$ from link $j$;

`$s = s + r;$`

}
else { Send $s$ to neighbor via link $j$;
    SENDSWITCH = 0;
}

Figure 17: An example of splitting operation

The *Global Sum* operation which uses the parameters including
*DIMENSIONSEQ*$_i$, *NOTRECNODE*, and *DIRECTION*$_i$ is shown in
Fig. 18. In each step of message-passing operation in Algorithm *GSFnode*, the
communication cost can be expressed as follows: \( T_{\text{commu}} = S_C + wT_C \), where \( T_{\text{commu}} \) is the communication cost, \( S_C \) is the set-up cost for the communication, \( T_C \) is the transmission cost per floating point addition operation and \( w \) is the number of words to be transferred. Since Algorithm \( GSF\text{node} \) performs the \textit{Global Sum} operation in \( d \) Steps. Therefore, the communication cost of Algorithm \( GSF\text{node} \) can be modeled as follows: \( T_{GSF\text{node}} = d(S_C + wT_C + wT_{\text{add}}) + T_{\text{overhead}} \approx d(S_C + wT_C + wT_{\text{add}}) \), since \( d(S_C + wT_C + wT_{\text{add}}) \gg T_{\text{overhead}} \), where \( d \) is the dimension of hypercube used, \( T_{\text{add}} \) is the cost of a floating point addition operation and \( T_{\text{overhead}} \) is the overhead cost. The difference of the communication cost between Algorithm \( GSF\text{node} \) and the \textit{Global Sum} algorithm in a healthy hypercube [9] is the overhead cost. The overhead cost is caused by the execution times required by Algorithm \( SPLIT \) and the extra conditional checks used in Algorithm \( GSF\text{node} \).

The time complexity of splitting operation can be analyzed as below. Given a set of faulty nodes \( F = \{ f_1, f_2, \ldots, f_r \} \), where \( r \leq d \). In the worst case, an upper bound for the time complexity of performing intersection is \( O(d) \) since there are at most \( d \) faulty nodes in a subcube of a splitting tree. The number of subcubes at level \( i \) is at most \( 2^i \) and there are \( d \) levels at most. Therefore, in the worst case, the time complexity of intersection operations for each splitting trial can be expressed as: \( O \left( d + 2^1 d + 2^2 d + 2^3 d + \cdots + 2^{d-1} d \right) = O \left( d(2^{d+1} - 1) \right) \approx O(d2^d) \). The number of splitting trials can be enumerated in the following way. Initially there are \( d \) different ways to split the faulty nodes and two \((d - 1)\)-subcubes are obtained. The next split has \((d - 1)\) choices for each \((d - 1)\) faulty subcube. All possible selections of this level is \((d - 1)^2\). After each of these splits, we have four \((d - 2)\) faulty subcubes. There are \((d - 2)\) choices of the next
Figure 18: An example showing Global Sum operation in a faulty hypercube
split for each \((d - 2)\) faulty subcube and so on. Finally, a d-level tree is obtained. The number of all possible choices of level \(i\) is \((d - i)^2i\). Therefore, an upper bound for the time complexity of all splitting trials can be expressed as:
\[
O\left(d(d - 1)^2(d - 2)^4(d - 3)^8 \cdots (d - (d - 2))^{2(d - 2)}\right) = O\left(\prod_{k=0}^{d-2}(d-k)^{2^k}\right).
\]
At end of each splitting trial, the Consistency operation is applied. The time complexity of the Consistency operation is \(O(d)\) since there are \(d\) pairs of healthy-faulty subcubes. Therefore, an upper bound for the time complexity of splitting operation can be expressed as follows: \(O\left(\prod_{k=0}^{d-2}(d-k)^{2^k}\right)(d + d^2d)\) or \(O\left(\prod_{k=0}^{d-2}(d-k)^{2^k}\right)d^2d\).

The worst case time complexity of Algorithm \textit{SPLIT} is exponential since the splitting trials are performed on a d-level tree. However, the average complexity of the algorithm is expected to be much lower especially when the number of faulty nodes is small. This is because, when a good-faulty split is found, no more trials are made in that subtree, and in most cases a solution is found early in the search.

5.3 Global Broadcast in Faulty Hypercubes

The \textit{Global Broadcast} operation in a faulty hypercube can be performed by reversing the dimension sequence found for the \textit{Global Sum} operation stated in Section 5.2. It is performed by Algorithm \textit{GBFnode} and the splitting operation. The splitting operation has been stated in Section 5.2 for preparing the parameters used in Algorithm \textit{GBFnode} which is executed in each healthy node in the faulty hypercube. In an application program, Algorithm \textit{GBFnode} will perform the \textit{Global Broadcast} operation in the faulty hypercube in \(d\) steps.
**Algorithm GBFnode:**

if (MYNODE ∈ SENDER) then SENDSWITCH = 1;

for i = d-1 to 0

{   
   j = DIMENSIONSEQi;
   
   if (SENDSWITCH = 1)
   
       then CSEND s via link j to neighbor;
   
   else
   
       if ((MYNODE - 2^j) ∈ SENDER )
       
           then CRECV s from link j;
       
   else
   
       { 
       
           if (MYNODE ∉ NOTRECNODE)
       
               then
       
                   { 
                   
                       SENDSWITCH =1;
                   
                       include MYNODE in SENDER;
                   
                   } 
       
       }

}

An example of the *Global Broadcast* operation in a faulty hypercube with faulty node $f_1 = 0011$, $f_2 = 1000$ and $f_3 = 1111$ is shown Fig. 19. The operation is performed by reversing the dimension sequence of the example used in Fig. 19.

The communication cost of Algorithm GBFnode can be modeled as follows:

$T_{GBFnode} = d(S_C + wT_C) + T_{overhead} \approx d(S_C + wT_C), \text{ since } d(S_C + wT_C) >> $
Figure 19: An example showing Global Broadcast operation in a faulty hypercube
Algorithm \textsc{GBFnode} has an overhead cost which is larger than the \textit{Global Broadcast} operation in a healthy hypercube.

### 5.4 Global Sum and Global Broadcast with Exchange-Add

The \textit{Exchange-Add} algorithm [20] can also be used to calculate the global sum. If bidirectional communication channels are available, the global sum is accumulated in every node after \( d \) bidirectional communication steps.

In a faulty \( d \)-cube, if a \((d-1)\)-cube containing all the faulty nodes can be found, then the global sum can be accumulated as follows:

1. All healthy nodes of the faulty \((d-1)\)-cube send their data to the neighbors of the healthy \((d-1)\)-cube in one communication step.

2. An exchange-add operation is performed in the healthy \((d-1)\)-cube in \((d-1)\) bidirectional communication steps.

3. The global sum is sent by the nodes of the healthy \((d-1)\)-cube to their healthy neighbors of the faulty \((d-1)\)-cube in one communication step.

Thus the global sum is accumulated in each node in \((d+1)\) steps. After the split coordinate is determined, the collecting node can be any of the nodes in the healthy \((d-1)\)-cube. The \textit{GS-GB with partial Exchange-Add} operation in the faulty hypercube can be performed in \( d+1 \) steps. For example, \( f_1 = 1111 \) and \( f_2 = 1010 \), we have \( f_1 \oplus f_2 = 0101 \) and subcube \( 1xxr \) or subcube \( xx1x \) contains all the faulty nodes. The collecting node can be any one of nodes in the healthy \((d-1)\) sub-cube. Similarly, the parameters such as \textit{DIMENSIONSEQ}_i, \textit{DIRECTION}_i, \textit{NOTRECNODE}, and \textit{SENDER}, are determined by the method stated previously. The first operation of data communication is performing the \textit{Global Sum}
operation which sends all the partial sums in the healthy nodes of the faulty \((d-1)\)-subcube to the corresponding nodes in the healthy \((d-1)\)-subcube in one step. Then, the \textit{Exchange-Add} Algorithm is applied in the healthy \((d-1)\)-subcube in d-1 steps [9]. As a result of that, each node in the healthy \((d-1)\)-subcube holds the result of the global sum. Next, the result of the global sum holding in the healthy \((d-1)\)-subcube will be sent to the faulty \((d-1)\)-subcube in one step.

In each healthy node, Algorithm \textit{GSGBEAFnode} performs \textit{GS-GB with the partial Exchange-Add} operation in a faulty hypercube. The input parameters to each node for executing the algorithm are \textit{SENDER, DIMENSIONSEQ}_i, \textit{NOTRECNODE}, and \textit{DIRECTION}_i. If \(EA=1\), then the first step is to collect all the partial sums in each healthy node of the \((d-1)\) faulty subcube to the corresponding node of the \((d-1)\) healthy subcube. Next, the \textit{Exchange-Add} Algorithm [9] is applied to do the GS-GB operation in \((d-1)\) steps. After \textit{Exchange-Add} operation, each healthy node in the healthy \((d-1)\) subcube has the global sum. Then, the global sum is distributed from the corresponding nodes of the healthy \((d-1)\) subcube to the healthy nodes of the faulty \((d-1)\) subcube. Finally, each healthy node in the faulty hypercube has the global sum.

\textbf{ALGORITHM GSGBEAFnode:}

\begin{verbatim}
/* Initially each node holds its own partial sum in buffer s */
if (EA=1) then
  \{ i = 0;
  j = DIMENSIONSEQ_i;
  if (MYNODE \neq NOTRECNODE) and
    (coordinate j of MYNODE = DIRECTION_i)
    then {

\end{verbatim}
CRECV partial sum \( r \) from link \( j \);
\[
s = s + r;
\]
include MYNODE in SENDER;
}

else CSEND partial sum \( s \) to neighbor via link \( j \);

perform Exchange-Add operation in the healthy \((d-1)\)-cube;
\[
i = 0;
\]
\[
j = DIMENSIONSEQ_i;
\]
if (MYNODE \( \in \) SENDER)
\[
\text{then CSEND} \ s \ \text{via link} \ j \ \text{to neighbor};
\]
else if ((MYNODE \( - 2^j \) \( \in \) SENDER)
\[
\text{then CRECV} \ s \ \text{from link} \ j;
\]
}

An example showing Global Sum and Global Broadcast with partial Exchange-Add operation in a faulty hypercube is shown in Fig. 20. Assume that communications are initiated simultaneously. The communication cost of Algorithm \( GSGBEAFnode \) can be modeled as follows:

\[
T_{GSGBEAFnode} = (d - 1)(S_C + wT_C + wT_{add}) + (S_C + wT_C + wT_{add}) + (S_C + wT_C) + T_{overhead}.
\]

Comparing to the Exchange-Add operation in a healthy hypercube, only an additional step of bi-directional communication and a different overhead cost are observed.
Figure 20: An example showing Global Sum and Global Broadcast with partial Exchange-Add operation in a faulty hypercube
5.5 Summary

Algorithms for global data communication in a faulty hypercube have been developed. They are: *Global Sum (GS)* operation, *Global Broadcast (GB)* operation, and *GS-GB with partial Exchange-Add* operation. The performance model of each operation has also been presented. In addition, the splitting operation has been developed to determine the necessary parameters for controlling the sequences of global data communication.
CHAPTER VI
APPLICATIONS OF EMBEDDING AND COMMUNICATION SCHEMES

6.1 Introduction

The embedding and data communication schemes described in the previous chapters are applied to implement the parallel Conjugate Gradient Algorithm and the parallel sorting algorithm in [45]. First, the parallel Conjugate Gradient Algorithm in a healthy hypercube and its cost of data communication are introduced. Next, the parallel Conjugate Gradient Algorithm in a faulty hypercube is proposed by applying the linear array embedding algorithms to obtain a linear array and the data communication algorithms to perform the global communication. The time performance model and the performance degradation of the parallel CG algorithm in a faulty hypercube are presented. The embedding and data communication schemes developed for a faulty hypercube can also be applied to implement the parallel sorting algorithm described in [45]. First, the parallel sorting algorithm developed for a mesh-connected parallel computer and its time performance are introduced. Next, the parallel sorting algorithm can be applied to a faulty hypercube by embedding a 2-D mesh in the faulty hypercube and performing local communication as required. The performance degradation of the parallel sorting algorithm is also analyzed. Note that any parallel algorithm which requires 1-dimensional communication (or 2-dimensional communication), Global Sum op-
operation, and *Global Broadcast* operation can be implemented in a faulty hypercube by applying the embedding and data communication schemes developed.

6.2 Application to the Parallel Conjugate Gradient Algorithm

6.2.1 The Parallel CG Algorithm in a Healthy Hypercube

Parallel iterative algorithms based on the *Conjugate Gradient* method have been developed for hypercubes [9,20]. The CG algorithm is widely used to solve the linear equations in the form of $Ax = b$ obtained by finite element discretization, where $A$ is a large, sparse and banded matrix with proper ordering of the variables $x$. The parallel CG Algorithm on healthy hypercubes requires different types of data communications and topologies [9]. The types of data communication per iteration of the parallel CG algorithm consist of the *Global Sum*-*Global Broadcast*, (or *Exchange-Add* operation), and local communication between the neighboring nodes in a linear array, or 2-D mesh. Thus, the topologies required are the hypercube and the linear array. The hypercube is for the *Global Sum* operation, the *Global Broadcast* operation, and the distributed *Exchange-Add* operation. The linear array is for the local communication. The steps for the coarse grain parallel CG algorithm are described as follows [9]:

Choose $x_0$, let $r_0 = p_0 = b - Ax_0$ and compute $< r_0, r_0 >$.

Then, for $k = 0, 1, 2, \ldots$

1. form $q_k = Ap_k$ (local communication).

2. a. form $< p_k, q_k >$ and $< q_k, q_k >$
   
   (in one *GS-GB* communication step or *Exchange-Add* operation).

3. a. $\alpha_k = \frac{< r_k, r_k >}{< p_k, q_k >}$.
   
   b. $\beta_k = \alpha_k \frac{< q_k, q_k >}{< p_k, q_k >} - 1$.
   
   c. $< r_{k+1}, r_{k+1} > = \beta_k < r_k, r_k >$. 


4. \( r_{k+1} = r_k - \alpha_k q_k \).
\[ x_{k+1} = x_k + \alpha_k p_k \]
\[ p_{k+1} = r_{k+1} + \beta_k p_k \]

The communication time in the parallel CG algorithm can be expressed as follows [9]:

\[ T_{LC} = 2(S_C + m T_C) \]
for local communication.
\[ T_{EA} = d(S_C + 2T_C + 2T_{add}) \]
for Exchange-Add operation.

where \( T_{LC} \) is the cost of local communication, \( S_C \) is the set-up cost, \( m \) is the number of inter-node variables in \( p_k \), \( T_C \) is the transmission cost per floating point word, \( T_{EA} \) is the cost of Exchange-Add operation, \( d \) is the dimensional of hypercube, and \( T_{add} \) is the cost of addition per floating-point operation.

The performance model of the coarse grain parallel CG algorithm per iteration can be expressed as follows:

1. for \( q_k = Ap_k \) (local communication)
\[ T_1 = Parallel\ computation\ (Ap_k) + Local\ communication \]
\[ = z \frac{N}{p} T_{muladd} + 2(S_C + m T_C). \]

2. a. for \( \langle p_k, q_k \rangle \) and \( \langle q_k, q_k \rangle \).
\[ (Exchange - Add\ operation). \]
\[ T_2 = Parallel\ inner\ product + Exchange\ Add\ Operation \]
\[ = \frac{N}{p} T_{muladd} + \frac{N}{p} T_{muladd} + d(S_C + 2T_C + 2T_{add}). \]
3. a. \( \alpha_k = \frac{<r_k, r_k>}{<p_k, q_k>}, \) b. \( \beta_k = \alpha_k \frac{<q_k, q_k>}{<p_k, q_k>} - 1. \)

c. \( <r_{k+1}, r_{k+1}> = \beta_k <r_k, r_k>. \)

\[ T_3 = T_{\text{div}} + (T_{\text{mul}} + T_{\text{div}} + T_{\text{add}}) + T_{\text{mul}}. \]

4. \( r_{k+1} = r_k - \alpha_k q_k, x_{k+1} = x_k + \alpha_k p_k, p_{k+1} = r_{k+1} + \beta_k p_k \)

\[ T_4 = 3 \frac{N}{P} T_{\text{muladd}}. \]

Let \( T_{\text{PCG}} \) be the cost of the coarse grain parallel CG algorithm per iteration. By adding the cost in each step, we have the following result:

\[
T_{\text{PCG}} = T_1 + T_2 + T_3 + T_4
= z \frac{N}{P} T_{\text{muladd}} + 2(S_C + m T_C) + \frac{N}{P} T_{\text{muladd}} + \frac{N}{P} T_{\text{muladd}} + d(S_C + 2T_C + 2T_{\text{add}}) + T_{\text{div}} + (T_{\text{mul}} + T_{\text{div}} + T_{\text{add}}) + T_{\text{mul}} + 3 \frac{N}{P} T_{\text{muladd}}
= [2(z + 5) \frac{N}{P} + (5 + 2d)] T_{\text{fop}} + (2 + d) S_C + (2m + 2d) T_C,
\]

where \( T_{\text{PCG}} \) is the cost of the coarse grain parallel CG algorithm per iteration, \( N \) is the number of variables, \( P \) is the number of processors \( (2^d) \), \( z \) is the average number of variables per row, \( T_{\text{add}}, T_{\text{div}} \) and \( T_{\text{mul}} \) are the cost of addition, division and multiplication per floating-point operation respectively. \( T_{\text{muladd}} \) is the cost of multiplication and addition per floating-point operation. \( T_{\text{fop}} \) is one floating-point operation (\( \simeq T_{\text{add}} \simeq T_{\text{div}} \simeq T_{\text{mul}} \simeq \frac{1}{2} T_{\text{muladd}} \)).

### 6.2.2 The Parallel CG Algorithm in a Faulty Hypercube

The parallel CG algorithm in a faulty hypercube requires a linear array topology to perform 1-dimensional local communication and the global data communication scheme to perform the operation of \textit{Global Sum} and \textit{Global Broadcast}. First, the linear array embedding algorithms developed, such as Algorithm \textit{LNRLATIN}, Algorithm \textit{LNRCSQ} and Algorithm \textit{LNRDCC}, can be used to embed a linear array in the faulty hypercube. Then the unused nodes are marked as faulty nodes. Next,
the data communication schemes such as Global Sum operation, Global Broadcasting operation, and GS-GB with partial Exchange-Add operation can be used to perform the required data exchange for implementing the parallel CG algorithm in the faulty hypercube. Thus, the performance model of one iteration of the coarse grain parallel CG algorithm in a faulty hypercube, denoted as $T_{PCGF}$, can be derived as follows:

1. Form $q_k = A p_k$ (local communication)
   
   $T_1 = \text{Parallel computation } (Ap_k) + \text{Local communication}$
   
   $= z \frac{N}{p^*} T_{muladd} + 2(S_C + mT_C)$.

2. a. Form $<p_k, q_k>$ and $<q_k, q_k>$
   
   (in one GS-GB communication step or with partial EA operation)
   
   $T_2 = \text{Parallel inner product } + \text{GS-GB communication}$
   
   $= \frac{N}{p^*} T_{muladd} + \frac{N}{p^*} T_{muladd} + d(S_C + 2T_C + 2T_{add}) + d(S_C + 2T_C)$.
   
   or
   
   $T_{2ea} = \text{Parallel inner product } + \text{GS-GB communication with partial Exchange-Add operation}$
   
   $= \frac{N}{p^*} T_{muladd} + \frac{N}{p^*} T_{muladd} + (d + 1)S_C + 2(d + 1)T_C + 2dT_{add}$.

3. a. $\alpha_k = \frac{<r_k, r_k>}{<p_k, q_k>}$, b. $\beta_k = \alpha_k \frac{<q_k, q_k>}{<p_k, q_k>}$ - 1.
   
   c. $<r_{k+1}, r_{k+1}> = \beta_k <r_k, r_k>$
   
   $T_3 = T_{div} + (T_{mul} + T_{div} + T_{add}) + T_{mul}$.

4. $r_{k+1} = r_k - \alpha_k q_k$, $x_{k+1} = x_k + \alpha_k p_k$, $p_{k+1} = r_{k+1} + \beta_k p_k$
   
   $T_4 = 3 \frac{N}{p^*} T_{muladd}$.

Let $T_{PCGF}$ be the cost of one iteration of the coarse grain parallel CG algorithm by GS-GB operation in a faulty hypercube and $T_{PCGF_{ea}}$
be the cost of one iteration of the coarse grain parallel CG algorithm by GS-GB with partial Exchange-Add operation in a faulty hypercube. By adding the cost in each step, we have the following result:

\[ T_{PCGF} = T_1 + T_2 + T_3 + T_4 \]

\[ = z \frac{N}{P*} T_{muladd} + 2(S_C + mT_C) + \frac{N}{P*} T_{muladd} + \frac{N}{P*} T_{muladd} + d(S_C + 2T_C) + d(dS_C + 2T_C) + T_{div} + (T_{mul} + T_{div} + T_{add}) + T_{mul} + 3\frac{N}{P*} T_{muladd} \]

\[ = [2(z + 5)\frac{N}{P*} + (5 + 2d)]T_{fop} + (2 + 2d)S_C + (2m + 4d)T_C. \]

or with partial Exchange-Add operation,

\[ T_{PCGF_{ea}} = T_1 + T_{2ea} + T_3 + T_4 \]

\[ = z \frac{N}{P*} T_{muladd} + 2(S_C + mT_C) + \frac{N}{P*} T_{muladd} + \frac{N}{P*} T_{muladd} + (d + 1)S_C + (d + 1)2T_C + d2T_{add} + T_{div} + (T_{mul} + T_{div} + T_{add}) + T_{mul} + 3\frac{N}{P*} T_{muladd} \]

\[ = [2(z + 5)\frac{N}{P*} + (5 + 2d)]T_{fop} + (3 + d)S_C + (2m + 2d + 2)T_C. \]

where \( P* \) is the number of nodes in the linear array embedded and \( z \) is the average number of variables per row. A comparison of the parallel CG Algorithm in a healthy hypercube and the parallel CG Algorithm in a faulty hypercube is given in Table 11.

The performance degradation \( (T_{increase}) \) of parallel CG algorithm in a faulty hypercube can be expressed as follows:

\[ T_{increase} = T_{PCGF} - T_{PCG} = 2(z + 5)N \frac{P - P*}{P*P} T_{fop} + dS_C + 2dT_C, \]

when the GS-GB operation is used, or

\[ T_{increase} = T_{PCGF_{ea}} - T_{PCG} = 2(z + 5)N \frac{P - P*}{P*P} T_{fop} + S_C + 2T_C, \]
<table>
<thead>
<tr>
<th>Step</th>
<th>Parallel CG in Healthy hypercube</th>
<th>Parallel CG in Faulty hypercube</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>( z \frac{N}{P} T_{muladd} + 2(S_C + mT_C) )</td>
<td>( z \frac{N}{P} T_{muladd} + 2(S_C + mT_C) )</td>
</tr>
<tr>
<td>Step 2</td>
<td>( \frac{N}{P} T_{muladd} + \frac{N}{P} T_{muladd} + d(S_C + 2T_C + 2T_{add}) ) (EA operation)</td>
<td>( \frac{N}{P} T_{muladd} + \frac{N}{P} T_{muladd} + d(S_C + 2T_C + 2T_{add}) + d(S_C + 2T_C) ) (GS-GB Operation) or ( \frac{N}{P} T_{muladd} + \frac{N}{P} T_{muladd} + (d + 1)S_C + 2(d + 1)T_C + 2dT_{add} ) (GS-GB with partial EA)</td>
</tr>
<tr>
<td>Step 3</td>
<td>( T_{div} + (T_{mul} + T_{div} + T_{add}) + T_{mul} )</td>
<td>( T_{div} + (T_{mul} + T_{div} + T_{add}) + T_{mul} )</td>
</tr>
<tr>
<td>Step 4</td>
<td>( 3\frac{N}{P} T_{muladd} )</td>
<td>( 3\frac{N}{P} T_{muladd} )</td>
</tr>
<tr>
<td>Total</td>
<td>( [2(z + 5)\frac{N}{P} + (5 + 2d)]T_{fop} + (2 + d)S_C + (2m + 2d)T_C )</td>
<td>( [2(z + 5)\frac{N}{P} + (5 + 2d)]T_{fop} + (2 + 2d)S_C + (2m + 4d)T_C ) (GS-GB) ( [2(z + 5)\frac{N}{P} + (5 + 2d)]T_{fop} + (3 + d)S_C + (2m + 2d + 2)T_C ) (Partial EA)</td>
</tr>
</tbody>
</table>

Table 11: Comparison of performance
when the partial Exchange-Add operation can be used.

Fig. 21 shows the performance degradation (increase in execution time) of parallel CG algorithm in a faulty hypercube by using Global Sum-Global Broadcast operation with respect to the number of processors used in the faulty cube for different dimensions of hypercubes. Fig. 22 shows the performance degradation of the parallel CG algorithm in a faulty hypercube by using Global Sum-Global Broadcast with partial Exchange-Add operation with respect to the number of processors used in faulty cube for different dimensions of hypercubes.

6.3 Application to the Parallel Sorting Algorithm

6.3.1 The Parallel Sorting Algorithm on a 2-D Mesh Embedded in a Healthy Hypercube

The parallel sorting algorithm developed by Thompson and Kung [45] can be implemented in a healthy hypercube. First, a 2-D mesh required is embedded in a healthy hypercube by finding two Gray code sequences $L_{m_1}$ and $L_{m_2}$ for the $m_1$ and $m_2$ sides of the mesh respectively and forming the required mesh by the Cartesian product $L_{m_1} \times L_{m_2}$. Then the sorting algorithm is implemented on the 2-D mesh embedded. The $M(j, k)$ sorting algorithm [45] is recursively specified as follows:

$M_1$: If $j > 2$, perform a single interchange step on even rows so that columns contain either all evens or all odds. If $j=2$, the columns are already segregated, so nothing else needs to be done.

$M_2$: Unshuffle each row.

$M_3$: Merge by calling algorithm $M(j, k/2)$ on each half of the array.

$M_4$: Shuffle each row.
Figure 21: Performance degradation of the parallel CG algorithm using GS-GB in a faulty hypercube
Figure 22: Performance degradation of the parallel CG algorithm using GS-GB with partial EA in a faulty hypercube
$M_5$: Interchange on even rows.

$M_6$: Comparison-interchange adjacent elements (every even with the next odd)

Figure 23 presents an example illustrating merge sorting steps in the $M(4, 4)$ sorting algorithm [45]. Steps $M_1$ and $M_2$ unshuffle the elements. The odd subsequences and the even subsequences are merged recursively by Step $M_3$. Steps 4 and 5 shuffle the odd and even together. Step $M_6$ does the final comparison interchange. The final sorted $4 \times 4$ 2-D mesh array has snake-like row major ordering.

In [45], the performance measures are defined as $T_R$ and $T_C$. $T_R$ is the routing time required to move one item from a PE to one of its neighbors, and $T_C$ is the comparison time required for one comparison step. However, for a 2-D mesh embedded in a hypercube, $T_R$ is equal to $S_C + lT_t$, where $S_C$ is the set-up time and $T_t$ is the transmission time per byte, and $l$ is the number of bytes in one item to be sorted. Let $T_{cmp}$ be equal to $T_C$ which is the time required for one comparison step. Therefore, it means that a comparison-interchange step between two items in adjacent processors can be done in time $(S_C + lT_C + T_{cmp})$ provided that a bidirectional communication mechanism is available. The time performance for each step can be expressed as follows:

$M_1 : S_C + lT_t$.

$M_2 : \frac{k-2}{2} (S_C + lT_t)$.

$M_3 : T_M(j, k/2)$.

$M_4 : \frac{k-2}{2} (S_C + lT_t)$.

$M_5 : S_C + lT_t$. 
Figure 23: Merge sorting steps in the $M(4,4)$ sorting algorithm
In general, the performance of the \( M(j, k) \) algorithm implemented in a \( j \times k \) mesh can be expressed recursively as follows:

\[
T_{M(j,k)} = SC + lT_l + \frac{k-2}{2}(SC + lT_l) + T_{M(j,k/2)} + \\
\frac{k-2}{2}(SC + lT_l) + SC + lT_l + SC + lT_l + TC \\
= (k + 1)SC + (k + 1)lT_l + TC + T_{M(j,k/2)},
\]

where \( T_{M(j,2)} = 4SC + 4lT_l + 2TC \) and \( k = 2^i, i = 1, 2, 3 \ldots \).

### 6.3.2 The Parallel Sorting Algorithm on a 2-D Mesh Embedded in a Faulty Hypercube

The parallel sorting algorithm stated in the previous section can be applied to a faulty hypercube. In a faulty hypercube, the 2-D mesh embedding schemes developed such as Algorithm \textit{MESHLATIN}, Algorithm \textit{MESHCUBE}, and Algorithm \textit{MESHDCC} are applied to find a \( j \times k \) mesh for performing the required parallel sorting algorithm. If the required \( j \times k \) mesh is found, the \( M(j, k) \) sorting algorithm is applied. The performance of \( M(j, k) \) algorithm is equal to \( T_m(j,k) \). The performance difference of the parallel sorting algorithm in a healthy hypercube and a faulty hypercube depends on the maximum value of \( k_1 \) in the \( j \times k_1 \) mesh which can be embedded in the faulty hypercube, where \( k_1 \) is a value of power 2.

Let \( T_{diff} \) denote the performance time difference. It can be expressed as follows:

\[
T_{diff} = T_m(j,k) - T_m(j,k_1) + T_{M(j,k/2)} - T_{M(j,k1/2)} \\
= (k - k_1)SC + (k - K_1)lT_l + T_{M(j,k/2)} - T_{M(j,k1/2)}.
\]
6.4 Summary

The embedding and data communication schemes have been applied to the parallel CG algorithm and the parallel sorting algorithm in a faulty hypercube. The performance of the parallel CG algorithm in a healthy hypercube and a faulty hypercube are compared. The performance degradation of the parallel CG algorithm is mainly due to the reduction in the number of processors; while, the performance degradation of the parallel sorting algorithm depends on the maximum value of \( k_1 \) in the \( j \times k_1 \) mesh which can be embedded in the faulty hypercube, where \( k_1 \) is a value of power 2. Note that the embedding and data communication schemes developed can be applied to any parallel application which require 1-dimensional local communication (or 2-dimensional local communication), Global Sum operation, and Global Broadcast operation.
CHAPTER VII

CONCLUSION AND FUTURE RESEARCH

7.1 Conclusion

In this research, algorithms for embedding linear arrays or 2-D meshes, and algorithms for performing data communication in a faulty hypercube have been developed. The embedding and data communication schemes developed have been applied to the parallel Conjugate Gradient (CG) algorithm and the parallel sorting algorithm. Both the time performance model and the performance degradation of each algorithm in faulty hypercubes have been evaluated.

Algorithms developed for embedding linear arrays include Algorithm **LNR-CUBE**, Algorithm **LNRLATIN**, Algorithm **PLNRLATIN**, Algorithm **LNRCSSQ**, Algorithm **PLNRCSSQ** and Algorithm **LNRDCC**. Algorithm **LNR-CUBE** applies the algorithm proposed in [33] to find a maximum fault-free subcube, then embeds the desired linear array in it. It may not find the maximum array in a faulty hypercube. Algorithm **LNRLATIN** was developed to overcome the limitation of Algorithm **LNR-CUBE**. Algorithm embeds a linear array \( L_m \) of length \( m \) in a faulty \( d_l \)-cube with \( r_l \) faulty nodes and \( l_l \) faulty links by making use of a modification of the Latin multiplication method [27], where \( m \leq 2^{d_l} \). It always finds the maximum array. However, it has a very pessimistic time complexity. Nevertheless it has been shown that hypercube embedding is NP-complete [36]. Any algorithm for embedding linear arrays in a hypercube will have an exponential-time complexity.
and no algorithms of polynomial time complexity can be found. Algorithm PLNLATIN is a parallel version of Algorithm LNRLATIN. A faster algorithm, namely LNRCSQ, first uses the variations of the Coordinate sequence approach to embed a linear array in a faulty hypercube. It may not find a linear array with the maximum length because it does not try all the healthy links of a node when this node is visited for embedding. This is different from Algorithm LNRLATIN which always finds the linear array with the maximum length. Algorithm PLNRCSQ is a parallel version of Algorithm LNRCSQ. Algorithm LNRDCC first embeds a linear array of length $2^d$ in a d-cube that has the Direct-Connect Capability by making use of the Coordinate sequence approach, then obtains the linear array directly by removing the faulty nodes. Algorithm BYPASS was proposed to be resided in the routing chip of each node. In this approach, it is assumed that the routing chip (DCH) and links are fault-free. With this assumption, Algorithm LNRDCC is the best algorithm for embedding linear array in faulty hypercubes.

Algorithms developed to embed 2-D meshes include Algorithm MESHCUBE, Algorithm MESHLATIN, Algorithm PMESHLATIN1, Algorithm PMESHLATIN2, Algorithm MESHDCC and Algorithm PMESHDCC. Algorithm MESHCUBE uses the algorithm proposed in [33] to find a maximum fault-free subcube, then embeds the desired mesh in the maximum fault-free subcube. It may not find the maximum 2-D meshes in a faulty hypercube. Algorithm MESHLATIN was developed to overcome this limitation. It first chooses $d_1$ coordinates among the $d$ coordinates to construct the $m_1$ side of the $m_1 \times m_2$ mesh, where $d_1 = \lceil \log_2 m_1 \rceil$, and then constructs the $m_2$ side of the mesh by making use of the remaining $(d-d_1)$ coordinates. The coordinates of the $d_1$-tuple and the coordinates of the $(d-d_1)$-tuple are selected among the coordinates of the $d$-tuple exclusively. It always finds the maximum 2-D meshes. Although the worst-case time complexity
is exponential, the experimental results show that the performance is acceptable for up to $d=9$. Algorithm \textit{PMESHLATIN1} is one of the parallel approaches of Algorithm \textit{MESHLATIN}. It distributes the fault list and uses Algorithm \textit{PLNRLATIN} in each healthy node to construct both sides of the desired 2-D mesh. Algorithm \textit{PMESHLATIN2} is the other parallel approach of embedding 2-D mesh in a faulty hypercube. It distributes the partitioned combinations of choosing $d_1$ coordinates among the $d$ coordinates for constructing the $m_1$ side of the mesh to each healthy node and uses Algorithm \textit{MESHLATIN} to find the desired 2-D mesh in each healthy node. Algorithm \textit{MESHDCC} first embeds a $2^{d_1} \times 2^{d_2}$ mesh represented by a 2-D Gray code ordering in a faulty hypercube, where $d_1 \geq \lfloor \log_2 m_1 \rfloor$, $d_2 \geq \lfloor \log_2 m_2 \rfloor$, and $(d_1 + d_2) \leq d$, then removes faulty columns and/or rows. Finally, it forms the desired $m_1 \times m_2$ mesh from the healthy columns and rows directly. Algorithm \textit{BYPASS} uses the properties of the \textit{Coordinate sequence} and the identifications of the source and the destination to determine a unique predefined path. Algorithm \textit{PMESHDCC} is a parallel approach of Algorithm \textit{MESHDCC}. It distributes the partitioned combinations of choosing $d_1$ coordinates among the $d$ coordinates for constructing the $m_1$ side of the mesh to each healthy node and uses Algorithm \textit{LNRDCC} in each healthy node to find both sides of the desired 2-D mesh.

In addition, algorithms of performing data communication in a faulty hypercube are developed. The splitting operation is developed to find the necessary parameters for performing the desired data communication in a faulty hypercube. The \textit{Global Sum} operation accumulates the partial sum in each node to a collecting node in a faulty hypercube. The \textit{Global Broadcast} operation distributes a unique message from the collecting node to all the other healthy nodes in a faulty hypercube. The \textit{Global Sum and Global Broadcast with partial Exchange-Add} operation
in a faulty hypercube performs the accumulation of all partial sum's in each node and distribution of the sum accumulated to all nodes. Finally, the embedding and data communication schemes developed are applied to the parallel Conjugate Gradient algorithm [9,20] and the parallel sorting algorithm [45] in a faulty hypercube. Any parallel algorithms which require 1-dimensional communication (or 2-dimensional communication), Global Sum operation, and Global Broadcast operation can be implemented in a faulty hypercube by applying the embedding and data communication schemes developed.

7.2 Research Extensions

Although algorithms for embedding linear arrays/2-D meshes, and data communication in faulty hypercubes have been developed, some related research extensions still open. They are suggested as follows:

1. Investigating two or three dilation embeddings.
   Unit dilation embedding is considered in this research. However, embeddings in hypercube can be accomplished by two or three dilation. It is useful to investigate two or three dilation embedding in a hypercube.

2. Embedding other topologies in faulty hypercubes.
   We have developed algorithms for embedding linear arrays and 2-D meshes in a faulty hypercube. Some applications need other topologies such as trees, rings, and multi-dimensional meshes. Embedding these topologies in a faulty hypercube needs to be investigated.

3. Allocating different topologies in a multi-user environment.
   In a multi-user system, different users may ask different topologies for their
applications. The allocation of different topologies to different users is a promising research area.

4. Embedding different topologies in the augmented hypercube.

A hypercube can be augmented by adding spare nodes and links for fault-tolerance. Embedding different topologies in the augmented hypercube will be easier and useful.
LIST OF REFERENCES


