A Fault-Tolerant Tree Communication Scheme for Hypercube Systems

Yuh-Rong Leu and Sy-Yen Kuo

Abstract—The *tree communication* scheme was shown to be very efficient for global operations on data residing in the processors of a hypercube with time complexity of $O(\log_2 N)$, where N is the number of processors. This communication scheme is very useful for many parallel algorithms on hypercube multiprocessors. If a problem can be divided into independent subproblems, each subproblem can first be solved by one of the processors. Then, the tree communication scheme is invoked to merge the subresults into the final results. All the algorithms for problems with this property can benefit from the tree communication scheme. We propose a more general and efficient tree communication scheme in this paper. In addition, we also propose fault-tolerant algorithms for the tree communication scheme, by exploiting the unique properties of the tree communication scheme. The computation and communication slowdown is small (< 2) under the effect of multiple link and/or node failures.

Index Terms—Hypercube, failures, tree communication, uniform data distribution, fault-tolerance.

1 Introduction

Fault tolerance is a significant property of modern computer systems [1]. In various applications, high reliability is often required. For example, computers used for controlling a nuclear power plant should be very reliable because a small failure within the control system may cause catastrophic damages. Military defense systems also require high reliability under harsh environments, for their fatal influence on human lives. In many space applications, if some parts of a space ship fail, no repair is possible. In such cases, fault tolerance capabilities must be included in the space systems to make them operate correctly under hardware failures. Computers used for financial applications also require high reliability since their failures may jeopardize customers' economic well-being. In general, fault tolerance gives computer systems robustness and resistance against physical failures and/or operational faults.

Hypercube multiprocessors have a good inherent fault-tolerance property that single component failure may not jeopardize the whole system. Therefore, much interest has been paid on the hypercubes. An n-dimensional (n – D) hypercube has 2^n nodes and $n2^{n-1}$ links. Each node in a hypercube is connected to n adjacent nodes by n links [2]. In [3], a matrix-vector multiplication algorithm using the *tree communication* scheme was proposed. Fault-tolerance issues of this matrix operation algorithm were discussed in [4], but only node failures were considered. Uniform Data Distribution (UDD) strategy was used by the authors of [4] to resolve the problem of single node failure on a hypercube. By redistributing the data to all nonfaulty nodes uniformly, the matrix operation can still be completed, but with degraded performance. All these previous works concentrated on matrix operations. The tree communication scheme can also be applied on other types of computation. If a problem can be divided into independent subproblems, each subproblem can first be solved by one of the processors. Then, the tree communication scheme is invoked to merge the subresults into the final results. Algorithms for solving problems with the above property can benefit from this communication scheme. In this paper, we present a generalized tree communication scheme and describe how it can be utilized for practical applications. Furthermore, both off-line and on-line fault-tolerant algorithms for the tree communication scheme are proposed. Our approach is shown to be able to tolerate multiple link failures and/or node failures.

A communication tree is a binomial tree embedded in a hypercube, and the communication directions are from the leaves up to the root. An n-level binomial tree BT_n can be constructed recursively by connecting the roots of two BT_{n-1}s and assigning one of the roots of the BT_{n-1}s as the root of BT_n. Without failures, the nodes and links involved in a broadcasting session also form a binomial tree except that the communication directions are reversed (from the root down to the leaves). Extensive literature exists on fault-tolerant broadcasting, multicasting, and point-to-point communication in hypercubes [5], [6], [7], [8], [9], [10]. Park and Bose [7] proposed an n + 1-step fault-tolerant broadcasting algorithm in the presence of link failures. Raghavendra et al. [8] presented the concept of free dimension and exploited this concept to devise an n + 1-step broadcasting algorithm in the presence of node failures. Even though their works are optimal for fault-tolerant broadcasting, it is not efficient to use their results on global operations. The scenario of using the n + 1-step fault-tolerant broadcasting strategy is as follows:

1) A node is arbitrarily chosen to be the source of broadcasting, and then it broadcasts a message to all the other nodes.

• *The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, Republic of China. E-mail: sykuo@cc.ee.ntu.edu.tw.*

Manuscript received May 21, 1994; revised October 1995. For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96044.

2) Each node can decide which way to send its own subresults to be merged according to the message received. The global operations can be accomplished after $n + 1$ merging steps.

Therefore $2(n + 1)$ parallel steps are necessary for global operations. However, the detail of this scenario has not yet been explored. There are difficulties in using this approach and we leave them as open problems. The approach proposed in this paper takes just $O(n + k)$, where $0 \leq k < n$, parallel steps to accomplish global operations in the presence of link/node failures.

First, we assume that only link failures exist in the hypercube system. The fault-tolerant algorithms presented in this paper are based on this assumption. Later, we will discuss how these algorithms can be applied to node failures. Our approach has two phases. In the first phase, we devise an algorithm that can find a communication tree with as few faulty links as possible. This is a static fault-tolerance phase because the majority of faulty links would be excluded from the communication tree before execution. In the second run-time phase, the data messages are routed dynamically according to the link fault pattern in the communication tree. Our strategy for tolerating node failures is similar to that of [4] (UDD). The data is uniformly distributed to all non-faulty nodes, and a faulty node is treated as a node with all the links to its neighbors being faulty. Then, the algorithm to find a communication tree and the run-time fault-tolerant algorithm can be applied.

Since link failures are randomly distributed, it is often the case that a communication tree with no link failure does not exist if the number of link failures in the hypercube is significant. Even if the number of link failures is small, finding a fault-free communication tree in a faulty hypercube is still not a simple matter (see Subsection 2.2). So, instead of finding a fault-free communication tree which may not exist, our approach is based on a two-phase strategy. With this strategy, the tree communication scheme can still work even when a fault-free communication tree does not exist. This is one of the advantages of our approach.

Some notations and the style of algorithm representation are adopted from [5]. However, the goal of [5] is very different from ours. The topic of [5] is point-to-point routing and broadcasting in hypercubes with faulty nodes. The routing pattern we consider is binomial-tree-like. The unique properties of the tree communication scheme are exploited to devise our fault-tolerant algorithms. Furthermore, our algorithms are based on link failures.

The rest of this paper is organized as follows. Section 2 describes the generalized tree communication scheme. Section 3 presents the fault-tolerant algorithms for the tree communication scheme. Conclusions are given in Section 4. For more details on hypercubes and background knowledge for this paper, please refer to [14], [15], [16], [17], [18], [19], [20].

2 Generalized Tree Communication Scheme

Fig. 1 shows a feasible communication tree for a 4D hypercube. Note that there are four (= $\log_2 16$) communication stages. The labeling of nodes is based on the binary reflective Gray codes. Two nodes with their binary labels differing in only one bit can communicate with each other directly in one step. The arrow lines in Fig. 1 represent the directions of data flow. The node at the starting end of an arrow line is a sending node, and the one at the arrowhead is a receiving node. For instance, node 0001 sends data and node 0000 receives data at stage 0, that is, they form a send-receive pair. A receiving node receives data and then performs computations on the received data and its own data. After $\log_2 N$ ($N = 16$ in this case) stages of operation, the final results will be in node 0000. This communication scheme can be efficiently exploited by many parallel algorithms. In the following subsection, we will present several example applications.

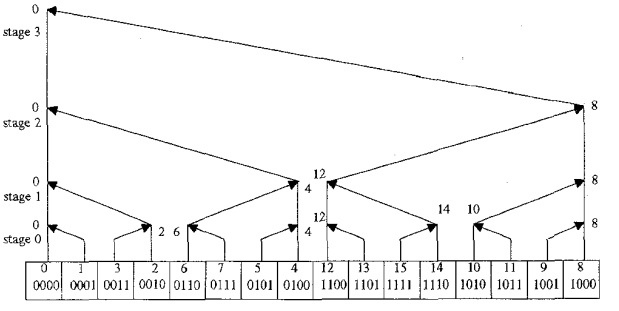


Fig. 1. The tree communication scheme for a 4D hypercube.

2.1 Applications

By labeling the processor nodes of a hypercube based on the Gray code in two dimensions, the hypercube structure can be viewed as a grid [11]. Fig. 2 illustrates such a Gray code labeling (row indexes and column indexes are Gray codes), where each cell stands for a processor node. The binary representation of the decimal number in cell (i, j) is obtained by concatenating the two binary numbers representing row i and column j, respectively. For instance, decimal number 17 in cell (2, 4) is represented by the concatenation of 10 and 001, i.e., $17_{10} = 10001_2$.

	00	01	11	10
000	0 (1,1)	8 (1,2)	24 (1,3)	16 (1,4)
001	1 (2,1)	9 (2,2)	25 (2,3)	17 (2,4)
011	3 (3,1)	11 (3,2)	27 (3,3)	19 (3,4)
010	2 (4,1)	10 (4,2)	26 (4,3)	18 (4,4)
110	6 (5,1)	14 (5,2)	30 (5,3)	22 (5,4)
111	7 (6,1)	15 (6,2)	31 (6,3)	23 (6,4)
101	5 (7,1)	13 (7,2)	29 (7,3)	21 (7,4)
100	4 (8,1)	12 (8,2)	28 (8,3)	20 (8,4)

Fig. 2. Grid embedding using the binary reflective Gray code on a 5D hypercube.

If a vector Y is to be computed by performing a multiplication of a large matrix A and a vector x, i.e., $Y = A \cdot x$, the algorithm can be expressed as follows:

1) Distribute the matrix A and the vector x to all processors evenly. Thus, each processor (i, j) acquires a portion of A denoted by A_{ij} and a portion of x denoted by x_j.
2) Each processor computes sequentially the *intermediate subvector* $y_{ij} = A_{ij} \cdot x_j$.
3) Use the *tree communication* scheme to compute the subvectors Y_is from the intermediate subvectors obtained in step 2. All Y_i' will be stored on the processors of the first column (i.e., Y_i will be on processor $(i, 1)$). These computations are the operations of global summation. The tree communication is performed horizontally (row-wise) at this step. The equation describing this computation is:

$$Y_i = \sum_j A_{ij} \cdot x_j = \sum_j y_{ij}.$$

4) Collect all Y_is into processor (1, 1) using the *tree communication* scheme again, but this time it is performed vertically (column-wise).

The tree communication scheme can also be applied to parallel sorting algorithms. A sequence of data of size s is first divided into N subsequences each of size s/N. Next, distribute each subsequence into a different the processor. Each processor then sorts its own data. All the sorted subsequences of data are finally merged using the tree communication scheme. Many other applications are viable. We should observe two important facts about the algorithms that can utilize the tree communication scheme:

1) The solutions of subproblems must be able to be obtained independently.
2) The merging operations in the tree communication session are application-specific. They may be additions, multiplications, subtractions, or even more complicated operations (such as in the example above, i.e., merging sorted subsequences).

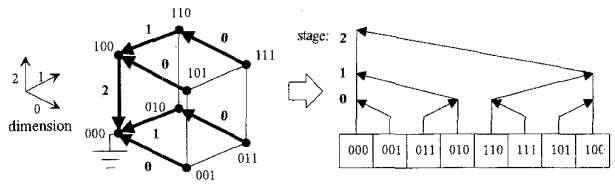
Obviously, the tree communication scheme is useful and efficient for applications on hypercube multiprocessors.

2.2 Properties and Definitions

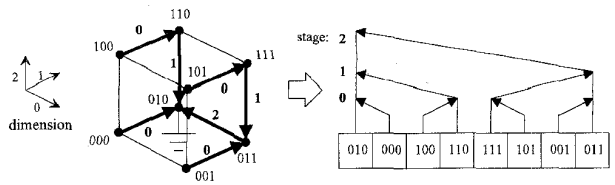
The communication tree for a hypercube is not unique, however. The tree demonstrated in Fig. 1 is the one used in [3] and gives only one of the possibilities. In Fig. 3, we show two different communication trees for a 3D hypercube. The boldfaced numbers in the figure represent the communication stage numbers. The receiver of the last stage is defined as the *sink*. We use the ground symbol for pointing out the sinks in Fig. 3. A communication tree can be determined by selecting a node as the sink and a dimension for the messages at a stage to traverse. For example, in Fig. 3b, node 010 is the sink, and the order of dimensions that the messages will traverse is (1, 2, 0), i.e., messages traverse dimension 1 at stage 0, dimension 2 at stage 1, and dimension 0 at stage 2. The order of dimensions in Fig. 3a is (0, 1, 2). We know from the above example that there are two ways to get a new communication tree:

1) choosing another node as the sink and
2) changing the order of dimensions.

The following properties give the number of feasible communication trees and the number of used links in a communication tree.



(a) Node 000 as the sink with dimension order (0, 1, 2).



(b) Node 010 as the sink with dimension order (1, 2, 0).

Fig. 3. Two example communication trees for a 3D hypercube.

PROPERTY 1. *There are* $n! \times 2^n$ *feasible communication trees for an* n*-D hypercube.*

PROOF.: Different sinks and different dimension orders determine different communication trees. An n-D hypercube has 2^n nodes, therefore we have that many choices of selecting a node as the sink. Let d_i denote the dimension on which messages at stage i traverse, then the dimension order $(d_0, d_1, d_2, \ldots, d_{n-1})$ is a permutation of $(0, 1, 2, \ldots, n-1)$. Thus there are $n!$ ways of permuting. Consequently $n! \times 2^n$ communication trees exist. □

PROPERTY 2. *The number of links used by the tree communication scheme is* $2^n - 1$.

PROOF. There are n communication stages in the tree scheme. At stage i, 2^{n-i-1} links are required. Therefore, the total number of used links is

$$\sum_{i=0}^{n-1} 2^{n-i-1} = 2^n - 1. \qquad \square$$

Since there are many feasible communication trees for a hypercube, we may find one with as few link failures as possible. An algorithm guaranteed to find an optimal communication tree (i.e., the communication tree with the fewest link failures) is inefficient because it has to check all $n! \times 2^n$ possibilities and whether the $2^n - 1$ (from Property 2) used links in a tree are faulty. Fig. 4 illustrates an algorithm for exhaustive searching. Its time complexity is $O(n! \times 2^n) \times O(2^n - 1) = O(n! \times 4^n)$. In the next section, we will propose a more efficient heuristic algorithm although it does not guarantee to find an optimal communication tree. An optimal communication tree is not always a fault-free communication tree. The reason is that fault patterns are random but the relationship between the links of a communications tree is deterministic. So, even an optimal communication

tree has been found, the tree communication scheme is not guaranteed to work. Therefore, in addition to the first phase of finding a good communication tree, we need a second fault-tolerance phase to reroute messages if link failures exist in a communication tree.

Algorithm EXHAUST
{There are $n! \times 2^n$ feasible communication trees. A tree has $2^n - 1$ links.}

begin
  repeat
    Generate 1 communication tree T;
    for all links l of T **do**
      Check if l is nonfaulty;
    if all links are nonfaulty
      then output T;
  until all possible trees have been searched;
end.

Fig. 4. Algorithm EXHAUST.

Before discussing our fault-tolerant algorithms, let us make some definitions for the tree communication scheme. The nodes sending data in the communication tree are *active* nodes, those receiving data are *passive* nodes, and the nodes doing nothing are said to be *idle*. An *active set* of stage i (denoted by A_i) is composed of all the active nodes at stage i. A *passive set* of stage i (denoted by P_i) is defined similarly. Formally, let I_i be a binary word of which the ith (the least significant bit is the 0th bit) bit is 1, and the others are 0. Let $b[i]$ denote the ith bit of a binary word b and $0^n(1^n)$ represent a binary word of all 0s(1s). What a node does at a certain stage depends on the set containing this node. This can be done by checking whether a node's binary label satisfies the rules listed below.

Node v is:

Active at stage i ($v \in A_i$) if

$$v[d_i] = r[d_i] \text{ and } v[d_j] \neq r[d_j], \forall j, 0 \leq j < i. \quad (1)$$

Passive at stage i ($v \in P_i$) if

$$v[d_i] \neq r[d_i] \text{ and } v[d_j] \neq r[d_j], \forall j, 0 \leq j < i. \quad (2)$$

Idle at stage i ($v \notin A_i$ and $v \notin P_i$) if

$$\exists j, 0 \leq j < i \text{ s.t. } v[d_j] = r[d_j]. \quad (3)$$

In the above, r is a binary word containing the information on message flow directions, and $r = sink \oplus 1^n$. For example, $r[2] = 1(0)$ means that the direction is from 1 to 0 (from 0 to 1) on dimension 2.

EXAMPLE 1. In Fig. 3b, the dimension order is (1, 2, 0), i.e., $d_0 = 1$, $d_1 = 2$, and $d_2 = 0$. And $r = 010 \oplus 111 = 101$. Observe the behavior of node 110 at each stage.

At stage 0: $v[d_0] = 110[1] = 1 \neq 0 = 101[1] = r[d_0]$. It satisfies (2), so it is passive.

At stage 1: $v[d_1] = 110[2] = 1 = 101[2] = r[d_1]$, $v[d_0] = 110[1] = 1 \neq 0 = 101[1] = r[d_0]$. It satisfies (1), so it is active.

At stage 2: $v[d_1] = 110[2] = 1 = 101[2] = r[d_1]$. It satisfies (3), so it is idle.

If a node is active at a stage, it will be idle in subsequent stages. Fig. 5 depicts the adjacency relationship among the nodes in the active set of each stage for the communication tree shown in Fig. 1. Those active nodes which are adjacent to each other are *active neighbors*.

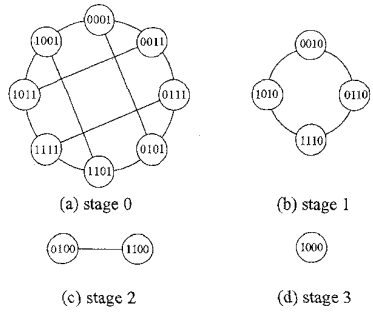


Fig. 5. The nodes in the active set of each stage and their adjacency relationships in a 4D hypercube.

An *active neighbor* of active node v at stage i is a node whose binary label differs from node v's label in only one bit and is *active* at stage i (i.e., it is in A_i). All the nodes satisfying the above conditions form a set $AN_i(v)$. An active neighbor of active node v can be found by the operation

$$aneib(v, d_j) = v \oplus I_{d_j} \quad (j > i). \quad (4)$$

An active (passive) node v at stage i can find its passive (active) neighbor $pneib(v, d_i)$ ($aneib(v, d_i)$) by the following operation:

$$pneib(v, d_i) = v \oplus I_{d_i}, \text{ if node } v \text{ is active;}$$
$$aneib(v, d_i) = v \oplus I_{d_i}, \text{ if node } v \text{ is passive.} \quad (5)$$

Notice that an active node at a certain stage may have more than one active neighbor, but it has only one passive neighbor. Also, a passive node at a certain stage has exactly one active neighbor. By the above definitions, the tree communication scheme is represented in Fig. 6.

The following theorems show more properties of the tree communication scheme.

THEOREM 1. *There are $n - i - 1$ nodes in $AN_i(v)$, where v is a node in A_i, i.e., the number of active neighbors of an active node v at stage i is $n - i - 1$.*

PROOF. If there are $N = 2^n$ nodes involved in the tree communication scheme ($N = 16$ and $n = 4$ in Fig. 1), the binary label of each node has n bits. Nodes in $AN_i(v)$ are also in A_i, i.e., $AN_i(v) \subset A_i$. They should satisfy (1), thus node v and all nodes u in $AN_i(v)$ satisfy

$$v[d_k] = u[d_k], \forall k, 0 \leq k \leq i \text{ and } \forall u \in N_i(v). \quad (6)$$

It means that node v and all its active neighbors have $i + 1$ bits of their labels in common. According to the operation for finding active neighbors in (4), there are $n - i - 1$ possible values of j that satisfy the condition $j > i$ (i and j are from 0 to $n - 1$). Consequently, there are $n - i - 1$ nodes in $AN_i(v)$. □

Algorithm TC;
{*curr* is the current mode, *aneib* stands for active neighbor, and *pneib* stands for passive neighbor.}

begin
  for $i = 0$ **to** $n - 1$ **do**
    case *curr* **of**
      $curr \in A_i$: {*curr* is active}
        Send data messages to *pneib*(*curr*, d_i) via link d_i;

      $curr \in P_i$: {*curr* is passive}
        Wait for data messages from *aneib*(*curr*, d_i);
        Receive data messages from *aneib*(*curr*, d_i);
        Merge received data and *curr*'s local data;

    endcase;
end.

Fig. 6. Algorithm TC.

THEOREM 2. *There are* 2^{n-i-1} *nodes in* A_i.

PROOF. The binary labels of the nodes in the active set of stage i (A_i) must satisfy (1). This condition assigns values to $i + 1$ bits of the label of a satisfying node, but it leaves other $n - i - 1$ bits unspecified. So there are 2^{n-i-1} nodes satisfying (1) and thus in A_i.

Another way to prove this theorem is by applying Theorem 1. Because an active node has $n - i - 1$ active neighbors and all the active nodes at stage i form a *subcube* (a hypercube of smaller dimension), there are 2^{n-i-1} nodes in the active set [5]. □

3 Fault-Tolerant Tree Communication Scheme

Without failures, the simple communication tree (node 0^n as the sink and the dimension order is (0, 1, 2, ..., $n - 1$)) can function efficiently and correctly. However, if hardware failures exist in the system, this scheme will fail. The best way to solve this problem is finding a new communication tree with no faulty link. Since the faults are randomly distributed, this goal cannot be always reached. There may still be link failures in an optimal communication tree (i.e., the communication tree with fewest link failures). Thus, the tree itself should be provided with the capability of fault tolerance if high reliability is required. In this section, we present an algorithm which can find a communication tree with as few link failures as possible and a run-time fault-tolerant tree communication algorithm. We will also discuss how node failures can be tolerated in the end of this section.

3.1 Finding a Communication Tree in the Presence of Link Failures

First, we devise a simple but efficient parallel algorithm to detect link failures. Failures are assumed to be permanent before task initialization, and no new failures will occur at run-time. In addition, we assume that the link failures are distributed evenly in the hypercube. A node can check if the links to its neighbors are faulty by sending test messages to its neighbors. Because each node has n neighbors, n test messages are sent by a node. Each node keeps an n-bit word *FAULT* to record the fault information; node v keeps *FAULT*(v). All the *FAULT*s can be collected by the front end host of the hypercube, hence we can have a global view of the fault pattern. It is based on the fact that we can find different communication trees according to different fault patterns. Fig. 7 illustrates the algorithm DETECT to detect link failures adjacent to a node. Obviously, $O(n)$ parallel steps are required.

Algorithm DETECT;
{*curr* is the current node and *neib* is the neighbor node}

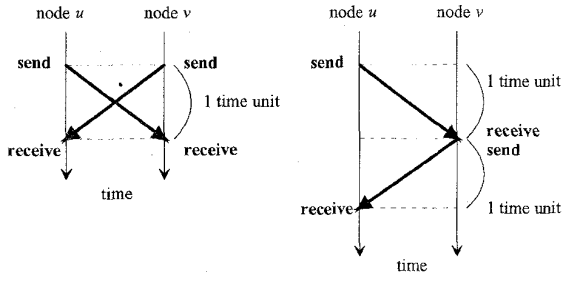
begin
  FAULT(*curr*) $\leftarrow 0^n$;
  for $i = 0$ **to** $n - 1$ **do begin**
    $neib \leftarrow curr \oplus I_i$;
    Send a test message to node *neib* via link i;
    {node *neib* also sends a test message to node *curr*}
    Delay;
    if not received a test message from node *neib*
      then *FAULT*(*curr*)[i] $\leftarrow 1$;
  endfor;
end.

Fig. 7. Algorithm DETECT for detecting link failures.

The delay operation in Algorithm DETECT is for the current node to make sure that the test message from its neighbor has passed through the link. We assume full-duplex communication, thus one time unit is required to complete each iteration. Nevertheless, if the communications are half-duplex, two time units are needed. The reason is that only one message is allowed to traverse one link at a time. It is very easy to modify the algorithm to adopt half-duplex communication. We can split each iteration into two phases. Node u with $u[i] = 1$ sends a message in the first phase and node v with $v[i] = 0$ sends a message in the second phase. Fig. 8 shows the differences between full-duplex and half-duplex communications. In the figure, node u and node v are neighbors and $u \oplus v = I_i$. Current multiprocessor machines may support multiport communication, i.e., one node can simultaneously communicate with several neighbors. If this is the case, Algorithm DETECT can be modified with little efforts to exploit such hardware support.

Fig. 9 shows an example for Algorithm DETECT. In the fig., the *FAULT* of each node is shown in an angle bracket. For instance, node 101 has two link failures on dimensions 0 and 1, hence its *FAULT* is <011>.

Each node should also keep the fault status of its neighbors. The reason is that each node must use this fault information to reroute messages in run-time for the second-level fault-tolerance. Thus, the information *FAULT* of a node is distributed to all its neighbors. In the next subsection, we will give more details.



(a) Full-duplex communication (b) Half-duplex communication

Fig. 8. Full-duplex vs. half-duplex in Algorithm DETECT.

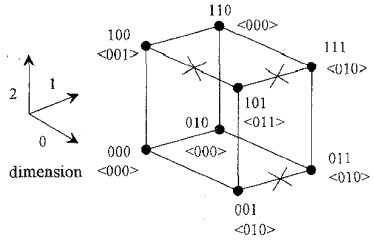


Fig. 9. The *FAULT* of each node of a faulty 3D hypercube.

The sequential algorithm for finding a communication tree in the presence of link failures is composed of two parts. A node is first selected to be the sink and the dimension order is then determined according to the fault pattern. This algorithm is shown in Fig. 10.

Because the sink (root of the communication tree) is the most important node in a communication tree, it can not have any adjacent link failures. So in the first part of the algorithm, we choose a node without any adjacent link failure as the sink. The number of active nodes decreases as the stage number i increases (see Fig. 5). Therefore, the cost of rerouting messages increases as the stage number i increases (refer to Subsection 3.2). Accordingly, we can determine from d_{n-1} to d_0. The strategy to determine d_i is:

in iteration i, we choose dimension j with minimal COST(j) as d_i, where

$$COST(j) = \sum_{\forall u \in USED\_N} \sum_{\forall k \notin USED\_D} FAULT(neib(u, j))[k]. \quad (7)$$

We can find out how many links adjacent to node u are faulty by counting the number of 1s in $FAULT(u)$, i.e., Σ_k $FAULT(u)[k]$. The neighbor of node u on dimension j is denoted as $neib(u, j)$. Since the dimensions in *USED_D* have been used in the communication tree before iteration i, and they have no effect in later iterations, we exclude these dimensions from counting the number of failures. Fig. 11 shows the relationship between sets *USED_N*, ~*USED_D*, and $neib$(*USED_N*, ~*USED_D*), where "~" is set complement operation. This makes the idea of the cost function much clearer. After the sink and the dimension order have been determined by the front end host, this information is distributed to all nodes. Each node then uses the information received to decide what to do, as stated in Subsection 2.2, at each stage. Let's look at an example on how Algorithm FIND_TREE works.

Algorithm FIND_TREE (*sink*, dimension order);
{*USED_N*: the set of nodes having been used in the tree;
USED_D: the set of dimensions having been used in the tree.
All sets are initially empty.}

begin
{choose a node as the *sink*}
for all nodes v **do**
if $FAULT(v) = 0^n$ **then begin**
Choose node v as the *sink*;
$USED\_N \leftarrow USED\_N + \{v\}$;
Exit loop;
endif;

{determine d_i; ($d_0, d_1, \ldots, d_{n-1}$) is the dimension order}
for $i = n - 1$ **downto** 1 **do begin**
for all dimensions not in *USED_D* **do begin**
Choose dimension j with minimal COST(j);
$d_i \leftarrow j$;
endfor;
$USED\_D \leftarrow USED\_D + \{d_i\}$;
$USED\_N' \leftarrow USED\_N$;
for all nodes u in $USED\_N'$ **do**
$USED\_N \leftarrow USED\_N + \{neib(u, d_i)\}$;
endfor;
$d_0 \leftarrow$ the remaining dimension not in *USED_D*;
end.

Fig. 10. Algorithm FIND_TREE for finding a communication tree.

EXAMPLE 2. Assume the fault pattern is the same as in Fig. 9.

Step 1: The algorithm finds $FAULT(000)$ = <000>, so it picks this node as the *sink*. Now, *USED_N* = {000} and *USED_D* = {}.

Step 2: The neighbors of the node in *USED_N* (000) are 001, 010, and 100. Thus COST(0) = 1, COST(1) = 0, and COST(2) = 1. Recall the definition of the cost function. Dimension 1 is the one with minimal COST, so $d_2 \leftarrow 1$. *USED_N* = {000, 010} and *USED_D* = {1}.

Step 3: The neighbors of the nodes in *USED_N*, which are 000 and 010, are 001 and 011 (100 and 110) along dimension 0 (2). Therefore COST(0) = 0 and COST(2) = 1. So $d_1 \leftarrow 0$. *USED_N* = {000, 010, 001, 011} and *USED_D* = {1, 0}.

Step 4: The remaining dimension is dimension 2, so $d_0 \leftarrow 2$.

We get a communication tree with node 000 as the *sink* and the dimension order (2, 0, 1), and there is no faulty link in the tree. Fig. 12 demonstrates these steps. The shaded nodes are the nodes in *USED_N*.

THEOREM 3. *Algorithm FIND_TREE will succeed if the number of faulty links is smaller than* 2^{n-1}.

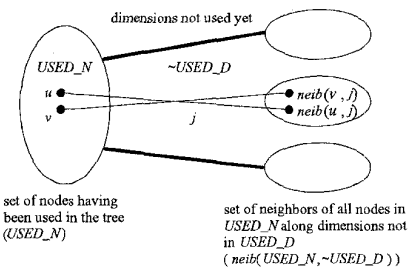


Fig. 11. Cost function.

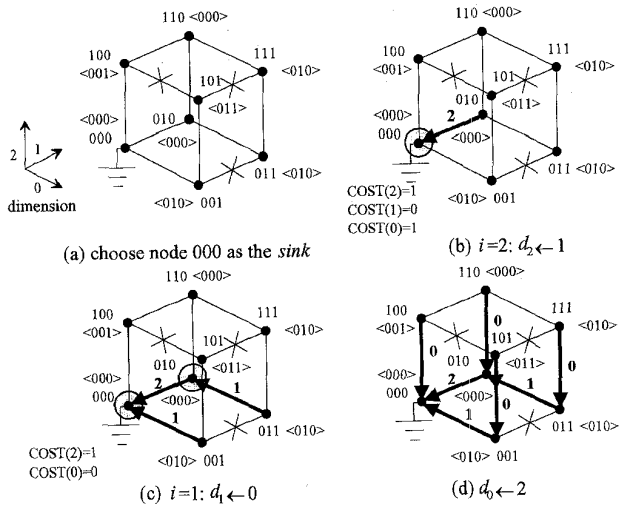


Fig. 12. An example for Algorithm FIND_TREE.

PROOF. The adjacent links of the sink cannot be faulty. Algorithm FIND_TREE fails if it cannot choose one node as the sink. This happens when every node has at least one adjacent faulty link. There are 2^{n-1} faulty links in a hypercube if every node has only one adjacent faulty link. This is because that each link has two nodes at its both ends and there are 2^n nodes in a hypercube. Consequently, if the number of link failures is smaller than 2^{n-1}, there is at least one node with all adjacent links being nonfaulty. Algorithm FIND_TREE will select this node as the sink. □

THEOREM 4. *The complexity of Algorithm FIND_TREE is O(N), where N is the number of nodes.*

PROOF. The complexity of selecting the sink is obviously $O(N)$. The analysis in determining the dimension order is more complicated. At iteration i, the number of nodes in *USED_N* is 2^{n-i-1}, and the number of dimensions not in *USED_D* is $i+1$. Accordingly, $2^{n-i-1} \cdot (i+1)$ iterations are necessary for computing COST(j). And COST(j) is computed $i+1$ times in the second for-loop. The third for-loop for updating *USED_N* needs 2^{n-i-1} iterations. Therefore, the total number of iterations for determining the dimension order is:

$$\sum_{i=n-1}^{1}[(i+1)\cdot 2^{n-i-1}(i+1)+2^{n-i-1}] = 6\cdot 2^n - n^2 - 4n - 7. \quad (8)$$

It is dramatically $O(N)$ ($N = 2^n$). Therefore, the complexity of Algorithm FIND_TREE is $O(N)$, i.e., linear with the number of nodes. □

The tree communication scheme is suitable for large-grain-size parallel applications. If the grain size is small, the communication overhead would overwhelm the computation speed-up brought by parallelism. Besides, in most applications on hypercube multiprocessors, a full-size hypercube is seldom required. The hypercube is often partitioned into several subcubes, and different tasks are run on the subcubes. Consequently, the typical number of nodes N will be small, e.g., $N = 128$. This fact makes Algorithm FIND_TREE very practical for important applications.

3.2 Run-Time Fault Tolerance of the Tree Communication Scheme

Since the faults are randomly distributed, the communication tree determined by Algorithm FIND_TREE may still have faulty links. This is the reason why a second-level fault tolerance is required. An active node except the one at stage $n-1$ has at least one active neighbor. If an active node cannot send messages to its passive neighbor, it instead sends these messages to its active neighbors.

Fig. 13 shows the algorithm FTTC (Fault-Tolerant Tree Communication). The capability of fault tolerance is embedded in the tree communication scheme. If there is no link failure in the tree, Algorithm FTTC behaves just like a normal tree communication, except that little delay is introduced. There are three types of node—active, passive, and idle. Each node performs according to the role it has to play at each communication stage. However, the links from the active node, which has difficulty in sending messages to its passive neighbor, to its active neighbors may also be faulty. Our algorithm is able to resolve this potential problem, too.

In Algorithm FTTC, the current node first check whether itself is active or passive at stage i. If it is active, then it checks the link between itself and its passive neighbor. If this link is faulty, it cannot send data messages to its passive neighbor. So it tries to send the data messages to all its active neighbors. Recall that the tree communication scheme is utilized for application-specific global merging operations for the data in every node. The order of merging and the nodes performing the merging operations are irrelevant. Hence there is no problem that the active neighbors, instead of the passive neighbor, of the current node perform the merging operations. For load balancing, each active neighbor of the current node receives only one partition of the data messages. But the links from the active node, which has a faulty link to its passive neighbor, to its active neighbors might also be faulty, so the number of partitions m should be determined. The function W which returns the number of link failures between an active node v and its active neighbors is defined as (refer (4)):

$$W(v) = \sum_{k=i+1}^{n-1} FAULT(v)[d_k]. \quad (9)$$

And from Theorem 1 we know an active node has $n-i-1$ active neighbors, thus m is equal to $n-i-1-W(curr)$. If m is 0, the current node tries to send data messages to its passive neighbor using *fault-tolerant point-to-point* routing algo-

rithms via a detour path. Otherwise, the current node divides the data messages into m partitions and then sends one partition to an active neighbor if the link to this active neighbor is non-faulty. Due to the assumption that the link failures are distributed evenly in the hypercube and the strategy of Algorithm FIND_TREE, the probability that $m = 0$ is small. Accordingly, we omit this case in the later analysis.

```
Algorithm FTTC;
{curr is the current node, aneib stands for active neighbor,
and pneib stands for passive neighbor.}

begin
  for i = 0 to n − 1 do
    case curr of
curr ∈ A_i : {curr is active}
      if FAULT(curr)[d_i] = 1 then begin
      {The link from curr to its passive neighbor is faulty.}
      m ← n − i − 1 − W(curr);
      if m = 0 then send data messages to pneib(curr, d_i)
      via a detour path;
      else begin
        Divide the data messages into m partitions;
        for j = i + 1 to n − 1 do
          if FAULT(curr)[d_j] = 0
          {The link to the active neighbor along dimen-
           sion d_j is nonfaulty.}
           then send one of the partitions to aneib(curr, d_j)
           via link d_j;
        endelse;
      endif
      else begin
      {curr has no difficulty but may have to help its active
       neighbors.}
      {check whether curr's each active neighbor can send
       messages to its passive neighbor}
       for j = i + 1 to n − 1 do
         if FAULT(aneib(curr, d_j))[d_i] = 1 and FAULT(curr)[d_j]
          = 0 then begin
         {The active neighbor along dimnsion d_j cannot
          send the messages and the link to it is nonfaulty.}
           Wait for data messages from aneib(curr, d_j);
           Receive data messages from aneib(curr, d_j);
           Merge received data and curr's local data;
         endif;
      Send data messages to pneib(curr, d_i) via link d_i;
  endelse;

curr ∈ P_i : {curr is passive}
      if FAULT(curr)[d_i] = 0 then begin
      {The link to curr's active neighbor is nonfaulty.}
       Wait for data message from aneib(curr, d_i);
       Receive data message from aneib(curr, d_i);
       Merge received data and curr's local data;
      endif;

  endcase;
end.
```

Fig. 13. Algorithm FTTC.

If the current node has no difficulty in sending data messages to its passive neighbor, it can help its active neighbors. It checks if any active neighbor needs help and the link to the active neighbor needing help is non-faulty. If this is the case, then the current node waits for one partition of data messages from the active neighbor needing help. When the messages are available, the current node merges them with its own data. Finally, the current node sends the results to its passive neighbor.

An active node with no difficulty in sending messages to its passive neighbor and no active neighbor to help simply sends its own data to its passive neighbor. If there is no link failure, all fault-tolerant operations are skipped and Algorithm FTTC behaves the same as the non-fault-tolerant tree communication scheme, i.e., Algorithm TC.

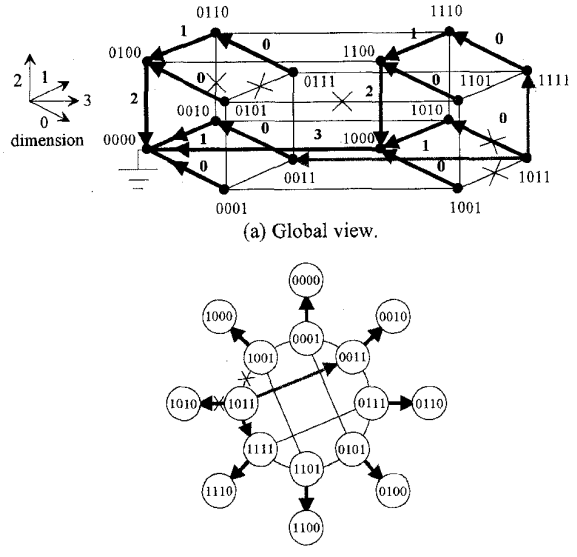
What a passive node does is simpler. It just checks if the link to its active neighbor is faulty. If this link is faulty, it does nothing at this stage, otherwise it receives the data from its active neighbor and merges the received data and its own data.

EXAMPLE 3. In Fig. 14a, the communication tree has one faulty link between nodes 1011 and 1010. Since active node 1011 cannot send messages to its passive node 1010 at stage 0, the active neighbors of node 1011—nodes 1001, 1111, and 0011—must help it. But the link between nodes 1011 and 1001 is also faulty. The number of partitions m is 2, so node 1011 divides the data into 2 partitions and sends them to nodes 1111 and 0011. Each of the two nodes gets one partition. Active nodes 1111 and 0011 receive the data messages from node 1011 and merge the received data and their own data. As for the passive node 1010, it does nothing at this stage. Fig. 14b shows the data flow at stage 0.

THEOREM 5. *Algorithm FTTC takes $O(n + k)$ parallel steps, where k is the number of faulty stages and $k < n$. The communication slowdown is less than* 2.

PROOF. There are n stages in a communication tree. Based on our fault-tolerance strategy, when a certain active node has difficulty in sending data messages to its passive neighbor, it tries to send the messages to all its active neighbors. This introduces an additional parallel step to a stage. Besides, all the adjacent link of the sink (the only passive node at stage $n - 1$) should be nonfaulty. So at stage $n - 1$ the only active node must be able to send data message to the sink. Consequently, $O(n + k)$ parallel steps are required, where k is the number of faulty stages and $0 \le k < n$. The value of k depends on fault patterns. The communication slowdown is $(n + k)/n < (n + n)/n = 2$. □

The traditional rerouting strategy is: when a node has difficulty in sending messages to another node via the direct link between them, it tries a detour path of three hops. If this strategy is adopted for the tree communication scheme, three parallel steps will be introduced to a stage. Therefore, $O(n + 3k)$



(a) Global view.

(b) Data flow at stage 0.

Fig. 14. An example for Algorithm FTTC.

parallel steps are required, where $0 \le k \le n$. And the communication slowdown is $(n + 3k)/n \le (n + 3n)/n = 4$.

THEOREM 6. *Algorithm FTTC is deadlock-free.*

PROOF. Reversing the directions of the arrows of a fault-free communication tree, we get the Resource Dependency Graph (RDG) [5] of the nodes. Since this graph is just a tree, it does not have any cycle. This means the normal tree communication scheme has no deadlock problem. If a certain link between two nodes in the tree is faulty, the corresponding edge of the RDG must be removed. According to the strategy of Algorithm FTTC, m edges are added to the RDG. This operation also does not introduce any cycle in the RDG. Hence Algorithm FTTC does not generate deadlocks. Fig. 15 shows the RDG of Example 3. □

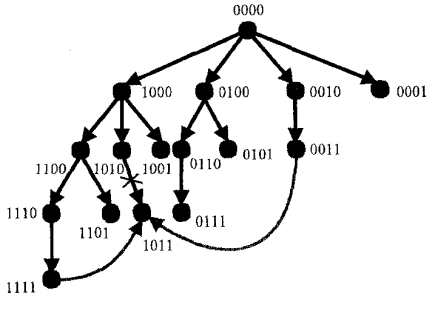


Fig. 15. The RDG of Example 3.

All the nodes communicate in an asynchronous fashion; they wait for data until data is available. Consequently we have no synchronization problem in the fault-tolerant tree communication scheme.

An interesting issue about our approach is how many link failures it can tolerate. By Theorem 3, we know that Algorithm FIND_TREE is guaranteed to generate a communication tree if the number of faulty link is smaller than 2^{n-1}. However, Algorithm FIND_TREE succeeds if there exists at least one node with all adjacent links non-faulty even when the number of faulty links exceeds 2^{n-1}. Furthermore, Algorithm FTTC always routes messages successfully unless there are nodes that are isolated due to faulty links. A node v is said to be *isolated* if $FAULT(v) = 1^n$. When isolated nodes exist, data for computation must not be distributed to these nodes. Instead, the data is uniformly distributed to all the other non-isolated nodes. Thus, combined with the UDD (Uniform Data Distribution) strategy, our approach can tolerate 2^{n-1} link failures. And we have the following theorem.

THEOREM 7. *The combination of Algorithm FIND_TREE, Algorithm FTTC, and the UDD strategy can tolerate at least 2^{n-1} link failures.*

3.3 Node Failures

Although in the previous discussions we assume that there are only link failures, node failures can also be tolerated. By employing hardware self-test, a faulty node can be easily detected, and this information is available to all its neighbors. A faulty node can be treated as a node with all its adjacent links being faulty, i.e., an isolated node. Thus, if node u has a faulty neighbor v, the word $FAULT$ kept by node u for node v is composed of all 1s, i.e., $FAULT(v) = 1^n$. The data to be manipulated is distributed uniformly to all nonfaulty nodes. Then, Algorithms FIND_TREE and FTTC see no difference between faulty nodes and isolated nodes , and they behave in the same way as if only link failures exist. Fig. 16 illustrates an example communication tree in the presence of link failures and node failures.

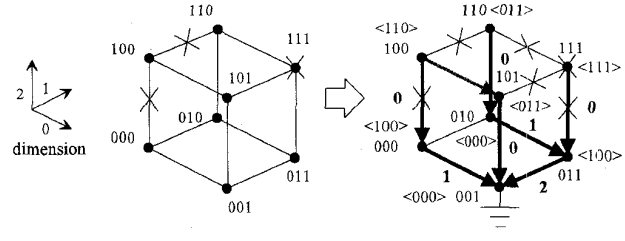


Fig. 16. A communication tree with both link and node failures.

Because of the UDD strategy, data is uniformly distributed. It is obvious that the computation slowdown is $N/(N - f)$, where f is the number of faulty nodes.

EXAMPLE 4. In Fig. 16, links 100-110 and 100-000 and node 111 are faulty. Since node 111 is faulty, all links around node 111 are treated as faulty. At stage 0, node 100 is not able to send messages to node 000, thus it sends them to node 101. The data to be computed is uniformly distributed to all nonfaulty nodes, therefore node 111 does not have to and cannot do anything. Although we have two link failures and one node failure in a 3D hypercube, the computation is still completed correctly.

4 Conclusions

In this paper, the tree communication scheme is explored extensively. We have studied the generalization of the tree communication scheme and described in detail how it can be utilized for practical applications. In addition, fault-tolerant algorithms based on the tree communication scheme for hypercube multiprocessors are presented. These algorithms are designed to detect link failures in a hypercube computer, to find a good communication tree in the presence of link failures, and to reroute messages in runtime. Moreover, these algorithms can also be applied even if node failures exist. We apply Uniform Data Distribution method upon node failures. Thus the faulty nodes in a communication tree will not make the tree communication scheme invalid.

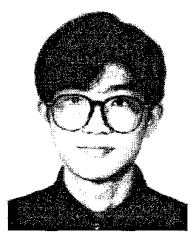
With our approach, a large number (at least 2^{n-1}) of link failures can be tolerated. The computation slowdown is small if the number of faulty nodes is small. The communication slowdown is also small (< 2) because the proposed approach introduces at most $n - 1$ parallel steps to a tree communication session. However, the approach to achieving fault tolerance by emulating the entire hypercube with the residual hypercube [12], [13] would incur a computation and communication slowdown of 2. With the incorporation of fault tolerance into the tree communication scheme, parallel algorithms utilizing the tree communication scheme function correctly under multiple nodes and/or link failures. Therefore, the presented two-level fault-tolerance approach can enhance the reliability considerably with little performance degradation both in computation and communication.

Acknowledgment

This research was supported by the National Science Council, Taiwan, R. O. C., under grant NSC 84-0408-E002-008.

References

[1] A. Avizienis, "Fault-Tolerance: The Survival Attribute of Digital Systems," *Proc. IEEE*, vol. 66, pp. 1,109-1,125, Oct. 1978.
[2] Y. Saad and M.H. Schultz, "Topological Properties of Hypercube," *IEEE Trans. Computers*, vol. 37, no. 7, pp. 867-872, July 1988.
[3] A.C. Elster and A.P. Reeves, "Block-Matrix Operations Using Orthogonal Trees," *Proc. SIAM Third Int'l Conf. Hypercube Multiprocessors*, pp. 1,554-1,561, Pasadena, Calif., Jan. 1988.
[4] A.C. Elster, M.U. Uyar, and A.P. Reeves, "Fault-Tolerant Matrix Operations on Hypercube Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, pp. III 169-177, Aug. 1989.
[5] T.C. Lee and J.P. Hayes, "A Fault-Tolerant Communication Scheme for Hypercube Computers," *IEEE Trans. Computers*, vol. 41, no. 10, pp. 1,242-1,256, Oct. 1992.
[6] T.C. Lee and J.P. Hayes, "Routing and Broadcasting in Faulty Hypercube Computers," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, vol. I, pp. 346-354, Jan. 1988.
[7] S. Park and B. Bose, "Broadcasting in Hypercubes with Link/Node Failures," *Proc. Fourth Symp. Frontiers of Massively Parallel Computation*, pp. 286-290, 1992.
[8] C.S. Raghavendra et al., "Free Dimension—An Effective Approach to Achieving Fault Tolerance in Hypercubes," *Proc. IEEE 22nd Int'l Symp. Fault-Tolerant Computing*, pp. 170-177, July 1992.
[9] B.S. Chlebus, K. Diks, and A. Pelc, "Optimal Broadcasting in Fault Hypercubes," *Proc. IEEE 21st Int'l Symp. Fault-Tolerant Computing*, pp. 266-273, June 1991.
[10] P. Ramanathan and K.G. Shin, "Reliable Broadcast in Hypercube Multiprocessors," *IEEE Trans. Computers*, vol. 37, no. 12, pp. 1,654-1,657, Dec. 1988.
[11] J. Salmon, "Binary Gray Codes and the Mapping of Physical Lattice into a Hypercube," *Caltech Concurrent Processor (ccp)* Hm-51, 1983.
[12] J. Bruck, R. Cypher, and D. Soroker, "Running Algorithms Efficiently on Faulty Hypercubes," *Computer Architecture News*, vol. 19, no. 1, pp. 89-96, Mar. 1991.
[13] J. Bruck, R. Cypher, and D. Soroker, "Tolerating Faults in Hypercubes Using Subcube Partitioning," IBM Technical Report, RJ 8142 (74555), May 1991.
[14] H. Sulivan, T. Bashkow, and D. Klappholz, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine," *Proc. Fourth Symp. Computer Architecture*, pp. 105-124, Mar. 1977.
[15] L.G. Valiant, "A Scheme for Fast Parallel Communication," *SIAM J. Computers*, vol. 11, no. 2, pp. 350-361, May 1982.
[16] H. Kataeff, "Incomplete Hypercubes," *Hypercube Multiprocessors*, M.T. Heath, ed., pp. 258-264, 1987.
[17] A.L. Decegama, *The Technology of Parallel Processing (Parallel Processing Architectures and VLSI Hardware*, vol. 1) , pp. 441-457. Prentice Hall, 1989.
[18] G.H. Golub and C.F. Van Loan, *Matrix Computations*. Baltimore: John Hopkins, 1983.
[19] S. Latifi, S.Q. Zheng, and N. Bagherzadeh, "Optimal Ring Embedding in Hypercubes with Faulty Links," *Proc. IEEE 22nd Int'l Symp. Fault-Tolerant Computing*, pp. 178-184, July 1992.
[20] F.T. Luk and H. Park, "An Analysis of Algorithm-based Fault Tolerance Techniques," *J. Parallel and Distributed Computing*, vol. 5, pp. 172-184, Apr. 1988.



Yuh-Rong Leu received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1991. He is currently working toward the PhD degree in electrical engineering at the same university. His major research interests are fault-tolerant computing and parallel processing.

Sy-Yen Kuo received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1979, the MS degree in electrical and computer engineering from the University of California at Santa Barbara in 1982, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1987.

Dr. Kuo is a professor in the Department of Electrical Engineering, National Taiwan University. He is currently on leave from National Taiwan University while serving as the chairman of the Institute of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. He was a faculty member in the Department of Electrical and Computer Engineering at the University of Arizona, Tucson, Arizona, from 1988 to 1991, and an engineer at Fairchild Semiconductor and Silvar-Lisco, both in California, from 1982 to 1984. In 1989, he also worked as a summer faculty fellow at the Jet Propulsion Laboratory of the California Institute of Technology. His current research interests include high speed and low power VLSI systems design, fault-tolerant computing, computer architecture and parallel processing, and testing and design for testability of VLSI systems.

Dr. Kuo was a recipient of the Best Paper award in the simulation and test category at the 1986 IEEE/ACM Design Automation Conference (DAC), the National Science Foundation's Research Initiation award in 1989, and the IEEE/ACM Design Automation Scholarship in 1990 and 1991.