JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 20, 707-732 (2004)

Distributed Fault-Tolerant Embedding of Several Topologies in Hypercubes*

SHIH-CHANG WANG, YUH-RONG LEU AND SY-YEN KUO+
Department of Electrical Engineering
National Taiwan University
Taipei, 106 Taiwan
+*E-mail: sykuo@cc.ee.ntu.edu.tw*

Embedding is of great importance in the applications of parallel computing. Every parallel application has its intrinsic communication pattern. The communication pattern graph is mapped onto the topology of multiprocessor structures so that the corresponding application can be executed. To increase the reliability of parallel applications, fault-tolerant embedding is necessary. In this paper, we propose a distributed approach, based on the *faulty link model*, for embedding several topologies into hypercubes with faulty links and/or faulty nodes. The topologies include the ring, the torus, the binomial tree, and a hybrid topology which is a combination of rings and binomial trees. The approach exploits the recursive property of the hypercube, and the proposed algorithms all have of only $O(n)$ parallel steps. Since the distribution of faulty links is arbitrary, an embedded graph with no faulty link may not exist. Therefore, we adopt a 2-phase fault-tolerance strategy to attack this problem. In the first phase, a near-perfect embedding is found, and in the second phase, existing fault-tolerant point-to-point communication schemes are employed. Based on the 2-phase strategy, applications with associated communication pattern graphs with the ring, torus, binomial tree, or hybrid topology can be executed on hypercube multiprocessors with faulty links. For faulty nodes, a technique called UDD (*Uniform Data Distribution*) is proposed. Therefore, with the UDD and the proposed algorithms, both faulty links and faulty nodes can be tolerated.

Keywords: fault-tolerant embedding, hypercube, ring, torus, binomial tree

1. INTRODUCTION

The hypercube is a topology that has received much interest from researchers. Due to its superior mathematical properties, there are many interesting research topics related to the hypercube, including communication, task distribution, embedding, parallel processing application, etc. The hypercube is a symmetric structure and has a high degree of connectivity. Therefore, in addition to research on fault-free hypercubes, much research has concentrated on faulty hypercubes. In general, two fault-tolerance strategies widely used in *interconnection networks* (*IN*) can be applied to the hypercube. One strategy is to incorporate additional nodes and/or links in the normal hypercube to make it more resistant against hardware failures. This strategy can only be adopted in the design phase of a hypercube system. The other strategy is to design application algorithms that can work

Received April 29, 2002; revised June 23, 2003; accepted August 14, 2003.
Communicated by Gen-Huey Chen.
* This research was supported by the National Science Council, Taiwan, under grant NSC 90-2213-E-002-113.

on a faulty hypercube. The main disadvantage of this strategy is the performance degradation incurred by fault-tolerant application algorithms. Since there is no fault-tolerance hardware in most commercial hypercube systems, the latter strategy seems to be more practical than the former.

Due to the high level of connectivity of a hypercube, many topologies can be embedded into it, e.g., chains, rings, meshes, toruses, binomial trees, binary trees, etc. The embedding of the above topologies in fault-free hypercubes has been studied extensively. Fault-tolerant embedding in hypercubes with faulty nodes also has been widely researched extensively [1-6]. However, embedding in hypercubes with faulty links has not been explored completely [7-9]. In [8], we proposed distributed algorithms for ring embedding and reconfiguration in hypercubes with faulty links, and in [9], a sequential algorithm of complexity $O(2^n)$ for binomial-tree embedding was proposed. In this paper, we further generalize the distributed approach in [8] to deal with embedding of the ring, the multi-dimensional torus, the binomial tree, and a hybrid topology consisting of rings and binomial trees in hypercubes with faulty links and/or faulty nodes. We assume first that only faulty links exist in the hypercube. All algorithms are devised based on this assumption. Then, the algorithms as well as a technique called UDD (*Uniform Data Distribution*) can be utilized to deal with faulty nodes. The details will be given in section 7.

Since the distribution of faulty links is random, an embedded graph without any faulty link may not always exist. Therefore, we adopt a 2-phase fault-tolerance strategy. In the first phase, a near-perfect embedding is found, and in the second phase, existing fault-tolerant point-to-point communication schemes are employed. Based on the 2-phase strategy, applications with communication pattern graphs of the ring, torus, binomial tree, or hybrid topology can be executed on hypercube multiprocessors with faulty links.

Our approach is based on the recursion property of the hypercube structure, and it is bottom-up. An embedded topology in a hypercube is formed by recursively merging sub-topologies embedded in subcubes. The basic building blocks are rings and binomial trees. Embedded subrings and sub-binomial trees are then recursively merged into a multi-dimensional torus, a large ring, a large binomial tree, or a hybrid torus-binomial-tree topology. The proposed algorithm has $O(n)$ parallel steps, where n is the dimension of the hypercube.

The organization of this paper is as follows. In section 2, definitions of the topologies are given. Notations and definitions of terms are also provided. In section 3, we discuss the fault-tolerant ring and torus embedding approach. In section 4, fault-tolerant binomial embedding is presented. The generalized algorithm that can handle the embedding of several topologies is described in section 5. In section 6, we compare our works with several previous research results. Section 7 presents how messages are rerouted in the second fault-tolerance phase. Additionally, the strategy for handling faulty nodes is provided in this section. A simulation of ring embedding is presented in section 8. Finally, conclusions are drawn in section 9.

2. PRELIMINARIES

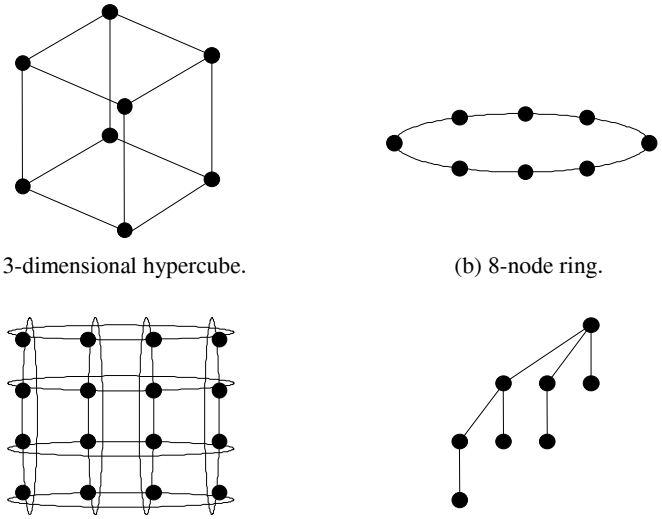
An n-dimensional hypercube denoted as Q_n has 2^n nodes and $n2^{n-1}$ links. Each node is labeled with an n-bit binary number such that the binary labels of two adjacent nodes

differ in only one bit position. The bit positions are numbered from 0 to $n - 1$ with the least significant bit as bit 0. If the labels of two nodes differ only in bit i, the two nodes are said to be connected by an *i-link*, i.e., a link on dimension i. It is easy to see that there are 2^{n-1} links on each dimension. Cutting all links on a dimension, a Q_n can be partitioned into two subcubes, each with 2^{n-1} nodes. A subcube is labeled with an n-bit ternary string. Each bit in the string is one of {0, 1, x}, where x stands for *don't care*. For instance, subcube 0xx1 is composed of 4 nodes – 0001, 0011, 0101, and 0111. An m-dimensional subcube is called an m-subcube. Note that a similar concept can be found in [17].

An N-node ring has N nodes and N links. Each node has two links to adjacent nodes. Each node is given an identification between 0 and $N - 1$. Node i and node $(i + 1 \bmod N)$ are adjacent, where $0 \le i \le N - 1$.

A d-dimensional N-node torus has N nodes and dN links. Each node has $2d$ links to adjacent nodes, 2 links on each dimension. Each node is identified as a d-tuple $(t_0, t_1, \ldots, t_{d-1})$, where $0 \le t_i \le N_i - 1$, $N = \Pi_i(N_i)$, and $0 \le i \le d - 1$. Node $(t_0, t_1, \ldots, t_i, \ldots, t_{d-1})$ is adjacent to nodes $(t_0, t_1, \ldots, t_{i+1 \bmod N_i}, \ldots, t_{d-1})$ and $(t_0, t_1, \ldots, t_{i-1 \bmod N_i}, \ldots, t_{d-1})$.

An n-level binomial tree denoted as BT_n has 2^n nodes and $2^n - 1$ links. It 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. Fig. 1 shows examples of the above topologies.



(a) 3-dimensional hypercube. (b) 8-node ring.

(c) 2-dimensional torus. (d) 3-level binomial tree.

Fig. 1. Example topologies.

One *divide-and-conque* approach to solving the routing problem in Q_n is to apply the concept of supernode partitioning. A *supernode* is a set of nodes and links. Note that a supernode is identical to a subcube. The nodes and links inside a supernode can be connected as any topology. A *superlink* connecting two supernodes is a set of links which connect all the nodes in one supernode and all the nodes in another. Two super-

nodes that can be connected with a superlink must be of the same topology. That is, the numbers of nodes in the supernodes are identical, and so are the numbers of links in the supernodes. Moreover, the topologies of the two supernodes are the same. In fact, they must also be a "*mirror image*" of each other in our approach. The details will be given in section 3.

A complex topology can be constructed based on the supernode and superlink concept. For example, a Q_n can be constructed by connecting two supernodes, each of which is a Q_{n-1}, with a superlink. In other words, we say that a Q_n can be *partitioned* into two Q_{n-1}'s. A fairly complicated hybrid topology can be constructed in this way. For instance, the supernodes can be connected to form a 3-dimensional torus, while the internal topology of each supernode is a binomial tree. A 2-dimensional torus is formed by connecting the supernodes, each of which is a ring topology, into a ring. On the other hand, we say that a 2-dimensional torus can be *collapsed* into a ring, and the ring can be further collapsed into a single supernode.

A level-l superlink is composed of level-(l − 1) superlinks. And a level-0 superlink is a physical link. A level-l supernode is composed of level-(l − 1) supernodes and level-(l − 1) superlinks. And a level-0 supernode is a physical node. Level-l collapsing is the operation that collapses a level-(l − 1) topology consisting of level-(l − 1) supernodes and superlinks into a level-l topology consisting of level-l supernodes and superlinks. Level-0 collapsing is a null operation, and the level-0 topology is the original topology. A topology that can be collapsed l times into a level-l supernode is said to be an l-level topology. Fig. 2 shows an example of collapsing. It is easy to see that a 2-dimensional torus is a 2-level topology.

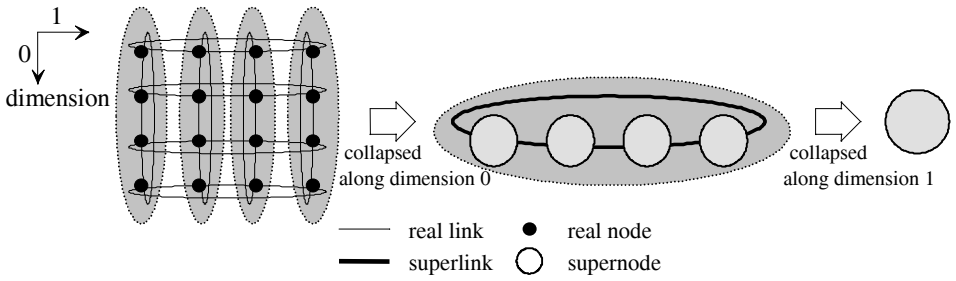


Fig. 2. An example of collapsing.

Recall that we are studying the embedding of several topologies into faulty hypercubes. In order not to confuse the dimension of a hypercube with the dimension of a torus, the dimension of a torus is called the *direction* of a torus. Thus, for example, a 3-dimensional torus will be referred to as a 3-directional torus in the rest of this paper. Similarly, the links of a ring, a torus, a binomial tree, or a hybrid topology are referred as *connections*.

A fault-free hypercube (subcube) is also called a healthy hypercube (subcube). A dimension i is said to be *injured* if one or more i-links are faulty. The *injury degree* of a dimension is defined as the number of faulty links on that dimension. The dimensions

can then be ranked according to their injury degrees. If all the links on a certain dimension are fault-free, this dimension is said to be *free*. A free dimension is the least injured.

The following notations will be used in the rest of this paper:
b_i: bit i of word b;
I_i: an n-bit word with bit $i = 1$ and other bits = 0;
0^k: a k-bit word with all bits = 0;
1^k: a k-bit word with all bits = 1;
x^k: a k-bit word with all bits = don't care;
pq: a word formed by concatenating two words p and q;
f_g: the injury degree of dimension g;
curr: the label of the current node, i.e., the node executing the algorithm;
neib(d): the label of a neighbor of the current node along dimension d;
u-v: a connection between nodes u and v;
$H(a, b)$: the Hamming distance between two words a and b, i.e., $H(a, b) = \Sigma_j c_j = \Sigma_j (a \oplus b)_j$.

3. RING AND TORUS EMBEDDING

In this section, we discuss the basic ring embedding approach and its extension to torus embedding. Our method for extending the ring embedding approach involves the key concepts of the generalized embedding method that will be discussed in section 5.

3.1 Ring Embedding

The basic idea of the ring-embedding algorithm is similar to that in [8]. However, we exploit the concept of a *cost function* (described below) to make the idea more general, and as a result, our presentations of our algorithms in this paper will be more concise. We assume that each node has the locations of all the faulty links. This assumption is reasonable because an optimal broadcasting algorithm which takes $n + 1$ steps in an injured hypercube with faulty links was developed in [10]. Therefore, it is possible for nodes in an injured hypercube with faulty links to broadcast the fault information to other nodes. Thus, each node can make a decision based on the information in its own memory instead of performing communications with other nodes. Note that each node obtains the fault information from the received broadcasting-messages, and that the distributed embedding algorithm shown in Fig. 6 is performed using such information in each node.

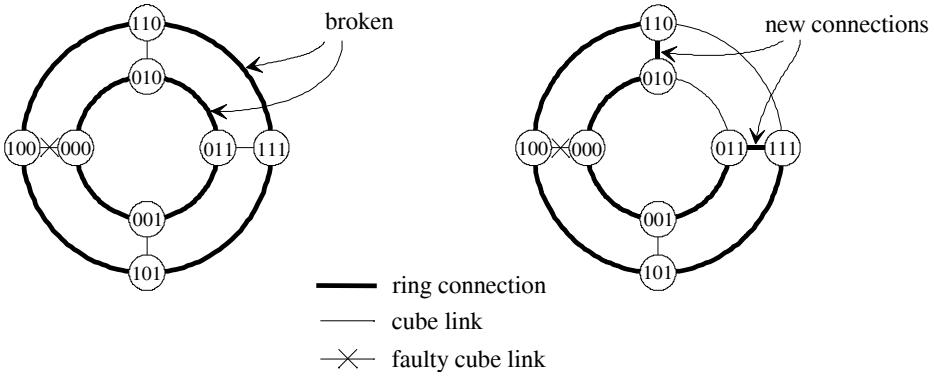
If faulty links are not allowed to exist in the embedded ring, the number of injured dimensions of the hypercube must be at most $n - 2$. That is, at least two free dimensions exist. If the number of injured dimensions exceeds $n - 2$, the number of faulty links must be greater than $n - 2$, and a fault-free ring can not exist if more than $n - 2$ faulty links are incident on one of the nodes. Therefore, a Q_n can be partitioned along the two free dimensions into 2^{n-2} healthy 2-subcubes. A 2-subcube is also a 4-node ring. A pair of 4-node rings can be merged into an 8-node ring embedded in a 3-subcube. The merging process continues recursively until a 2^n-node ring embedded in Q_n is formed. Since we adopt a 2-phase strategy, an embedded graph with faulty links is acceptable because other fault-tolerant point-to-point communication schemes can be utilized to tolerate these faulty links in the second phase. Consequently, no assumption about the number of faulty links in the hypercube or the number of injured dimensions is necessary.

The dimensions are sorted according to the injury degrees and then stored in a stack called D. The lower the injury degree a dimension has, the higher the position in stack D in which the dimension resides. Stack D is popped to provide the dimension to be processed in each merging step. The ring-embedding algorithm has n steps numbered from 0 to $n - 1$. Subrings are merged dimension by dimension. The dimension processed in step i is denoted as g_i, and $f_{g_i} \leq f_{g_j}$ $(0 \leq i < j \leq n - 1)$. In step 0, all nodes are merged across dimension g_0, and 2^{n-1} 1-subcubes are formed. In step 1, two 1-subcubes are merged across dimension g_1, and 2^{n-2} 2-subcubes are formed. The general case is as follows in step i, 2^{n-i} 2^i-node subrings are merged across dimension g_i into 2^{n-i-1} 2^{i+1}-node subrings. Two subrings that can be merged form a *merging group*.

Four nodes are involved in merging two subrings in each step i. Let nodes a and b be in one subring, and let c and d be in another. The relationships among them are:

1. $H(a, b) = 1$; $H(c, d) = 1$.
2. $(a \oplus b) = (c \oplus d)$.
3. $(a \oplus c) = (b \oplus d) = I_{g_i}$.

Merging the two subrings means breaking connections a-b and c-d, and establishing connections a-c and b-d. Fig. 3 shows a merging example for step 2. In Fig. 3, a = 110, b = 111, c = 010, and d = 011.



(a) Before merging. (b) After merging.

Fig. 3. Merging two subrings in the presence of faulty links.

Before proceeding with further discussion, we introduce the concept of *pseudo-faulty* links. Pseudo-faulty links are introduced to maintain uniformity of the structures of subrings formed in step i. That is, all node labels of a formed subring must be identical to those of another subring in bit positions g_0 through g_i. This property must be maintained so that we can guarantee that subrings can be recursively merged. If a certain g_i-link is faulty, the corresponding g_i-links in all other merging groups are treated as being pseudo-faulty. That is, a faulty link makes an "image" in each merging group.

Since each node has the locations of all the faulty links, the pseudo faulty links can be identified without using internode communications. The ring-embedding algorithm does not care whether a link is faulty or pseudo-faulty. Fig. 4 illustrates the concept of pseudo-faulty links. Because (000000-000100)-link is faulty, the corresponding g_i-links in all the other merging groups are treated as being pseudo-faulty. Based on the concept of pseudo-faulty links, the operations in each merging group are uniform.

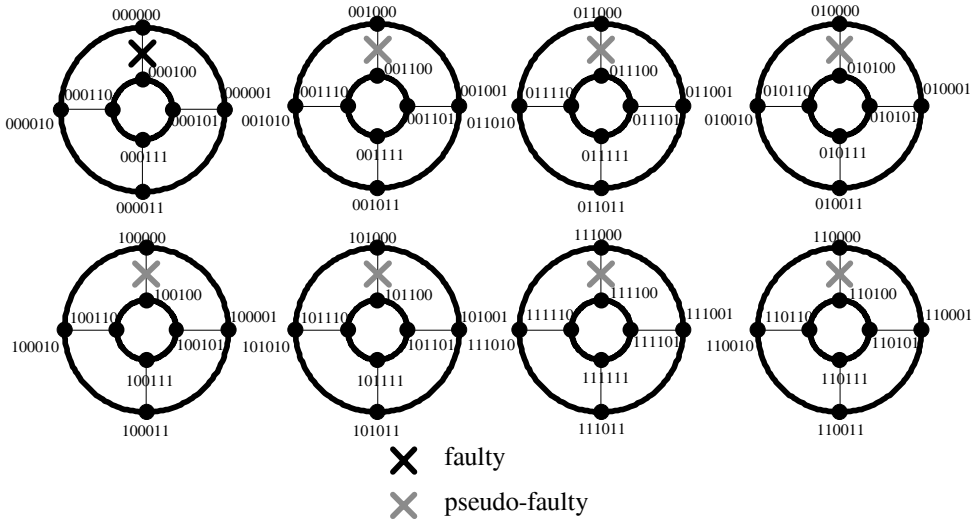


Fig. 4. The concept of pseudo-faulty links for ring embedding.

To obtain a embedded ring without faulty links, the condition for a node Y that can be involved in the merging operations in step i is as follows:

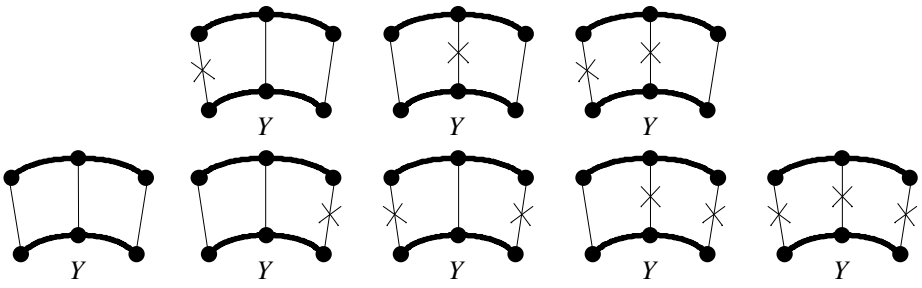
the g_i*-links that are connected to node Y and at least one of its ring neighbors are non-faulty or non-pseudo-faulty.*

That is, node Y has to check 3 g_i-links. Note that two g_i-links are required for the merging operations in step i because they are used as ring connections in the merged ring. Let $f(Y) = 1$ if the g_i-link of node Y is faulty or pseudo-faulty; otherwise $f(Y) = 0$. The two ring neighbors of node Y are denoted as Y_1 and Y_2. Then, we define a cost function according to the above condition:

$$COST_R(Y) = f(Y) + \frac{1}{2} f(Y_1) + \frac{1}{2} f(Y_2).$$

The range of $COST_R$ is from 0 to 2. A certain node satisfying the condition has $COST_R = 0$ or 0.5, and it can be selected to initiate the merging operations. Nevertheless, a node with $COST_R \le 0.5$ may not exist due to the distribution of faulty links. Thus, we

choose the node with minimal $COST_R$ to initiate the merging operations. If there are still several remaining choices, the node with the minimal node label is selected. Obviously, if the selected node has $COST_R > 0.5$, faulty links will exist in the embedded ring. Fig. 5 shows the possible values of $COST_R$.



(a) 0. (b) 0.5. (c) 1. (d) 1.5. (e) 2.

Fig. 5. Possible values of $COST_R$.

The merging operations include establishing and breaking connections. To establish a connection, a node simply adds a record in its memory for identifying who is its neighbor in the ring. To break a connection, the corresponding connection record is deleted from the memory of a node. For example, in Fig. 3, node 011 (111) has minimal $COST_R$ (= 0) and it has to initiate the merging operations. It sends a break-ring-connection message to node 010 (110) to inform this node to delete the connection 011-010 (111-110). Finally, connections 010-110 and 011-111 are established.

Recall that in step i ($2 \le i \le n - 1$), 2^{n-i} 2^i-node subrings are merged into 2^{n-i-1} 2^{i+1}-node subrings. Furthermore, two g_i-links are required to merge two subrings. Consequently, a total of 2^{n-i} g_i-links are essential in step i. The number of g_i-links required in step i decreases as the value i increases. So it is desirable that there be fewer faulty links in dimension g_i with smaller i. This is the reason why the top of stack D is the dimension with the smallest number of faulty links. If the dimensions are not sorted according to the number of faulty links, then the node chosen to initiate the merging operations in each step is likely to have $COST_R \ge 0.5$ and, thus, the probability that faulty links exist in the embedded ring is high.

Two nodes are involved in the merging operations in a subring. The node that initiates the operations is said to be *active*; the other is said to be *passive*. Nodes which are neither active nor passive are said to be *idle*. The active node sends a break-ring-connection message to the passive node. The passive node receives this message and then breaks one of its ring connections. The connection record simply stores the dimension to which the connection belongs. Each node always keeps two ring connection records in every step except step 0. The ring-embedding algorithm is shown in Fig. 6.

In Algorithm Ring, Procedure Merge_Ring is executed in order to merge rings. What a node does in a step depends on its status in that step. Procedure Merge_Ring is shown in Fig. 7. For example, there is a 3-cube with a faulty link 000-100. From Algorithm Ring, it is easy to see that Stack D is composed of 0, 1, and 2 from top to bottom.

```
Algorithm Ring;
{Each node keeps two ring connection records. Stack D stores the sorted dimensions.}

begin
    for i = 0 to n − 1 do begin
      g ← pop (D);
      if curr is the minimal node label among the nodes with minimal COST_R
        then status ← ACTIVE;
      else begin
            wait for a break-ring-connection message;
            if a break-ring-connection message is received
              then status ← PASSIVE;
            else status ← IDLE;
      endelse;
      Merge_Ring(status, g);
    endfor;
end.
```

Fig. 6. Algorithm ring.

```
Procedure Merge_Ring(status, g)
{The two ring connections are on dimensions d1 and d2, respectively}
{Subrings are merged across dimension g.}

begin
    case status of

    ACTIVE:
    {Node curr is used to initiate the merging operations.}
      begin
        case of
          COST_R(neib(d1)) = COST_R(neib(d2)): r ← min(d1, d2);
          COST_R(neib(d1)) < COST_R(neib(d2)): r ← d1;
          COST_R(neib(d1)) > COST_R(neib(d2)): r ← d2;
        endcase;
        Send a break-ring-connection message to node neib(r) via link r;
        Break connection curr-neib(r);
        Establish connection curr-neib(g);
      end;

    PASSIVE:
    {Node curr receives a break-ring-connection message via link l.}
      begin
        Break connection curr-neib(l);
        Establish connection curr-neib(g);
      end;

    IDLE:
    {Node curr does nothing.}

    endcase;
end.
```

Fig. 7. Procedure Merge_Ring.

In the $i = 0$ step, four subrings, (000-001), (010-011), (100-101), and (110-111) along dimension 0 are established. Moreover, in the $i = 1$ step, two subrings, (000-001-011-010) and (100-101-111-110), along dimension 1 are established. In the $i = 2$ step, the Active nodes are 011 and 111. Clearly, those two ring connections are on dimensions $d_1 = 0$ and $d_2 = 1$. From Procedure Merge_Ring, r is $\min(d_1, d_2) = 0$. Nodes 011 and 111 break connections 011-010 and 111-110 (see Fig. 3 (a)). Finally, two connections 011-111 and 110-010, are established (see Fig. 3 (b)).

Theorem 1 The complexity of Algorithm Ring is $O(n)$ parallel steps.

Proof: The proof proceeds by induction on n. It is obviously true for $n = 2$. Assume that it is also true for $n = m - 1$. Considering $n = m$, partition $Q_n = Q_m$ on dimension $m - 1$ into 2 Q_{m-1}'s, i.e., $1x^{m-1}$ and $0x^{m-1}$. By the assumption, Algorithm Ring can find a ring in each Q_{m-1}, and the node label sequences of both rings are the same except for bit $m - 1$. Because Algorithm Ring_A is capable of merging subrings, these two 2^{m-1}-node ring can be merged into a 2^m-node ring in iteration $m - 1$. In addition, according to the operations in Procedure Merge_Ring, one parallel step (an inter-node communication step) is required to merge two subrings. Therefore, $(m - 1) + 1 = m$ parallel steps are needed. As a result, by the principle of mathematical induction, the theorem is proved. ❑

3.2 Torus Embedding

The ring-embedding approach can be further extended to the embedding of tori in faulty hypercubes. The torus-embedding algorithm utilizes the concept of collapsing. If we collapse a d-directional torus along a certain direction, it becomes a $(d - 1)$-directional torus. That is, all the rings of the torus on this direction become level-1 supernodes (see Fig. 2). To embed the torus in a hypercube, based on the collapsing operation, rings are embedded on the corresponding direction. The collapsing operations can continue until a level-d supernode is formed. For instance, in Fig. 2, 4 rings are embedded on direction 0 corresponding to level-1 collapsing, and 4 rings are embedded on direction 1 corresponding to level-2 collapsing. The concept of collapsing will also be exploited in section 5 to embed hybrid topologies in hypercubes.

Assume that $n = m_0 + m_1 + \ldots + m_{d-1}$, where d is the direction of a torus to be embedded, and that $m_i \geq m_j$ $(0 \leq i < j \leq d - 1)$. A Q_n can embed a $2^{m_0} \times 2^{m_1} \times \ldots 2^{m_{d-1}}$ d-directional torus. For example, if $n = 5$, $d = 2$, $m_0 = 3$, and $m_1 = 2$, then a Q_5 can embed an 8×4 2-directional torus.

The dimensions of the hypercube are also sorted based on the injury degrees so that $f_{g_i} \leq f_{g_j}$ $(0 \leq i < j \leq n - 1)$. Then, the sorted dimensions are divided into d groups so that group i has m_i dimensions, where $0 \leq i \leq d - 1$. The dividing strategy is as follows:

$g_0, g_d, g_{2d}, \ldots$ in group 0,
$g_1, g_{d+1}, g_{2d+1}, \ldots$ in group 1,
.
.
.
$g_{d-1}, g_{2d-1}, g_{3d-1}, \ldots$ in group $d - 1$.

The dimensions in group i are still sorted, and they are stored in stack D_i with the top of the stack being the dimension with the lowest injury degree.

In a d-directional torus, each node has $2d$ connections to adjacent nodes, 2 connections on every direction. Thus, each node must keep 2 connection records for each direction. A connection record also simply stores the dimension to which the connection belongs. The torus-embedding algorithm has d iterations. In iteration i, stack D_i is popped to give the dimension to be processed, and Procedure Ring, a modification of Algorithm Ring, is invoked to embed rings on direction i. Since there are m_i dimensions in stack D_i, m_i steps are required. Therefore, the total number of steps is n. The torus-embedding algorithm and the related procedures are shown in Figs. 8, 9, and 10.

Algorithm Torus shown in Fig. 8 is very simple. It simply embeds rings on each direction. Procedure Ring shown in Fig. 9 is similar to Algorithm Ring shown in Fig. 6 except that n is changed to m_i in line 2 and D is changed to D_i in line 3. Furthermore, Procedure Merge_Ring2 instead of Merge_Ring is called. However, Procedure Merge_Ring2 shown in Fig. 10 is modified from Merge_Ring shown in Fig. 7. The merge operations of Procedure Merge_Ring2 only influence the connection records for direction i, while no direction argument is needed in Procedure Merge_Ring.

```
Algorithm Torus;
{The direction of the torus is d.}

begin
    for i = 0 to d − 1 do
        Ring(i);
end.
```

Fig. 8. Algorithm torus.

```
Procedure Ring(i);
{Stack D_i stores sorted dimensions for direction i.}

begin
    for j = 0 to m_i − 1 do begin
      g ← pop (D_i);
      if curr is the minimal node label among the nodes with minimal COST_R
        then status ← ACTIVE;
      else begin
        wait for a break-ring-connection message;
        if a break-ring-connection message is received
          then status ← PASSIVE;
        else status ← IDLE;
      endelse;
      Merge_Ring2(status, g, i);
    endfor;
end.
```

Fig. 9. Procedure ring.

```
Procedure Merge_Ring2(status, g, i)
{The two ring connections for direction i are on dimensions d1 and d2.}
{Subrings are merged across dimension g.}

begin
    case status of

    ACTIVE:
    {Node curr is used to initiate the merging operations.}
      begin
        case of
          COST_R(neib(d1)) = COST_R(neib(d2)): r ← min(d1, d2);
          COST_R(neib(d1)) < COST_R(neib(d2)): r ← d1;
          COST_R(neib(d1)) > COST_R(neib(d2)): r ← d2;
        endcase;
        Send a break-ring-connection message to node neib(r) via link r;
        Break connection curr-neib(r);
        Establish connection curr-neib(g);
      end;

    PASSIVE:
    {Node curr receives a break-ring-connection message via link l.}
      begin
        Break connection curr-neib(l);
        Establish connection curr-neib(g);
      end;

    IDLE:
    {Node curr does nothing.}

    endcase;
end.
```

Fig. 10. Procedure Merge_Ring2.

Theorem 2 The complexity of Algorithm Torus is O(n) parallel steps.

Proof: According to our strategy, the dimensions are divided into d groups so that group i has m_i dimensions, where $0 \le i \le d - 1$. Corresponding to each group i, Procedure Ring is invoked to embed m_i-node rings, and $O(m_i)$ parallel steps are required (from **Theorem 1**). Therefore, the complexity of Algorithm Torus is

$$O(m_0) + O(m_1) + \ldots + O(m_{d-1})$$
$$= O(m_0 + m_1 + \ldots + m_{d-1})$$
$$= O(n)$$

parallel steps. ❑

4. BINOMIAL TREE EMBEDDING

A previous work on relative binomial tree embedding in a Q_n was given in [19]. A simple embedding algorithm was proposed that can embed an n-level binomial tree in Q_n with up to $n - 1$ faulty links in $\log(n - 1)$ steps, and they extended the result to show that spanning binomial trees exist in a connected Q_n with up to $\frac{3(n-1)}{2} - 1$ faulty links.

The major differences between binomial tree embedding and ring embedding lie in the definition of the cost function and the merging method. Let us first take a look at examples of merging two 2-level binomial trees that are embedded in two 2-subcubes (see Fig. 11).

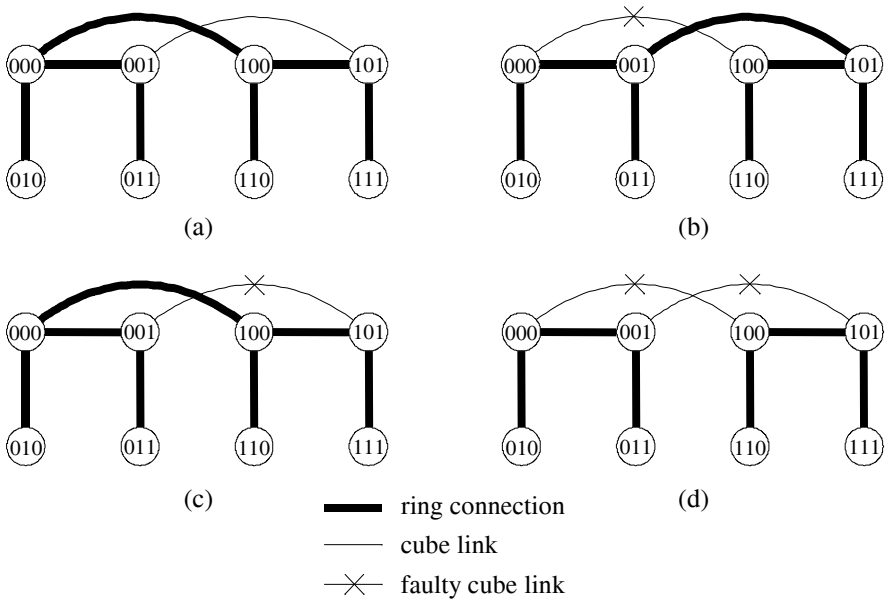


Fig. 11. Examples of merging binomial trees.

In Fig. 11, we illustrate 4 cases of merging, depending on the locations of the faulty links. Based on the definition of binomial trees, a BT_n is constructed by connecting the roots of two BT_{n-1}'s, and it is easy to see that the number of choices is 2. For example, in case (a), nodes 000 and 001 are the candidate roots of the left binomial tree while nodes 100 and 101 are the candidate roots of the right binomial tree. We can connect either 000-100 or 001-101 to form a BT_3. The candidate root with the smaller node label in each BT_2 is selected to be connected. Therefore, we connect 000-100. In case (b), link 000-100 is faulty, so 001-101 is the only choice. Similarly, in case (c), 000-100 is the only choice. In case (d), links 000-100 and 001-101 are both faulty; thus, either choice will result in a BT_3 with a connection on a faulty link. No matter what n is, the number of choices for merging is always 2 with this method of embedding binomial trees. Fig. 12 shows this limitation.

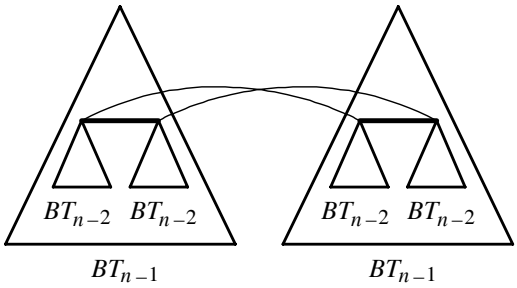


Fig. 12. Merging two binomial trees into one.

Since only candidate roots of binomial trees can be connected for merging, it is important for a node to check whether it is a candidate root or not. The root of a BT_n has n connections to other nodes, so if a node has n connections, it can be a candidate root. Note that there are two candidate roots in a BT_n.

The concept of pseudo-faulty links is also applied in binomial tree embedding to maintain the uniformity of the structure of the embedded sub-binomial trees. Fig. 13 shows how the concept of pseudo-faulty links is employed in binomial tree embedding.

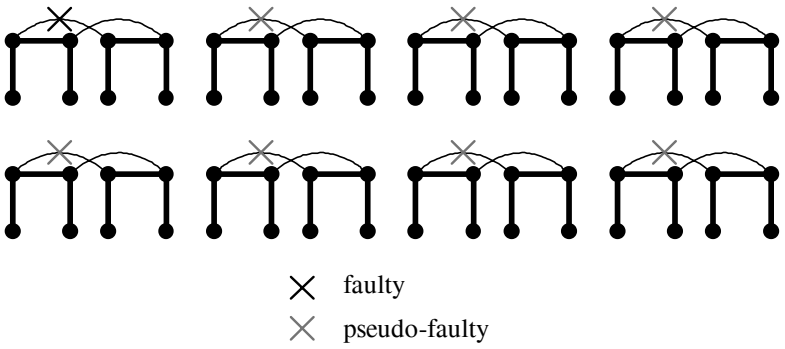


Fig. 13. The concept of pseudo-faulty links applied to binomial tree embedding.

The dimensions of the hypercube are sorted based on the injury degree and then stored in a stack D. The binomial-tree-embedding algorithm also has n steps, and in step i, BT_i's are merged across dimension g_i into BT_{i+1}'s. If faulty links are not allowed to exist in the embedded binomial tree, the necessary condition for a node Y to be involved in the merging operations in step i is as follows:

The number of binomial-tree connections of node Y *is* i, *and the* g_i*-link of node* Y *is non-faulty or non-pseudo-faulty.*

Let $f(Y) = 1$ if the g_i-link of node Y is faulty or pseudo-faulty; otherwise, $f(Y) = 0$. Next, we define a cost function:

$COST_{B-T}(Y) = f(Y)$.

The value of $COST_{B-T}$ is either 0 or 1. The candidate root node with the smaller $COST_{B-T}$ is chosen to be connected. If the $COST_{B-T}$ values of both candidate root nodes are equal, the one with the smaller node label is chosen. The binomial-tree-embedding algorithm and the related procedure are shown in Fig. 14.

```
Algorithm B-Tree;
{Each node keeps binomial-tree connection records. Stack D stores the sorted dimensions.}

begin
    for i = 0 to n − 1 do begin
      g ← pop (D);
      if node curr is a candidate root then
        case of
          COST_B-T(curr) = COST_B-T(another candidate root):
            if curr < the label of another candidate root then Merge_B-Tree(g);
          COST_B-T(curr) < COST_B-T(another candidate root): Merge_B-Tree(g);
          COST_B-T(curr) > COST_B-T(another candidate root): {Node curr does nothing.};
        endcase;
    endfor;
end.

Procedure Merge_B-Tree(g)

begin
    Establish connection curr-neib(g);
end.
```

Fig. 14. Algorithm B-Tree.

Theorem 3 The complexity of Algorithm B-Tree is $O(n)$.

Proof: Because no internode communication is necessary for Algorithm B-Tree, it is a sequential algorithm with n iterations. Therefore, the complexity is, obviously, $O(n)$. ❑

5. HYBRID TOPOLOGY EMBEDDING

The recursive merging method can be generalized for the embedding of hybrid topologies that are combinations of rings and binomial trees. In section 2, the concepts of supernodes and superlinks were introduced, and we also described how these concepts can be exploited to construct hybrid topologies. In section 3.2, we presented the torus-embedding algorithm based on the ring-embedding approach and the collapsing concept. The general embedding algorithm presented in this section is based on Algorithm Torus and Algorithm B-Tree.

Fig. 15 shows a hierarchical view of a 3-level hybrid topology. The internal topology of level-2 and level-1 supernodes is BT_2, and the level-2 supernodes are connected to form a ring. There are 64 nodes in this hybrid topology, and it can be embedded in a Q_6.

The internal topology of a supernode is either a ring or a binomial tree. Therefore, a variable "protocol" is required to keep the information of the merging approach (ring or binomial tree) used for each level in the process of merging operations. Assume that $n = m_1 + m_2 + \ldots + m_l$, where l is the level of the hybrid topology to be embedded and $m_i \geq m_j$ $(1 \leq i < j \leq l)$. A Q_n can embed a $2^{m_1} \times 2^{m_2} \times \ldots 2^{m_l}$ -node hybrid topology. For example, in Fig. 15, $l = 3$, $m_1 = 2$, $m_2 = 2$, and $m_3 = 2$; and a Q_6 can embed this topology.

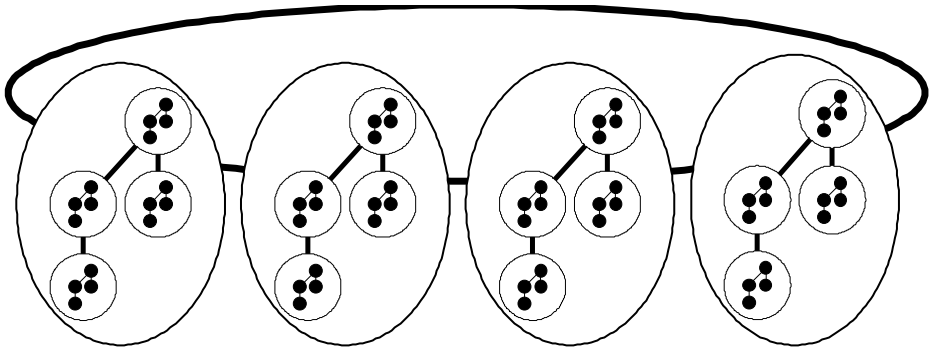


Fig. 15. Hierarchical view of a hybrid topology.

The dimensions of the hypercube are sorted according to the injury degree so that $f_{g_i} \leq f_{g_j}$ $(0 \leq i < j \leq n - 1)$. Then, the sorted dimensions are divided into l groups, and group i has m_i dimensions, where $1 \leq i \leq l$. The dividing strategy is as follows:

$g_0, g_l, g_{2l}, \ldots$ in group 1,
$g_1, g_{l+1}, g_{2l+1}, \ldots$ in group 2,
.
.
.
$g_{l-1}, g_{2l-1}, g_{3l-1}, \ldots$ in group l.

The dimensions in group i are still sorted, and then are stored in stack D_i with the top of the stack being the dimension with the lowest injury degree.

Each node must keep connection records for every level of the hybrid topology. If the protocol of a certain level is "ring", then each node must keep two ring connection records for this level. If the protocol of level i is "binomial tree", then at most m_i binomial-tree connection records are kept in each node. Algorithm General shown in Fig. 16 calls the existing procedures used for ring and binomial tree embedding. Fig. 17 shows Procedure B-Tree called in Algorithm General.

```
Algorithm General;
{The level of the hybrid topology is l.}

begin
    for i = 1 to l do begin
      case protocol_i of
      RING:
        Ring(i);
      BINOMIAL TREE:
        B-Tree(i);
      endcase;
    endfor;
end.
```

Fig. 16. Algorithm general.

```
Procedure B-Tree(i);
{Each node keeps binomial-tree connection records for level i.}

begin
    for j = 0 to m_i − 1 do begin
      g ← pop (D_i);
      if node curr is a candidate root then
        case of
          COST_B−T(curr) = COST_B−T(another candidate root):
            if curr < the label of another candidate root then Merge_B-Tree2(g, i);
          COST_B−T(curr) < COST_B−T(another candidate root): Merge_B-Tree2(g, i);
          COST_B−T(curr) > COST_B−T(another candidate root): {Node curr does nothing.};
      endcase;
    endfor;
end.
```

Fig. 17. Procedure B-Tree.

Procedure Ring called in Algorithm General is identical to the procedure called in Algorithm Torus. However, in Algorithm Torus, stack D_i stores the dimensions for direction i ($0 \le i \le d - 1$) of the torus, but stack D_i stores the dimensions for level i ($1 \le i \le l$) in Algorithm General. Procedure B-Tree is modified from Algorithm B-Tree, and the modification is similar to that applied to Algorithm Ring to obtain Procedure Ring. In addition, the modification made to Procedure Merge_B-Tree to obtain Merge_B-Tree2 is also similar to that made to Procedure Merge_Ring to obtain Merge_Ring2.

All the previous embedding algorithms are incorporated into Algorithm General. That is, they are subcases of Algorithm General. Table 1 gives a summary of Algorithm General. Note that the directions of a torus are numbered from 0 to $d - 1$, but that if Algorithm General is invoked for torus embedding, variable i ranges from 1 to d. A torus may be viewed as a *Multi-level ring-connected ring*. We can also have an interesting

Table 1. Summary of algorithm general.

Topology	l	protocol$_i$
Ring	1	RING
Torus	d	RING for all i
Binomial tree	1	BINOMIAL TREE
Multi-level binomial tree connected binomial tree	> 1	BINOMIAL TREE for all i
Hybrid topology	> 1	variable for each i

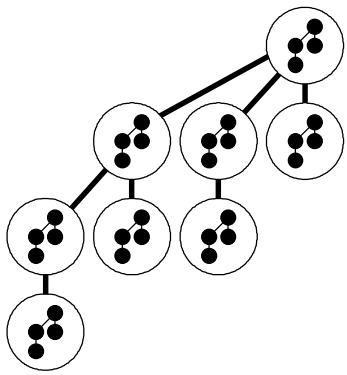


Fig. 18. 2-level binomial-tree-connected binomial tree.

topology called a *Multi-level binomial-tree-connected binomial tree* if the protocol is BINOMIAL TREE for all i and $l > 1$. Fig. 18 shows a 2-level binomial-tree-connected binomial tree that can be embedded in a Q_5. The level-1 topology is BT_2, and the level-2 topology is BT_3.

Theorem 4 The complexity of Algorithm General is $O(n)$ parallel steps.

Proof: According to our strategy, the dimensions are divided into l groups so that group i has m_i dimensions, where $1 \le i \le l$. Corresponding to each group i, Procedure Ring or B-Tree is invoked, and $O(m_i)$ parallel steps are required (from **Theorem 1**) if Procedure Ring is invoked, while no parallel steps are needed if Procedure B-Tree is invoked. Therefore, the complexity is

$$O(m_1) + O(m_2) + \ldots + O(m_l)$$
$$= O(m_1 + m_2 + \ldots + m_l)$$
$$= O(n)$$

parallel steps. ❑

6. COMPARISONS

As mentioned in the Introduction, fault-tolerant embedding in hypercubes with faulty nodes has been researched extensively, but embedding with faulty links has not yet been explored completely. A stronger existence theorem of Chan and Lee [18]. Any Q_n with at most $2n - 5$ faulty edges, in which each node is incident to at least two nonfaulty links, has a Hamiltonian cycle consisting of only nonfaulty edges. This theorem is nonconstructive, but it nearly doubles the number of faulty edges. Future research might try to design a polynomial time algorithm to find the circuit guaranteed by [2]. In [7], Latifi et al. proposed centralized sequential algorithms for ring embedding in hypercubes with faulty links. $O(n^2)$ time is required to compute the characterization of a Hamiltonian cycle, and $O(2^n)$ time is then needed to construct a Hamiltonian cycle. A Hamiltonian cycle is a ring embedded in a hypercube. Their approach is centralized since the computations are performed on the host of the hypercube. Our approach is, on the contrary, distributed such that each node makes decisions by itself. Moreover, their algorithms are based on the assumption that the number of faulty links is at most $n - 2$, while this assumption is not necessary for our approach thanks to the 2-phase strategy. Table 2 shows a comparison of the results obtained using the proposed ring-embedding algorithm and the approach in [7]. Let the number of faulty links in a Q_n be z, where $0 \le z \le n \times 2^{n-1}$. If no isolated node exists, then an embedded ring generated as shown in Fig. 6 can work through the 2-phase strategy if needed. In fact, the adoption of 2-phase routing will cause performance degradation. The performance of the proposed algorithm depends on z and the distribution of faulty links.

Table 2. Comparison 1 (for ring embedding).

	Proposed	Approach in [7]
Fault model	Link fault	Link fault
Algorithm	Distributed	Centralized
Based on	Injury Degrees of dimensions & Recursion property of hypercubes	Gray codes
Time complexity	$O(n)$ parallel steps	$O(n^2 + 2^n)$ sequential time or, $O(n^2)$ parallel time
Can handle more than $n - 2$ faulty links?	Yes	No
Suitable for	MIMD	SIMD

Yang et al. proposed the concept of free dimensions, and this concept as well as the Gray code have been applied to ring and torus embedding in hypercubes with faulty nodes [3-5]. Based on free dimensions, a hypercube can be partitioned into 3-subcubes, and there is at most one faulty node in each 3-subcube. An 8-node subring is embedded in each healthy 3-subcube, and a 6-node subring is embedded in each faulty 3-subcube.

Then, these subrings are merged into a ring. However, the internode communication details were not presented. Their method for embedding a torus in a faulty hypercube is to embed the first direction of the torus, and then use the Gray code sequence to embed other directions. Table 3 shows a comparison of the results obtained using the proposed ring-embedding and torus-embedding algorithms and the approach in [3-5].

Table 3. Comparison 2 (for ring and torus embedding).

	Proposed	Approach in [3-5]
Fault model	Link fault	Node fault
Algorithm	Distributed	Distributed
Based on	Injury Degrees of dimensions & Recursion property of hypercubes	Free dimensions & Gray codes
Time complexity	$O(n)$ parallel steps	$O(n)$ parallel steps
Suitable for	MIMD	MIMD

We have previously proposed a centralized sequential algorithm that runs in $O(2^n)$ time for binomial tree embedding in hypercubes with faulty links [9]. The binomial-tree-embedding algorithm proposed in this paper is, instead, a distributed algorithm and, thus, is more efficient. A comparison of the results obtained using the two works is shown in Table 4.

Table 4. Comparison 3 (for binomial tree embedding).

	Proposed	Approach in [9]
Fault model	Link fault	Link fault
Algorithm	Distributed	Centralized
Based on	Injury Degrees of dimensions & Recursion property of hypercubes	Cost function defined in [9]
Time complexity	$O(n)$ sequential time	$O(2^n)$ sequential time
Suitable for	MIMD	SIMD

7. SECOND FAULT-TOLERANCE PHASE AND STRATEGY FOR FAULTY NODES

In this section, the rerouting of messages performed in the second fault-tolerance phase is discussed. In addition, we explore how our embedding algorithms deal with faulty nodes. Note that if an embedded ring with faulty links is obtained due to the larger number of faulty links, then a second phase rerouting is needed to bypass the embedded faulty links.

Recall that the proposed approach is 2-phase, and in the second phase, existing fault-tolerant point-to-point communication schemes are invoked to reroute messages if

faulty links exist on the embedded graph. Since the general embedding algorithm is based on the ring-embedding and the binomial-tree-embedding algorithms, we present in this section a method for rerouting messages for ring embedding. A method for run-time message rerouting for binomial tree embedding can be found in [9].

The main idea in rerouting messages for a embedded ring with faulty links is to apply one of the fault-tolerant point-to-point communication schemes proposed in the literature [6, 8]. Because two neighboring nodes connected to a faulty link can not communicate with each other in a faulty embedded ring, these two nodes use an existing fault-tolerant routing algorithm to communicate each other. It is clear that every node in the embedded ring can send messages to any other node based on the rerouting scheme.

Theorem 5 For a faulty embedded ring without any isolated node, every node can send messages to its two neighbors in the second phase of the algorithm.

Proof: If no isolated node exists, a fault-tolerant point-point routing algorithm in [6, 8] can be applied. A node with faulty link can communicate to its neighbors through the fault-tolerant routing algorithm. It is straightforward to show that every node can send messages to any neighbor in the second phase. ❑

Fig. 19 shows an example of rerouting in the second fault-tolerance phase. In this example, since link 000-100 is faulty, ring connection 000-100 is not available. Nodes 100 and 000 must communicate with each other via a 3-hop path, e.g., 000-001-101-100. The 3-hop path can be found using existing fault-tolerant point-to-point communication schemes.

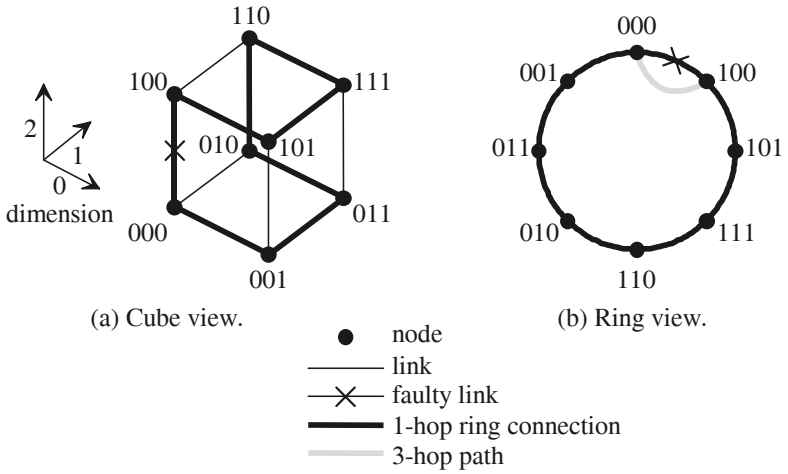
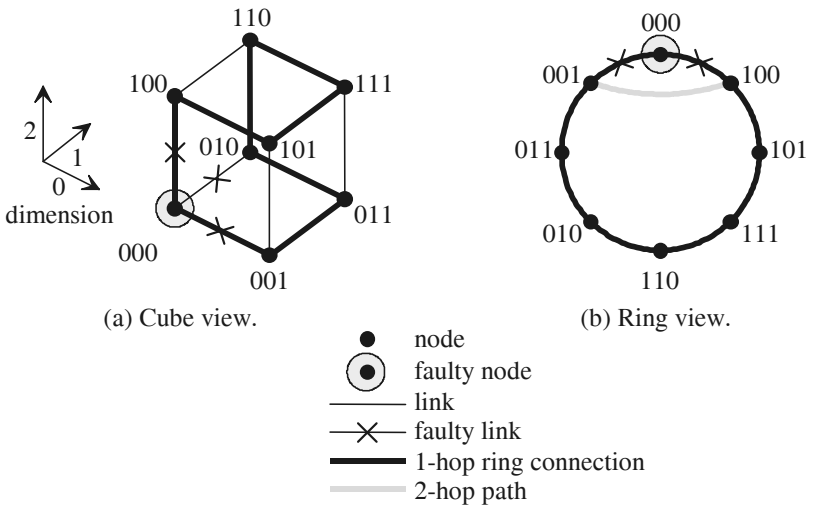


Fig. 19. Example of rerouting in the second fault-tolerance phase.

A faulty node can be treated as if all the links incident to this node are faulty. For example, in Fig. 20 (a), if node 000 is faulty, then this situation is equivalent to the case

where links 000-001, 000-100, and 000-010 are all faulty. The resulting embedded ring found by the ring-embedding algorithm is shown in Fig. 20 (b). Inevitably, there will be faulty links in the embedded ring; thus, the rerouting of messages is necessary when the ring is used in some applications. To use this ring, data for computation is distributed to all the non-faulty nodes. This technique is called UDD (*Uniform Data Distribution*) [10]. As node 000 is faulty, ring connections 000-100 and 000-001 cannot be used to route messages. Therefore, nodes 100 and 001 should route messages to each other. However, there is no link between these two nodes, so messages must be routed via a 2-hop path, e.g., 100-101-001. How do nodes 100 and 001 know that they should communicate with each other? The answer is simple. Nodes 100 and 001 broadcast messages saying that they are the two end-nodes of a broken ring; hence, they know about that they should communicate with each other.



(a) Cube view. (b) Ring view.

Fig. 20. Example of a case with faulty nodes.

Most algorithms for achieving fault-tolerance of hypercubes are based on the *faulty node model* (i.e., only faulty nodes are considered). If such algorithms are required to handle faulty links, a faulty link is modeled as if the two end-nodes of this link are faulty. The cost overhead is, thus, significantly based on this fault model since nodes are active components but links are passive components. On the other hand, modeling a faulty node as if all incident links to this node are faulty is quite natural because a faulty node will disable its adjacent neighbors from routing messages through these links. Accordingly, we believe the faulty link model is more general than the faulty node model. Fault-tolerance algorithms based on the faulty link model can handle both faulty links and/or nodes more efficiently. Fig. 21 shows a comparison between the faulty link model and the faulty node model.

(a) Modeling a faulty node by means of faulty links.

(b) Modeling a faulty link by means of faulty nodes.

Fig. 21. The faulty link model vs. the faulty node model.

8. SIMULATION RESULTS OF RING EMBEDDING

Fault-tolerant ring embedding was simulated on a 10-dimensional hypercube. The routing algorithms "Ring" shown in Fig. 6 and "Merge_Ring()" shown in Fig. 7 were implemented. The assignments of faulty links were randomly selected, and the range of the number of faulty links was varied from 1 to 50. Moreover, the $COST_R$ function for each node was adopted as shown in Fig. 5. To assign the status of each node in the embedding process, the node with the lowest $COST_R$ and the lowest label in each subring was taken as the "*ACTIVE*" node. In addition, the neighboring node of the active node with the lowest $COST_R$ and the lowest label was taken as the "*PASSIVE*" node. Nodes which were neither "*ACTIVE*" nor "*PASSIVE*" were considered as the "*IDLE*" nodes. This simulation was performed to evaluate the relationship between "*the number of faulty links*" and "*the perfect ring embedding*." Note that the perfect ring embedding means that there is no faulty link in the embedded ring. Two relative symbols used in the simulation are as follows:

FN: the number of faulty links in the 10-dimensional hypercube;
R_{FN}: the ratio used to obtain a non-perfect embedding under some *FN*.

We let *FN* be 1, 5, 10, 15, 20, 25, 30, 35, 40, 45, and 50 in the simulation, respectively. For each *FN*, algorithm "Ring" shown in Fig. 6 was executed repeatedly for 100 runs. Clearly, we could assume that: R_{FN} = (the number of non-perfect embeddings in 100 runs)/100. The simulation results are shown in Fig. 22.

From Fig. 22, it is clear that there is no non-perfect embedding when *FN* is not greater than *n* (note that *n* was equal to 10 in our simulation, i.e., the dimension of a hypercube.). When *FN* is greater than 1.5*n,* the probability of obtaining a non-perfect embedding is over 20%. The probability curve grows quickly from $FN = 1.5n$ to $FN = 2.5n$. When *FN* is greater than 5*n*, it is difficult to find a perfect ring embedding. Therefore, if the fault-assignments are made randomly, we can conclude that:

1. There is near 100% probability of obtaining a perfect ring embedding when *FN* is not greater than *n*.
2. There is near 50% probability of obtaining a perfect ring embedding when *FN* is about 2*n*.
3. There is near 0% probability of obtaining a perfect ring embedding when FN is greater than 5*n*.

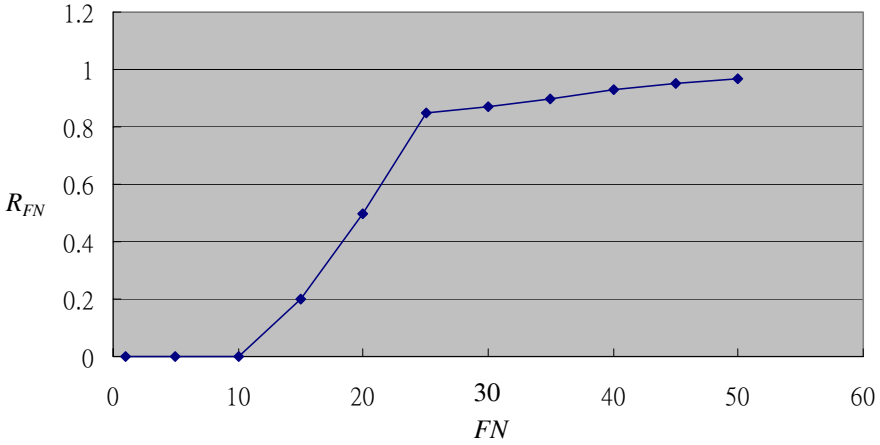


Fig. 22. The simulation results of ring embedding in a 10-dimensional hypercube.

9. CONCLUSIONS

In this paper, we have studied the fault-tolerant embedding of several topologies into hypercubes, including the ring, the torus, the binomial tree, and a hybrid topology, which is a combination of rings and binomial trees. Although these algorithms are based on the faulty link model, they can also be utilized to deal with faulty nodes. The complexity of all the algorithms except for the sequential Algorithm B-Tree is $O(n)$ parallel steps, where n is the dimension of the hypercube, and the complexity of Algorithm B-Tree is $O(n)$.

Communication is unavoidable in parallel computing applications, and the communication patterns are intrinsically associated with the applications themselves. Therefore, the embedding of communication pattern graphs into the topologies of multiprocessor structures is of great importance. In some cases, a 100-percent fault-tolerant embedding is possible. That is, there are no faulty nodes or links in the mapping of the communication pattern graph on the multiprocessor structure. However, in many cases, a perfect mapping is not achievable because of the distribution of faulty links/nodes. Under such circumstances, an efficient way to increase the reliability of the applications is to adopt a two-phase fault-tolerance strategy. First, a near-perfect embedding is found, and then fault-tolerant point-to-point communication schemes are applied to for the embedded communication pattern graph. We believe that combining fault-tolerant embedding and fault-tolerant communication is a better strategy than just using one of them alone.

REFERENCES

1. M. Y. Chan and S. J. Lee, "Distributed fault-tolerant embedding of rings in hypercubes," *Journal of Parallel and Distributed Computing*, Vol. 11, 1991, pp. 63-71.
2. F. J. Provost and R. Melhem, "Distributed fault-tolerant embedding of binary trees and rings in hypercubes," in *Proceedings of International Workshop on Defect and Fault Tolerance in VLSI Systems*, 1988.

3. P. J. Yang, S. B. Tien, and C. S. Raghavendra, "Embedding of rings and meshes onto faulty hypercubes using free dimensions," *IEEE Transactions on Computers*, Vol. 43, 1994, pp. 608-613.
4. C. S. Raghavendra, P. J. Yang, and S. B. Tien, "Free dimension – an effective approach to achieving fault tolerance in hypercubes," in *Proceedings of IEEE 22th International Symposium on Fault-Tolerant Computing*, 1992, pp. 170-177.
5. P. J. Yang, S. B. Tien, and C. S. Raghavendra, "Embedding of multidimensional meshes onto faulty hypercubes," in *Proceedings of International Conference on Parallel Processing*, 1991, pp. I571-I574.
6. M. S. Chen and K. G. Shin, "On hypercube fault-tolerant routing using global information," in *Proceedings of Conference on Hypercubes*, *Concurrent Computers and Applications*, 1989, pp. 83-86.
7. S. Latifi, S. Q. Zheng, and N. Bagherzadeh, "Optimal ring embedding in hypercubes with faulty links," in *Proceedings of IEEE 22th International Symposium on Fault-Tolerant Computing*, 1992, pp. 178-184.
8. Y. R. Leu and S. Y. Kuo, "Distributed fault-tolerant ring embedding and reconfiguration in hypercubes," *IEEE Transactions on Computers*, Vol. 48, 1999, pp. 81-86.
9. Y. R. Leu and S. Y. Kuo, "A fault-tolerant tree communication scheme for hypercube systems," *IEEE Transactions on Computers*, Vol. 45, 1996, pp. 641-650.
10. S. Park and B. Bose, "Broadcasting in hypercubes with link/node failures," in *Proceedings of 4th Symposium on the Frontiers of Massively Parallel Computation*, 1992, pp. 286-290.
11. A. C. Elster, M. U. Uyar, and A. P. Reeves, "Fault-tolerant matrix operations on hypercube multiprocessors," in *Proceedings of International Conference on Parallel Processing*, 1989, pp. III 169-177.
12. M. S. Chen and K. G. Shin, "Processor allocation in an *n*-cube multiprocessor using gray codes," *IEEE Transactions on Computers*, Vol. c-36, 1987, pp. 396-407.
13. G. Fox et al., Ch. 9, *Long Range Interactions*, *Solving Problems on Concurrent Processors*, Prentice-Hall, Inc., Vol. 1, 1988, pp. 155-165, 1988.
14. Y. Saad and M. H. Schultz, "Topological properties of hypercube," *IEEE Transactions on Computers*, Vol. C-37, 1988, pp. 867-872.
15. T. C. Lee, "Quick recovery of embedded structures in hypercube computers," in *Proceeding of 5th Memory Computing Conference*, 1990, pp. 1426-1435.
16. E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms*, Prentice-Hall, Inc. 1977.
17. J. Bruck, R. Cypher, and D. Soroker, "Tolerating faults in hypercubes using subcube partitioning," *IEEE Transactions on Computers*, Vol. 41, 1992, pp. 599-605.
18. M. Y. Chan and S. J. Lee, "On the existence of Hamiltonian circuits in faulty hypercubes," *SIAM Journal on Discrete Mathematics*, 1991, pp. 511-527.
19. J. Wu, E. B. Fernandez, and Y. Luo, "Embedding of binomial trees in hypercubes with link faults," *Journal of Parallel and Distributed Computing*, 1998, pp. 59-74.

Shih-Chang Wang (汪世昌) received the B.S. (1992) degree in Computer Science and Engineering from the Tatung University, the M.S. (1994) and Ph.D. (2000) degrees in Electrical Engineering from the National Taiwan University, Taipei, Taiwan. Since 2000 he has joined the Telecommunication Lab., Chunghwa Telecom Co., Ltd., where he is currently an associate researcher. His current research interests include fault tolerance, enterprise resource planning, management information system, Java technology, and computer networks.

Yuh-Rong Leu (呂毓榮) graduated with an Electrical Engineering degree from National Taiwan University, Taiwan, in 1991 and received the Ph.D. in Electrical Engineering also from National Taiwan University in 1996. During 1996 to 2000, he worked for the government-supported Institute for Information Industry, Taiwan as a senior software engineer. Currently he is the CTO of InterEpoch Technology, Taiwan (http://www.interepoch.com.tw), which is focused on wireless LAN and AAA (Authentication, Authorization, and Accounting) technologies.

Sy-Yen Kuo (郭斯彥) received the B.S. (1979) in Electrical Engineering from National Taiwan University, the M.S. (1982) in Electrical and Computer Engineering from the University of California at Santa Barbara, and the Ph.D. (1987) in Computer Science from the University of Illinois at Urbana-Champaign. Since 1991 he has been with National Taiwan University, where he is currently a professor and the Chairman of Department of Electrical Engineering. He spent his sabbatical year as a visiting researcher at AT&T Labs-Research, New Jersey from 1999 to 2000. He was the Chairman of the Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan from 1995 to 1998, a faculty member in the Department of Electrical and Computer Engineering at the University of 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 Jet Propulsion Laboratory of California Institute of Technology. His current research interests include mobile computing and networks, dependable distributed systems, software reliability, and optical WDM networks.

Professor Kuo is an IEEE Fellow. He has published more than 200 papers in journals and conferences. He received the distinguished research award (1997-2005) from the National Science Council, Taiwan. He was also a recipient of the Best Paper Award in the 1996 International Symposium on Software Reliability Engineering, 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.