Distributed Fault-Tolerant Ring Embedding and Reconfiguration in Hypercubes

Yuh-Rong Leu and Sy-Yen Kuo, *Member*, *IEEE*

Abstract—To embed a ring in a hypercube is to find a *Hamiltonian cycle* through every node of the hypercube. It is obvious that no 2^n-node Hamiltonian cycle exists in an n-dimensional faulty hypercube which has at least one faulty node. However, if a hypercube has faulty links only and the number of faulty links is at most $n - 2$, at least one 2^n-node Hamiltonian cycle can be found. In this paper, we propose a distributed ring-embedding algorithm that can find a Hamiltonian cycle in a fault-free or faulty n-dimensional hypercube (Q_n), and the complexity is $O(n)$ parallel steps. The algorithm is based on the recursion property of the hypercube and the *free-link dimension* concept. In some cases, even when the number of faulty links is larger than $n - 2$, Hamiltonian cycles may still exist. We will show that the largest possible number of faulty links that can be tolerated is $2^{n-1} - 1$. The performance and the constraints of the fault-tolerant algorithm is also analyzed in detail in this paper. Furthermore, a dynamic reconfiguration algorithm for an embedded ring is proposed and discussed. Due to the distributed nature of the algorithms, they are useful for the simulation of ring-based multiprocessors on MIMD hypercube multiprocessors.

Index Terms—Hamiltonian cycle, faulty link, hypercube, free-link dimension, reconfiguration.

1 INTRODUCTION

THE hypercube is a topology with large connectivity. A vertex in a hypercube graph is connected to n adjacent vertices by n edges, where n is the dimension of the hypercube. Therefore, many other topologies can be embedded in a hypercube, such as the linear array, mesh, binary tree, ring, etc. For many applications on multiprocessors, selecting an appropriate and adequate topological structure of multiprocessors is a critical issue. The reason is that most applications have their own intrinsic communication patterns. By means of embedding, applications that are originally suitable for multiprocessors of other topological structures can be executed on hypercube multiprocessors.

It is well known that systems with a large number of components are more prone to failures. Accordingly, it is not uncommon that there are some link and/or node failures in a hypercube system. Under such circumstances, straightforward embedding may not be possible. Thus, fault-tolerant embedding has become an intriguing issue. Many researchers have addressed this topic [1], [2], [3], [4], [5], [6] extensively, but most of them considered node failures only. In [3], centralized algorithms are devised for optimal ring embedding in an n-dimensional hypercube Q_n with at most $n - 2$ faulty links. The goal of the work in [3] is to find a Hamiltonian cycle through every fault-free node of a hypercube Q_n. Their algorithms take $O(n^2)$ time to compute the characterization of a Hamiltonian cycle and, then, $O(2^n)$ time to construct a Hamiltonian cycle, i.e., $O(n^2 + 2^n)$ time totally. Our goal is the same, but we achieve this goal in a different and more efficient manner. In this paper, we present distributed algorithms for ring embedding and dynamic reconfiguration. Hence, our work is applicable on MIMD hypercube multiprocessors.

Our approach is based on the recursion property of the hypercube and the *free-link dimension* concept. The free-node dimension concept in [7] applies to node failures while we deal with link failures. The free-node dimension is defined in [7] as that a dimension in a hypercube is said to be free if no pair of nodes across any link in that dimension are both faulty. The concept of free-node dimension was used to embed a ring in a Q_n with faulty nodes [5]. In [5], the embedding procedure is briefly described. A six-node or eight-node subring is first embedded onto each partitioned Q_3 and, then, all the embedded subrings are merged in a distributed manner. However, the details about the internode communications are omitted in [5]. Considering link failures, we say that a dimension is free if all links of this dimension are nonfaulty. We use the concept of free-link dimension to construct subrings, and then these subrings are recursively merged to form larger subrings until the desired largest ring is obtained. Although our approach is motivated by the ring embedding method described in [5], the concentration on faulty links is very different and unique. Furthermore, we utilize the recursion property of the hypercube to merge subrings efficiently.

The organization of this paper is as follows. In Section 2, the basic concepts regarding ring-embedding are given. In Section 3, we propose a distributed fault-tolerant ring-embedding algorithm for hypercubes. Detailed analysis and comparisons are also provided in this section. Section 4 presents a dynamic reconfiguration algorithm for an embedded ring. Conclusions are given in Section 5.

2 PRELIMINARIES

An n-dimensional hypercube denoted as Q_n has 2^n (= 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 an arbitrary 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 in {0, 1, x}, where x stands for *don't care*. For instance, subcube 0xx1 is composed of four nodes—0001, 0011, 0101, and 0111. An m-dimensional subcube is called an m-subcube.

A Hamiltonian cycle in Q_n is a path $u_0 \rightarrow u_1 \rightarrow \dots \rightarrow u_{N-1} \rightarrow u_0$, where u_i ($0 \le i \le N - 1$) is a node of the hypercube and the labels of $u_{i \bmod N}$ and $u_{i+1 \bmod N}$ differ in only one bit position. Exploiting the Gray code (GC) concept, the node label sequence of a Hamiltonian cycle can be found easily [8]. For example, the binary reflective Gray code determines a Hamiltonian cycle (see Fig. 1).

Let b_i denote bit i of a binary reflected Gray code b and Ξ be a concatenation operator such that $\Xi(b, 0, 1, \dots, n - 1) = b_{n-1}b_{n-2}\dots b_0 = b$. Moreover, let $\pi = (g_0, g_1, \dots, g_{n-1})$ be a permutation of $\{0, 1, \dots, n - 1\}$. Then $b' = \Xi(b, \pi)$ is a new code whose bit i is bit g_i of b. Given π, a new sequence of codes can be generated by the above operation, namely, a new Hamiltonian cycle is found. By this method, $n!$ Hamiltonian cycles can be identified from a starting node u_0. That is, the upper bound of the total number of distinct Hamiltonian cycles is $n! \times 2^n$ if based on the GC concept. However, if not based on the GC, the total number $h(n)$ of Hamiltonian cycles in Q_n is found in [9] to be much larger than $n! \times 2^n$ for large n, e.g., $h(3) = 6$ and $h(4) = 1,344$. Our distributed approach is not based on GC, so the number of distinct Hamiltonian cycles which can be generated will be larger than that can be generated by using GC.

• *Y.-R. Leu is with the Communication Network Lab., Institute for Information Industry, Taipei, Taiwan.*
• *S.-Y. Kuo is with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan. E-mail: sykuo@cc.ee.ntu.edu.tw.*

Manuscript received 28 Oct. 1995; revised 30 Mar. 1998.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 107129.

0018-9340/99/$10.00 © 1999 IEEE

0000	1100
0001	1101
0011	1111
0010	1110
0110	1010
0111	1011
0101	1001
0100	1000

Fig.1. Four-bit binary reflective Gray code.

A fault-free hypercube is also called a healthy hypercube. A dimension i is said to be *injured* if one or more i-links are faulty. The level of injury of a dimension is defined as the number of faulty links on that dimension. The dimensions can then be ranked according to their level of injury. If all the links on a certain dimension are fault-free, this dimension is said to be *free*. In a Hamiltonian cycle, each vertex has exactly two edges. Consequently, if the number of faulty links in a hypercube is less than or equal to $n - 2$, at least one Hamiltonian cycle exists. By this assumption, at least two dimensions are guaranteed to be free. That is to say, a faulty hypercube can be partitioned along those injured dimensions into 2^{n-2} subcubes, each two-dimensional. A two-dimensional subcube is itself a four-node subring. Recursively merging these subrings, a 2^n-node ring can be eventually constructed.

The following notations will be used in the rest of this paper.

b_i: bit i of a binary word b.
I_i: an n-bit binary word with bit $i = 1$ and other bits 0.
0^k: a k-bit binary word with all bits = 0.
1^k: a k-bit binary word with all bits = 1.
x^k: a k-bit binary word with all bits = don't care.
pq: a binary word formed by concatenating binary words p and q.
mask: an n-bit binary word used for masking node labels. It is initialized to be 0^n.
curr: the label of the current node, i.e., the node executing the algorithm.
neib: the label of a neighbor of the current node.
u-v: a ring connection between nodes u and v.
$\oplus$: bitwise exclusive-OR operator.
$\wedge$: bitwise AND operator.
$H(a, b)$: the Hamming distance between binary words a and b, i.e., $H(a, b) = \Sigma_j c_j = \Sigma_j (a \oplus b)_j$.

3 Distributed Ring-Construction Algorithm for Q_n

3.1 Algorithm

The algorithm presented in this section is capable of finding a ring (Hamiltonian cycle) in Q_n with faulty links (up to $n - 2$ faulty links). We assume each node has the locations of all faulty links. This assumption is reasonable because an optimal broadcasting algorithm which takes $n + 1$ steps in a hypercube with faulty links has been developed [10]. Therefore, it is possible for nodes in a 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. The dimensions are divided into two groups: free dimensions and injured dimensions. The free dimensions are stored in a stack called E. The injured dimensions are sorted by the number of faulty links and kept in a stack called F, with the top of F being the dimension with the smallest number of faulty links. More details will be given later.

The algorithm has n steps numbered from 0 to $n - 1$. Stack E or F is popped to give a dimension in each step. Since at most $n - 2$ faulty links are assumed to exist in Q_n, at least two dimensions are free. Consequently, in Step 0, 2^{n-1} one-subcubes are formed across the first free dimension popped from stack E. And, in Step 1, each distinct pair of one-subcubes are merged across the second free dimension into a two-subcube, i.e., a $4(= 2^2)$-node subring. 2^{n-2} two-subcubes are generated in this step. If stack E is not empty yet, subrings can be further merged until stack E becomes empty. Then, stack F is popped to give the next dimension across which subrings are further merged. 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. Two subrings that can be merged in step i form a *merging group*. So, there are 2^{n-i-1} merging groups in Step i.

Four nodes are involved in merging two subrings in each Step i. Let nodes a and b be in one subring and 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. $(a \oplus c) = (b \oplus d) = I_{g_i}$,

where g_i is the dimension poped in Step i.

Merging the two subrings means breaking connections a-b and c-d and establishing connections a-c and b-d. Links a-c and b-d form a *merging link pair*. Link a-c is said to be the *merging partner* of link b-d and vice versa. The most complicated part of Algorithm RingConstruct is obviously how to find nodes a, b, c, and d in each merging step in the presence of faulty links. Fig. 2 shows an example for Step 2. In Fig. 2, $a = 101$, $b = 111$, $c = 001$, and $d = 011$.

Before further discussion, we introduce the concept of *pseudofaulty* links. The introduction of pseudofaulty links is to maintain the uniformity of the structures of subrings formed in Step i. That is, all node labels of a formed subring must be the same as 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 pseudofaulty. That is, a faulty link makes an "image" in each merging group. Algorithm RingConstruct does not care whether a link is faulty or pseudofaulty. Fig. 3 illustrates the concept of pseudofaulty links. By the introduction of the pseudofaulty link concept, the operations in each merging group are uniform.

The condition for a node N that can involve in merging operations in Step i is:

> the g_i -links connected to node N and at least one of its ring neighbors are nonfaulty or nonpseudofaulty.

For the convenience of presenting Algorithm RingConstruct, this condition is refered as the *merging condition*. Note that two g_i-links are required for merging operations in Step i because they are used as ring connections in the merged ring. Each node can check easily if itself and other nodes in the subring satisfy the merging condition since it has the information about the locations of all faulty links. The node with minimal label is chosen to initiate the merging operations. As an example, in Fig. 2, nodes 001, 011, and 010 satisfy the merging condition in the inner subring and nodes 101, 111, and 110 satisfy the merging condition in the outer subring. Node 001 (101) knows that it is the node to initiate the merging operations. And, then, node 011 (111) is informed to delete connection 001-011 (101-111). Finally, connections 001-101 and 011-111 are established. Algorithm RingConstruct is shown in Fig. 4.

In a ring, each node has two connections to its adjacent nodes. Two variables d_1 and d_2 are used to store the dimensions of these connections. The variable *mask* is used for testing node label and is

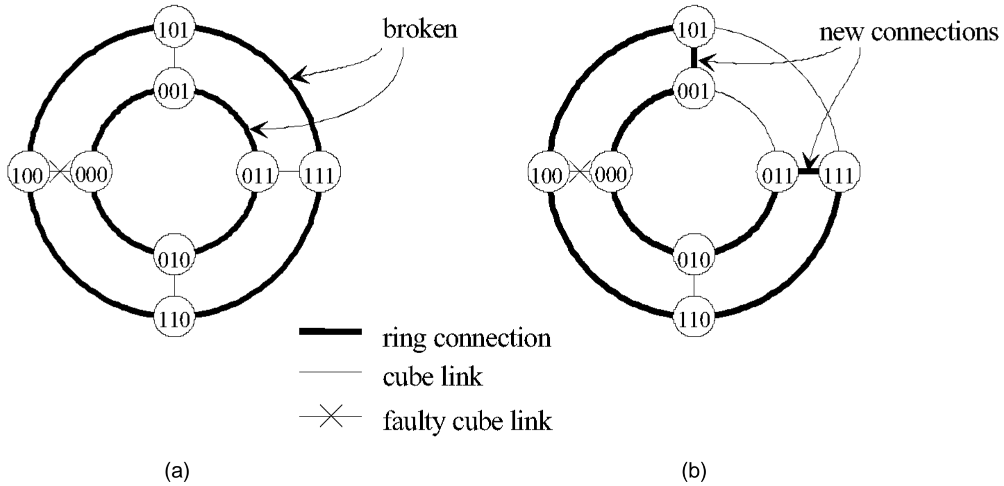

Fig. 2. Merging two subrings in the presence of faulty links. (a) Before merging. (b) After merging.

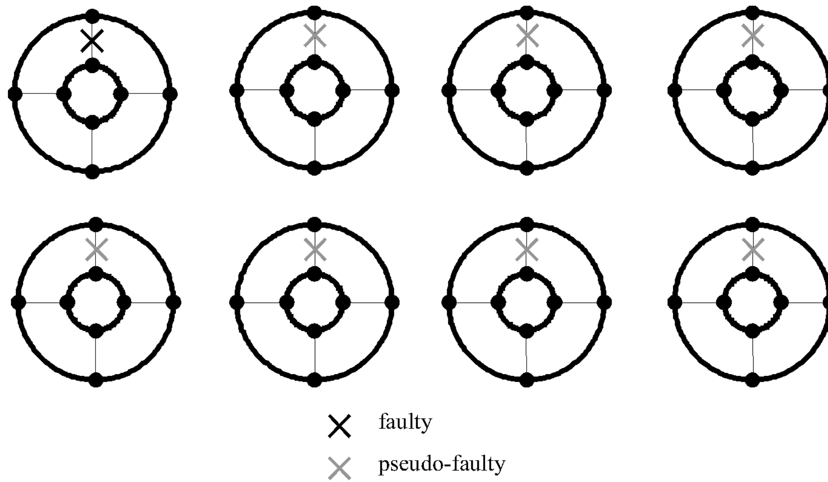

Fig. 3. The concept of pseudofaulty links.

initialized to 0^n. At the time $i = 0$, each node makes a connection to its neighbor (with label $curr \oplus Ig_0$) in dimension g_0. And, d_1 of each node is set to g_0. Messages are not required to establish connections. To make a connection, a node simply adds a record in its memory for identifying who is its neighbor in the ring. Finally, *mask* is changed to Ig_0. At time $i = 1$, each node makes a connection to its neighbor (with label $curr \oplus Ig_1$) in dimension g_1. And d_2 of each node is set to g_1. The variable *mask* is assigned to $Ig_0 \oplus Ig_1$. At the present time, there are 2^{n-2} four-node subrings in Q_n.

In iterations from $i = 2$ to $n - 1$, subrings are recursively merged. To merge two subrings, one connection of each subring must be broken and two new connections are established. A node simply deletes the corresponding connection record to break a connection. 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 (i.e., merging link pair) 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 of i increases. So, it is desired that there are fewer faulty links in a dimension g_i with smaller i. This is the reason why the top of stack F is the dimension with the smallest number of faulty links. If the injured dimensions are not sorted by the number of faulty links, after the free dimensions are used up, the next dimension g_i may have so many faulty links to make Algorithm RingConstruct fail (refer to Section 3.2). Since we sort the injured dimensions, the probability that the algorithm will fail is reduced.

3.2 Analysis

If two 2^k-node subrings can be merged across dimension g, there are 2^k g-links connecting them and at least two nodes in each subring satisfy the *merging condition.* What is the smallest number of faulty g-links, out of the 2^k g-links, that may prevent all nodes in the two subrings from satisfying the merging condition? Lemma 1 answers this question.

LEMMA 1. *Two 2^k-node subrings can be merged across dimension g if the number of faulty g-links corresponding to the two subrings is smaller than 2^{k-1}.*

PROOF. Let's focus on just one subring. Arbitrarily number each node in the subring from 1 to 2^k. The simplest case that all nodes fail to satisfy the merging condition is that the g-links of either nodes 1, 3, 5, …, and $2^k - 1$ or nodes 2, 4, 6, …, and 2^k are faulty. There are 2^{k-1} faulty g-links in both cases. So, if the number of faulty g-links is smaller than 2^{k-1}, the two subrings are guaranteed to be merged across dimension g. Fig. 5 shows an example that two subrings cannot be merged. □

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. Therefore, the total number of faulty

```
     Algorithm RingConstruct;
     {Each node has two ring connections in dimensions d1 and d2, respectively. Stack E (F) stores free (injured)
       dimensions.}
1.   begin
2.       mask ← 0^n;
3.       for i = 0 to n–1 do begin
4.           if stack E is not empty then g ← pop (E)
5.           else g ← pop (F);
6.           case i of
7.               0: begin d1 ← g; neib ← curr ⊕ I_g; establish connection curr-neib end;
8.               1: begin d2 ← g; neib ← curr ⊕ I_g; establish connection curr-neib end;
9.           2..n–1: begin
10.                  if (curr ∧ mask = 0^n and stack E is not empty)
11.                  or curr is the minimal node label among the nodes satisfying the merging condition
12.                    then begin
13.                     case of
14.                        stack E is not empty or both ring neighbors of node curr
15.                        also satisfy the merging condition:
16.                           begin d ← min(d1, d2); min(d1, d2) ← g end;
17.                        only ring neighbor of node curr along dimension d_k (k=1 or 2)
18.                        satisfies the merging condition:
19.                           begin d ← d_k; d_k ← g end;
20.                     endcase;
21.                        neib ← curr ⊕ I_d;
22.                        Send a break-ring-connection message to node neib via link d;
23.                        Break connection curr-neib;
24.                        neib ← curr ⊕ I_g;
25.                        Establish connection curr-neib;
26.                    endif
27.                  else
28.                  if a break-ring-connection message is received via link l
29.                    then begin
30.                        if d1 = l then d1 ← g
31.                        else d2 ← g;
32.                        neib ← curr ⊕ I_l;
33.                        Break connection curr-neib;
34.                        neib ← curr ⊕ I_g;
35.                        Establish connection curr-neib;
36.                    endif;
37.              end;
38.          endcase;
39.          mask ← mask ⊕ I_g;
40.      endfor;
41.  end.
```

Fig. 4. Algorithm RingConstruct.

g_i-links must be smaller than $2^{n-i-1} \times 2^{i-1} = 2^{n-2}$. Although this number is independent of *i*, yet the distribution of the locations of faulty g_i-links is crucial. It may be the case that the total number of faulty g_i-links is smaller than 2^{n-2}, but two 2^i-node subrings cannot be merged. This occurs if the number of faulty g_i-links corresponding to the two 2^i-node subrings is greater than or equal to 2^{i-1}. However, if the total number of faulty g_i-links is smaller than 2^{i-1}, all 2^{n-i} 2^i-node subrings can be merged in Step *i*. If the number of faulty links is at most $n - 2$, this fact is always true for any *i*.

THEOREM 1. *Algorithm RingConstruct will find a 2^n-node Hamiltonian cycle in a faulty Q_n if the number of faulty links is at most $n - 2$.*

PROOF. Let the number of injured dimensions be *t*. Then, the number of elements in stack *F* is *t* and the number of elements in stack *E* is $n - t$. Hence, dimensions g_0 through g_{n-t-1} are free dimensions, while dimensions g_{n-t} through g_{n-1} are injured dimensions. Let f_{g_i} be the total number of faulty g_i-links and *m* be the total number of faulty links. We have:

$$f_{g_i} = 0, \;\; 0 \le i \le n - t - 1 \tag{1}$$

$$f_{g_i} \le f_{g_j}, \;\; n - t \le i \le j \le n - 1 \tag{2}$$

$$\sum_{i=0}^{n-1} f_{g_i} = m \le n - 2 \tag{3}$$

Equation (2) is true because the injured dimensions are sorted. If $f_{g_i} < 2^{i-1}$ for $n - t \le i \le n - 1$, then this theorem is proved. However, this fact is implied by (2) and (3). By (2), f_{g_i} is a nondecreasing function of *i*, and $f_{g_{n-t}}$ is the minimum in interval $n - t \le i \le n - 1$. Since 2^{i-1} is an exponential function of *i*, the only possible case for $f_{g_i} \ge 2^{i-1}$ might be at

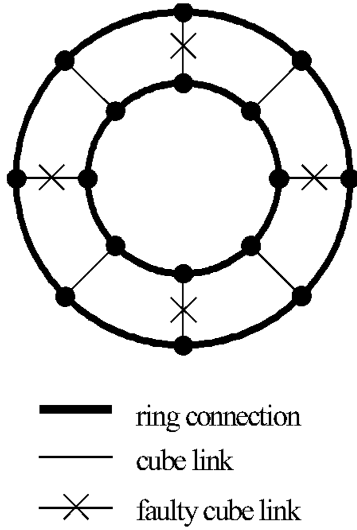

Fig. 5. Example for Lemma 1.

$i = n - t$. However, we will show that this cannot occur. According to (3), it can be easily seen that the maximum value of $f_{g_{n-t}}$ is $\lceil m/t \rceil \le \lceil (n-2)/t \rceil$. This happens when f_{g_i} is either $\lceil m/t \rceil$ or $\lfloor m/t \rfloor$. If $t = 0$, it's a fault-free case, so we consider just the case $1 \le t \le n - 2$. And, in this interval ($1 \le t \le n - 2$), it is true that $\lceil (n-2)/t \rceil < 2^{n-t-1}$. Let's check the boundary conditions:

$$t = 1,\ \lceil (n-2)/t \rceil = \lceil n-2 \rceil < 2^{n-2} = 2^{n-t-1}$$
$$t = n-2,\ \lceil (n-2)/t \rceil = 1 < 2 = 2^{n-t-1}.$$

Besides, $\lceil (n-2)/t \rceil$ and 2^{n-t-1} are both decreasing functions of t. Consequently, $\lceil (n-2)/t \rceil < 2^{n-t-1}$ is true for $1 \le t \le n - 2$. We can therefore conclude that $f_{g_i} < 2^{i-1}$ is true for $n - t \le i \le n - 1$. The theorem is proven. Fig. 6 demonstrates two cases of f_{g_i}. □

THEOREM 2. *Algorithm RingConstruct finds a Hamiltonian cycle in a Q_n with $O(n)$ parallel steps.*

PROOF. With the introduction of pseudofaulty links, each merging group is in the same status before merging. Therefore, all formed subrings are guaranteed to be of the same structure, and this fact makes Algorithm RingConstruct execute in a symmetric manner in each step. The remaining proof is by induction on n. It is obviously true for $n = 2$. Assume it is also true for $n = m - 1$. Considering $n = m$, partition $Q_n = Q_m$ along dimension $m - 1$ into two Q_{m-1}s, i.e., $1x^{m-1}$ and $0x^{m-1}$. By the assumption, Algorithm RingConstruct can find a Hamiltonian cycle in each Q_{m-1} and the node label sequences of both Hamiltonian cycles are the same except in bit $m - 1$. Because Algorithm RingConstruct is capable of merging subrings, these two 2^{m-1}-node Hamiltonian cycles can be merged into a 2^m-node Hamiltonian cycle in iteration $m - 1$. Therefore, by the principle of mathematical induction, the theorem is proven. □

When there are no faulty links (stack F is empty), the fault-tolerant portion of Algorithm RingConstruct is not reached and Algorithm RingConstruct can be easily modified to find a large number of distinct Hamiltonian cycles for fault-free hypercubes. Since the order of dimension we push into stack E can be any permutation of integers from 0 to $n - 1$, i.e., $(g_0, g_1, \ldots, g_{n-1})$, and if the condition in line 10 is modified to $curr \wedge mask = 0^{n-i}h(i)$, where $h(i)$ is an arbitrarily selected i-bit binary string, the total number of distinct Hamiltonian cycles can be found by our recursively merging approach is tremendous, much larger than by GC ($n! \times 2^n$).

THEOREM 3. *The number of distinct Hamiltonian cycles that can be found by the modified Algorithm RingConstruct for healthy hypercubes is $n! \times 2^{(n-1)(n-2)/2-1}$.*

PROOF. Let $R(n)$ be the number of distinct Hamiltonian cycles. The ring is formed in a bottom-up manner. However, we analyze $R(n)$ in a top-down manner. Partitioning Q_n along one of the n dimensions, we get two Q_{n-1}s. Thus, $R(n)$ can be represented by $R(n - 1)$. Moreover, to merge two 2^{n-1}-node subrings, we have 2^{n-1} candidate links to break. Accordingly, we have the following recurrence equation:

$$R(n) = \frac{1}{2} \cdot C_1^n \cdot 2^{n-1} \cdot R(n-1),$$
$$R(n) = 1.$$

Reading the node labels of a ring clockwise and counterclockwise gives two node-label sequences, i.e., two node-label sequences stand for just one ring. So, the term 1/2 is necessary. By solving the recurrence equation, we get $R(n) = n! \times 2^{(n-1)(n-2)/2-1}$. □

Fig. 6. f_{g_i} vs. 2^{i-1}.

TABLE 1
COMPARISON RESULTS

	Our approach	Latifi et al.'s approach [3]
Algorithm	Distributed	Centralized
Based on	Free-link dimension concept	Gray code concept
Time complexity	$O(n)$ parallel steps	$O(n^2+2^n)$ sequential time
Can handle more than $n-2$ faulty links?	Yes (up to $2^{n-1}-1$ faulty links)	No
Suitable for	MIMD	SIMD
Useful if fail?	Yes	No

3.3 Comparisons

There are many differences between our approach and that of [3]. The major difference is that our approach is distributed and theirs is centralized. They utilized the concept of Gray code to solve the problem, while our approach is based on the free-link dimension concept. If there are more than $n - 2$ faulty links in at most $n - 2$ dimensions, a 2^n-node Hamiltonian cycle may still exist. This fact was briefly mentioned in [3], but they left this as an open problem. Although originally we assume at most $n - 2$ faulty links exist in Q_n, our distributed approach still works if there are more than $n - 2$ faulty links in at most $n - 2$ dimensions. This can be easily verified by tracing Algorithm RingConstruct. From the proof of Theorem 3, we know that Algorithm RingConstruct will work successfully if $f_{g_i} < 2^{i-1}$ for $n - t \le i \le n - 1$. The situation that there are more than $n - 2$ faulty links in at most $n - 2$ dimensions means that $t \le n - 2$ and $\Sigma_i f_{g_i} = m \ge n - 2$. So, in Fig. 6, if the curve of function f_{g_i} is under the curve of function 2^{i-1}, Algorithm RingConstruct works. Since $f_{g_i} = 0$ for $0 \le i \le n - t - 1$, it is true that $f_{g_i} < 2^{i-1}$ for $0 \le i \le n - 1$. The upper bound of m can be determined as follows:

$$
\begin{aligned}
m &= \sum_{i=0}^{n-1} f_{g_i} < \sum_{i=0}^{n-1} 2^{i-1} \\
&= \frac{1}{2} + (1 + 2 + 2^2 + \cdots + 2^{n-2}) \\
&= \frac{1}{2} + (2^{n-1} - 1) = 2^{n-1} - \frac{1}{2} \\
&\therefore m \le 2^{n-1} - 1.
\end{aligned}
$$

This means that the largest possible number of faulty links which can be tolerated by Algorithm RingConstruct is $2^{n-1}-1$. We therefore have the following theorem.

THEOREM 4. *Algorithm RingConstruct will find a 2^n-node Hamiltonian cycle in Q_n if $t \le n - 2$, $m \le 2^{n-1} - 1$, and $f_{g_i} < 2^{i-1}$ for $0 \le i \le n - 1$.*

Because the conditions under which Algorithm RingConstruct works are not tight, it can be applied to most cases.

Latifi et al.'s work [3] just theoretically/mathematically gives the node sequence of a fault-free Hamiltonian cycle, but how this information is used to form a ring on a real hypercube system has not been addressed. That is, when someone wants to run a ring-based application on a hypercube system, how does he/she configure the nodes of the hypercube to simulate a ring-based multiprocessor using the node sequence? On the contrary, at the end of execution of Algorithm RingConstruct, a ring is formed and all the connections for the ring have been established, i.e., each node knows who its neighbors are. Then, the hypercube can be used as a ring-based multiprocessor.

There is one more advantage of our approach. With our strategy, Algorithm RingConstruct always strives to find Hamiltonian cycles of largest possible size even when a 2^n-node cycle cannot be found. For example, if $f_{g_i} \ge 2^{i-1}$ for $i = n - 1$, Algorithm RingConstruct will stop at $i = n - 1$. However, two 2^{n-1}-node rings have already been formed in the hypercube. That is, although the hypercube cannot simulate a 2^n-node ring in this case, it can simulate two 2^{n-1}-node rings instead. The time spent in executing Algorithm RingConstruct is therefore not wasted. Consequently, Algorithm RingConstruct can still be useful even when it fails to generate a 2^n-node ring. Table 1 shows the comparison results of our approach to that of [3].

4 Dynamic Reconfiguration

Embedding is a static fault tolerance strategy since the faulty links are identified before the fault-tolerant-embedding algorithm starts. In this section, we will show how an embedded ring is reconfigured when a new link failure occurs.

Our embedding approach is based on the recursive merging. In the last merging step, two subrings are merged into the final ring. When a link which is part of a ring connection becomes faulty, the ring is broken into a chain. The reconfiguration process first constructs two subrings from the chain and, then, remerges the two subrings.

Recall that there are four nodes involved in merging two subrings, say *a*, *b*, *c*, and *d*. If nodes *a* and *b* are in one subring and nodes *c* and *d* are in another, then merging the two subrings means breaking connections *a*-*b* and *c*-*d* and establishing connections *a*-*c* and *b*-*d*. Due to this nature of our merging scheme, when a link which is part of a connection of the embedded ring becomes faulty, say link *b*-*d*, the merging partner of *b*-*d*, i.e., link *a*-*c*, can be found. By breaking connections *a*-*c* and *b*-*d* and establishing connections *a*-*b* and *c*-*d*, we get two subrings (see Fig. 7). The two subrings can then be remerged into a ring. The remerging operations are the same as the operations of Algorithm RingConstruct in the last step. Thus, we focus on constructing two subrings from the faulty ring.

When a new link failure occurs, the end nodes of this faulty link broadcast this information to all other nodes. Hereby, all nodes will always have the most recent fault information. Based on this fact, the two constructed subrings can be remerged as performed by Algorithm RingConstruct. After each node obtains the new fault information, the reconfiguration algorithm is invoked. The reconfiguration algorithm for constructing subrings is shown in Fig. 8.

In line 2, the current node checks if one of its ring connections is faulty. If yes, the current node will initiate the reconfiguration operations. For example, nodes *b* and *d* in Fig. 7 satisfy line 2. Then, nodes *a* and *c* must be found. Recall that $(a \oplus b) = (c \oplus d)$ and $(a \oplus c) = (b \oplus d)$. Therefore, $a = b \oplus I_j$ and $c = d \oplus I_j$ for some j. The dimension j is found by operations in line 4 through line 10. Because the dimensions are checked form 0 to $n - 1$ and the first dimension satisfying line 5 is picked, it is guaranteed that nodes *b* and *d* pick the same dimension. After nodes *a* and *c* are found, connections *a*-*c* and *b*-*d* are broken and connections *a*-*b* and *c*-*d* are established. Operations in lines 2 through 15 are for nodes *b* and *d*, while operations in line 17 through 24 are for nodes *a* and *c*.

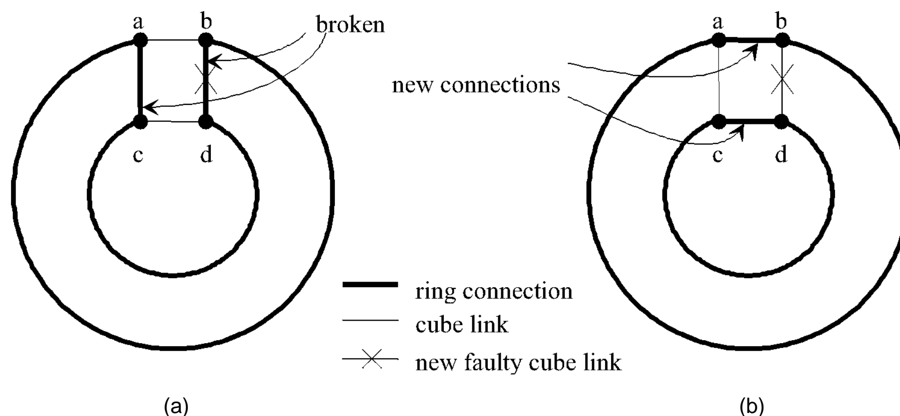

Fig. 7. Example reconfiguration. (a) Before breaking connections. (b) Two subrings are formed.

Algorithm RingReconfig;
{Each node has two ring connections in dimensions d_1 and d_2, respectively.}
1. **begin**
2. **if** d_k-link (k=1 or 2) is faulty
3. **then begin**
4. **for** $i = 0$ to $n-1$ **do begin**
5. **if** (i-link is non-faulty and $i \neq d_1$ or d_2)
6. **then begin**
7. $d \leftarrow i$;
8. Exit for-loop;
9. **endif**;
10. **endfor**;
11. $neib \leftarrow curr \oplus I_d$;
12. Send a break-d_k-connection message to node *neib* via link d;
13. Break the connection in dimension d_k;
14. Establish connection *curr-neib*;
15. **endif**
16. **else**
17. **if** a break-t-connection message is received via link l
18. **then begin**
19. **if** $d_1 = t$ **then** $d_1 \leftarrow l$
20. **else** $d_2 \leftarrow l$;
21. $neib \leftarrow curr \oplus I_l$;
22. Break the connection in dimension t;
23. Establish connection *curr-neib*;
24. **endif**;
25. **end**.

Fig. 8. Algorithm RingReconfig.

It is easy to see that Algorithm RingReconfig takes only one parallel step. Therefore, we have the following theorem.

THEOREM 5. *The complexity of Algorithm RingReconfig is* $O(1)$ *parallel step.*

Since remerging the two subrings needs another parallel step, our reconfiguration strategy requires only two (i.e., $O(1)$) parallel steps in total.

5 CONCLUSIONS

In this paper, we have proposed distributed algorithms for ring-embedding and dynamic reconfiguration in hypercubes. Algorithm RingConstruct works in faulty or fault-free hypercubes with $O(n)$ parallel steps. The number of distinct Hamiltonian cycles which can be identified in a healthy hypercube by our approach is larger than that by using GC. If the number of injured dimensions does not exceed $n-2$, Algorithm RingConstruct can tolerate up to $2^{n-1}-1$ faulty links. The conditions under which Algorithm RingConstruct works are analyzed and discussed. Therefore, Algorithm RingConstruct can be applied even when a large number of faulty links exist in the hypercube. In addition, comparisons with previous research results are given. Last, but not least, Algorithm RingReconfig takes only two parallel steps to reconfigure a faulty embedded ring and it is thus good for maintaining the integrity of an embedded ring.

One of the applications on ring-based multiprocessors is solving long-range problems [11]. Examples are molecular dynamics calculations, Newtonian gravitational n-body simulations, and the product of two large polynomials. In algorithms for solving these problems, computations must be performed on all pairs of objects. Thus, the communication patterns are ring-based. Our algorithms are devised for real application from the beginning. After executing the algorithms, the connections for a ring are formed and each

node has the information about who are its ring neighbors. Moreover, even when the fault-tolerant algorithm cannot complete the embedding successfully, the intermediate results can still be utilized for the simulation of ring-based multiprocessors. Therefore, our approach is practical and useful in terms of performance and fault tolerance capability.

ACKNOWLEDGMENT

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

REFERENCES

[1] M.Y. Chan and S.J. Lee, "Distributed Fault-Tolerant Embedding of Rings in Hypercubes," *J. Parallel and Distributed Computing*, vol. 11, no. 1, pp.63-71, Jan. 1991.

[2] F.J. Provost and R. Melhem, "Distributed Fault-Tolerant Embedding of Binary Trees and Rings in Hypercubes," *Proc. Int'l Workshop Defect and Fault Tolerance in VLSI Systems*, 1988.

[3] S. Latifi, S.Q. Zheng, and N. Bagherzadeh, "Optimal Ring Embedding in Hypercubes with Faulty Links," *Proc. IEEE 22th Int'l Symp. Fault-Tolerant Computing*, pp. 178-184, July 1992.

[4] M.S. Chen and K.G. Shin, "On Hypercube Fault-Tolerant Routing Using Global Information," *Proc. Conf. Hypercubes, Concurrent Computers, and Applications*, pp. 83-86, 1989.

[5] P.J. Yang, S.B. Tien, and C.S. Raghavendra, "Embedding of Rings and Meshes onto Faulty Hypercubes Using Free Dimensions," *IEEE Trans. Computers*, vol. 43, no. 5, pp. 608-613, May 1994.

[6] P.J. Yang, S.B. Tien, and C.S. Raghavendra, "Embedding of Multidimensional Meshes onto Faulty Hypercubes," *Proc. Int'l Conf. Parallel Processing*, pp. I571-I574, Aug. 1991.

[7] C.S. Raghavendra, P.J. Yang, and S.B. Tien, "Free Dimension—An Effective Approach to Achieving Fault Tolerance in Hypercubes," *Proc. IEEE 22th Int'l Symp. Fault-Tolerant Computing*, pp. 170-177, July 1992.

[8] M.S. Chen and K.G. Shin, "Processor Allocation in an *n*-Cube Multiprocessor Using Gray Codes," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 396-407, Dec. 1987.

[9] F. Harary, "A Survey of Hypercube Graphs," *Computer Math. Applications*, 1989.

[10] S. Park and B. Bose, "Broadcasting in Hypercubes with Link/Node Failures," *Proc. Fourth Symp. Frontiers of Massively Parallel Computation*, pp. 286-290, Oct. 1992.

[11] G. Fox et al., "Long Range Interactions," *Solving Problems on Concurrent Processors*, vol. 1, chapter 9, pp. 155-165. Prentice Hall, 1988.

[12] Y. Saad and M.H. Schultz, "Topological Properties of Hypercube," *IEEE Trans. Computers*, vol. 37, no. 7, pp. 867-872, July 1988.

[13] T.C. Lee, "Quick Recovery of Embedded Structures in Hypercube Computers," *Proc. Fifth Memory Computing Conf.*, pp. 1,426-1,435, Apr. 1990.

[14] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms*. Prentice Hall, 1977.

Fault Tolerance Properties of Pyramid Networks

Feng Cao, *Member*, *IEEE*, Ding-Zhu Du,
D. Frank Hsu, *Member*, *IEEE*, and Shang-hua Teng

Abstract—In this paper, we study the pyramid network (also called pyramid), one of the important architectures in parallel computing, network computing, and image processing. Some properties of pyramid networks are investigated. We determine the line connectivity and the fault diameters in pyramid networks. We show how to construct a path between two nodes in the faulty pyramid networks in polynomial time. A polynomial-time algorithm is also given for generating the containers in pyramid networks. Our results show that pyramid networks have very good fault tolerance properties.

Index Terms—Pyramid, fault-tolerant routing, line connectivity, parallel computing.

1 INTRODUCTION

IN the design of networks, one of the most important topics is their reliability. Connectivity and edge connectivity are widely used as measurements. In practice, we are often interested in a collection of multipaths between a pair of nodes. Some new concepts were proposed to study the collections of multipaths in graphs and networks [9].

The classical approach to studying routing in interconnection networks is to try to find the shortest path between the sending station and the receiving station. Whenever some stations are faulty on the path between the sending station and the receiving station, the management protocol has to find a way to bypass those faulty stations and set up a new path between them. Similarly, if this new path is disconnected again, a third path needs to be set up if it is possible (the network is still connected or there still exists a path between the sending station and the receiving station).

The current Internet provides end-to-end routing as above [16]. If any router on the shortest path becomes congested or goes down, the performance will be dramatically decreased due to routing overhead. IP multicasting [4] faces the same problem. The current routing protocols for IP multicasting, such as DVMRP, MOSPF, and PIM, cannot provide better services in a faulty interconnection networks.

One issue is missing in all the previous management protocols in terms of minimizing the lengths of the new paths in such a faulty interconnection network. It is not enough to only try to make the first connection as short as possible. In order to achieve better performance of communications, it is necessary to consider how to optimize the routing of alternative paths while searching for first connections in case the first connection is broken down, and so on.

This reflects one important aspect of the fault tolerance properties of the interconnection networks which has not been studied

- F. Cao is with Cisco Systems, Inc., 170 West Tasman Dr., San Jose, CA 95136-1706. E-mail: fcao@cisco.com.
- D.-Z. Du and S-h. Teng are with the Department of Computer Science, 4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455. E-mail: {dzd, steng}@cs.umn.edu.
- D.F. Hsu is with the Graduate Program in Computer Science, Fordham University, LL813, 113 W. 60th St., New York, NY 10023.

Manuscript received 19 Nov. 1996.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 102359.

0018-9340/99/$10.00 © 1999 IEEE

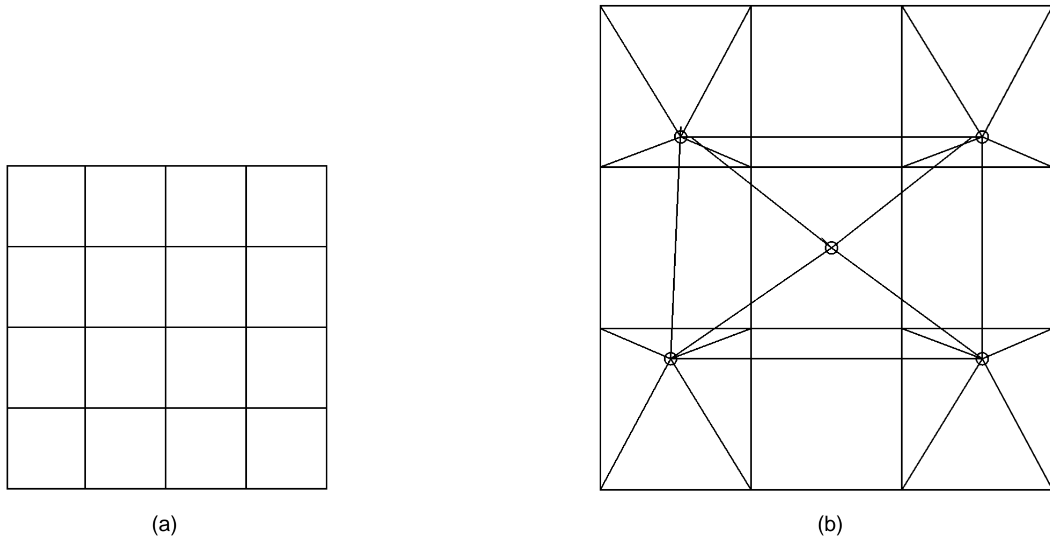

Fig. 1. Examples of mesh and pyramid network. (a) Mesh M[5, 5]. (b) Pyramid PM[2].

for most network topologies. The network researchers want to know what the effects of the faulty stations are, such as increasing the maximum delay in a faulty network. For real-time applications, such real-time control, multimedia applications, and image processing, the propagation delay must meet the time constraints. In a faulty system, connectivity only tells us if the system is still connected. But, people want to know what the longest delay in such a faulty system is and how to do efficient routing to meet the deadlines. In this paper, we answered these questions for pyramid networks.

In the following, we will introduce some concepts to study these effects quantitatively, such as *w*-distances, *fault* diameters.

Some of the important networks have been studied in terms of these concepts. For example, Hsu and Lyuu showed the exact bounds of the *w*-distance for directed toruses [10]. Butterly networks and optical passive stars networks were studied by Cao [1] and Cao et al. [2].

The pyramid network is one of the important structures in parallel and network computing [14], [18] and image processing [15]. In image processing, pyramid networks are used as both hardware architectures and software structures. In parallel and network computing, a lot of parallel algorithms are efficiently implemented on pyramid networks. For example, some parallel algorithms are implemented in supercomputers like Cray T3D and T3E and each processor acts as a node in the pyramid network. Other parallel algorithms are implemented by involving by several workstations, each workstation acting as a node in the pyramid network.

To support real-time applications, such as network computing and image processing in pyramid networks, communication among the nodes are very essential. The propagation delay must meet the time constraints. This becomes even more important in a faulty pyramid network for the longest delay in such a faulty environment determines if the pyramid network is suitable to support real-time applications. An answer also needs to be found as to how to efficiently do routing between any working nodes in such a faulty pyramid network.

In this paper, we study the fault tolerance properties of pyramid networks quantitatively in terms of the new concepts such as fault diameter and w-wide diameter, which will be defined in the rest of this section.

A pyramid network (also called *a pyramid*) is a hierarchy structure based on meshes. *A mesh*, $M[a, b]$, is a set of nodes $V(M[a, b]) = \{(x, y) \mid 1 \le x \le a, 1 \le y \le b\}$ and two nodes, (x_1, y_1) and (x_2, y_2), are connected by an edge iff $|x_1 - x_2| + |y_1 - y_2| = 1$.

A pyramid network of n levels, denoted by $PM[n]$, is a set of nodes $V(PM[n]) = \{(k; x, y) \mid 0 \le k \le n, 1 \le x, y \le 2^k\}$. A node, $(k; x, y) \in V(PM[n])$, is said to be a node at level *k*. All the nodes in level *k* form a mesh $M[2^k, 2^k]$. $(k; x, y) \in V(PM[n])$ is also connected to $(k + 1; 2x - 1, 2y)$, $(k + 1; 2x, 2y - 1)$, $(k + 1; 2x - 1, 2y - 1)$, and $(k + 1; 2x, 2y)$ for $0 \le k < n$. The node $(k - 1; x, y)$ is said to be $P((k; x_1, y_1))$ if $(k; x_1, y_1)$ and $(k - 1; x, y)$ are connected by an edge. The *root* of $PM[n]$ is $(0; 1, 1)$. The node $(k; x_2, y_2)$ is said to be *a neighboring node* of $(k; x_1, y_1)$ iff they are connected by an edge in level *k* and $P((k; x_2, y_2)) = P((k; x_1, y_1))$. It is easy to see that the diameter of PM[n] is $2n$ (see Fig. 1).

In this paper, we use *graph* and *network* interchangeably. *A network* or *graph* G = (V, E) consists of a set of nodes and a set of ordered pairs of nodes called edges. Denote by $d(G)$ its diameter and by $k(G)$ its connectivity. *A container* $C(x, y)$ between two distinct nodes *x* and *y* in G is a set of node-disjoint paths between *x* and *y*. The width of $C(x, y)$, written as $w(C(x, y))$, is its cardinality. The length of $C(x, y)$, written as $l(C(x, y))$, is the length of the longest path in $C(x, y)$. For example,

$$C(A, B) = \{A \to C \to B, A \to D \to E \to B\}$$

is a container, $w(C(A, B)) = 2$ and $l(C(A, B)) = 3$.

The *w-wide distance* between *x* and *y*, denoted by $d_w(x, y)$, is $l(C(x, y))$, where $C(x, y)$ is the minimum length container between *x* and *y* with width *w*. The *w-wide diameter* of G, written as $d_w(G)$, is the maximum of w-wide distance among all pairs of distinct nodes.

The fault diameter between two nodes, *x* and *y*, in G and *the fault diameter of G* are defined below:

$$D_w(x, y) = \max_{|S| \le w-1} \{d(x, y) \text{ in } G - S : x \ne y\}$$

$$D_w(G) = \max_{|S| \le w-1} \{d(G - S)\}.$$

The w-wide distance and w-wide diameter are defined by Hsu [9]. The notation of $D_w(G)$ was first defined by Krithnamoorthy and Krishnamurthy [13]. By the definition, we have $D_w(x, y) \le d_w(x, y)$ and $D_w(G) \le d_w(G)$.

For a general network *G*, the computation for $d_w(G)$ is NP-hard [9].

This paper is organized as follows: In Section 2, we give the exact bounds of the line connectivities. In Section 3, we study fault diameters in pyramid networks $PM[n]$ with $n \ge 1$. In Section 4, we design polynomial-time algorithms for generating the containers of pyramid networks $PM[n]$ with $n \ge 1$. Based on these algorithms, we give the estimation of the *w*-wide distances. The conclusion is made in Section 5.

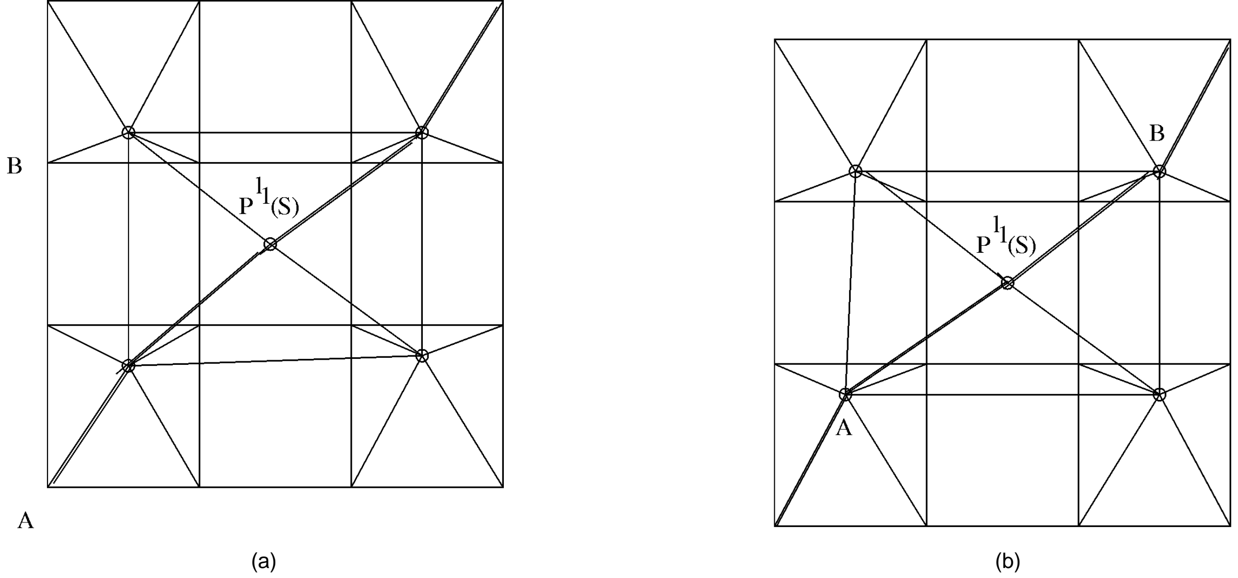

Fig. 2. A and B are two faulty nodes.

2 Line Connectivity

In this section, we show that the line connectivity of $PM[n]$ is three. To motivate our proof, we first give a lemma about the line connectivity of the mesh graph. The line connectivity of a connected network is defined to be the maximum number of links whose removal doesn't affect the connectness of the network. It is easy to see the line connectivity of a mesh.

LEMMA 2.1. *The line connectivity of* $M[a, b]$ *with* $a, b \geq 2$ *is two.*

THEOREM 2.2. *The line connectivity of* $PM[n]$ *is three.*

PROOF. If $n = 1$, it is easy to see that the line connectivity of $PM[1]$ is three. Assume that $n \geq 2$. Define $L_i = \{(k; x, y) \mid k = i, 0 < x, y < 2^k\}$ for $0 \leq i \leq n$. $V(PM[n]) = \bigcup_{i=0}^{n} L_i$.

To prove by contradiction, we assume that there are $A, B \neq \emptyset$ such that $V(PM[n]) = A \cup B$ and there are at most two edges between A and B. Since the line connectivity of $PM[1]$ is three, $L_0 \cup L_1 \subseteq A$ or $L_0 \cup L_1 \subseteq B$.

Claim that $L_i \subseteq A$ or $L_i \subseteq B$ for $2 \leq i \leq n$.

Suppose this is not the case. Then, there exists i such that $L_i \cap A \neq \emptyset$ and $L_i \cap B \neq \emptyset$. There are at least two edges in level i between $L_i \cap A$ and $L_i \cap B$. But, any node in both $L_i \cap A$ and $L_i \cap B$ is connected to the root by a path which intersects with L_i only at the node itself. That means there must exist the third edge between A and B, which contradicts with our assumption.

Thus, there exists a subset I of $\{0, 1, \ldots, n\}$, $A = \bigcup_{i \in I} L_i$ and $B = \bigcup_{i \in \bar{I}} L_i$. There must exist $i \in I$, $L_i \subseteq A$ and $L_{i+1} \subseteq B$ or $L_{i-1} \subseteq B$. But, the edges between any two consecutive levels are at least four. This contradicts our assumption. Because the degree of $(n; 2^n, 2^n)$ is only three, the line connectivity of $PM[n]$ is three. □

For the node connectivity, the following lemma can be proven by a simple induction on n for $PM[n]$:

LEMMA 2.3. *The node connectivity of* $PM[n]$ *is three.*

3 Fault Diameter

In this section, we give a constructive algorithm to find a path between two nodes in $PM[n]$ in which there are two faulty nodes. Th objective is to minimize the length of such a path. We first give a lower bound on the fault diameters of $PM[n]$.

LEMMA 3.1. $D_i(PM[n]) \geq 2n + 2i - 2$ *for* $i = 2, 3$ *and* $n \geq 3$.

PROOF. We just show $D_3(PM[n]) \geq 2n + 4$. The proof for $D_2(PM[n]) \geq 2n + 2$ is similar.

Let $S = (n; 1, 1)$, $T = (n; 2^n, 2^n)$, $A = (n-1; 1, 1)$, and $B = (n-1; 2^{n-1}, 2^{n-1})$. Consider the shortest distance between S and T in $PM[n] - \{A, B\}$, i.e., A and B are faulty nodes.

Let l be a path between S and T in $PM[n] - \{A, B\}$. If $(k; x, y) \in l$, we add $P((k; x, y))$ and delete all the nodes in l in level k except the first node in level k which is the nearest to S in l and the first node in level k which is the nearest to T in l for $k \leq n-1$. We get a new path l' and the length of l' is no more than that of l.

Thus, a shortest path between S and T is

$$\begin{array}{ll} S \quad \to (n;\ 1,\ 2) & \to (n;\ 1;\ 3) \\ \to (n-1;\ 1,\ 2) & \to (n-2;\ 1,\ 1) \\ \to \ldots & \to (0;\ 1,\ 1) \\ \to (1;\ 2,\ 2) & \to \ldots \\ \to (n-2;\ 2^{n-2},\ 2^{n-2}) & \to (n-1;\ 2^{n-1},\ 2^{n-1}-1) \\ \to (n;\ 2^n,\ 2^{n-2}) & \to (n;\ 2^n,\ 2^n-1) \\ \to T & \end{array}$$

By our construction, A and B are not on this path. The length of this path is $3 + 2(n-1) + 3 = 2n + 4$. We have $D_3(S, T) \geq 2n + 4$. Thus, $D_3(PM[n]) \geq 2n + 4$. □

THEOREM 3.2. $D_i(PM[n]) = 2n + 2i - 2$ *for* $i = 2, 3$ *and* $n \geq 3$. *Moreover,*

$$D_2(PM[2]) = 5,\ D_3(PM[2]) = 6,\ D_2(PM[1]) = D_3(PM[1]) = 2.$$

PROOF. We first show $D_3(PM[n]) \leq 2n + 4$ for $n \geq 3$. The proof for $D_2(PM[n]) \leq 2n + 2$ for $n \geq 3$ is similar.

$\forall S = (k_1; i_1, j_1)$, $T = (k_2; i_2, j_2) \in PM[n]$, define

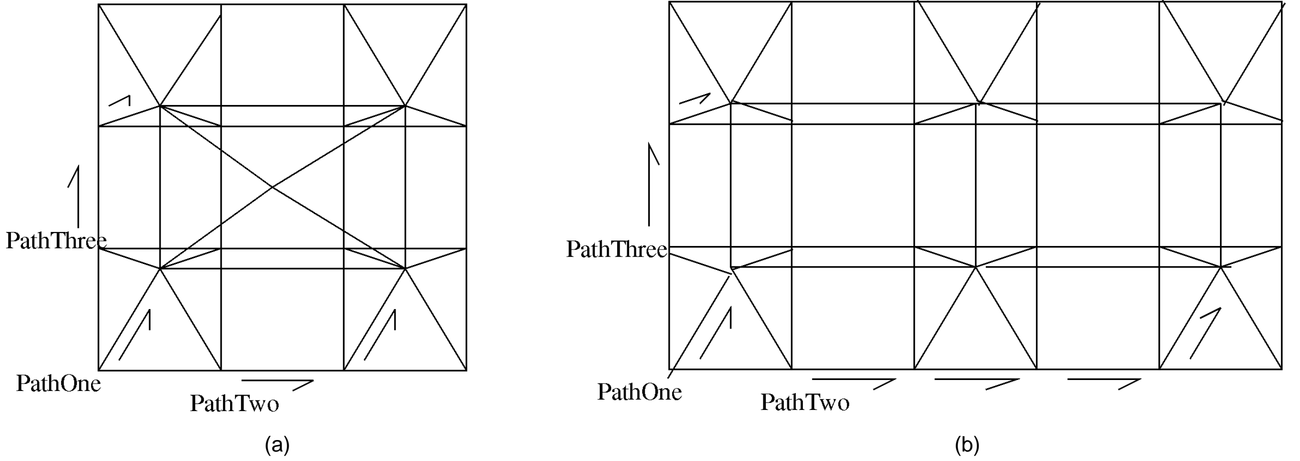

Fig. 3. Step 1 and part of Step 2.

$$l_1 = \min\{l \mid \exists k, P^l(S) = P^k(T)\}$$
$$l_2 = \min\{l \mid \exists k, P^k(S) = P^l(T)\}.$$

Consider the path L:

$$S \to P(S) \to \dots \to P^{l_1}(S) \to P^{l_2-1}(T) \to \dots \to P(T) \to T.$$

The length of L is l_1+l_2.

Let A and B be two faulty nodes in $PM[n]$. Without loss of generality, assume that at least one of A and B, say A, is on L. Otherwise, L is the path in $PM[n] - \{A, B\}$ and its length is no more than $2n$.

Case 1: $A \in L$ and B is not in L (see Fig. 2a).

Without loss of generality, assume $A = P^k(S)$. If $k = l_1$, $P^{l_1-1}(S)$ and $P^{l_2-1}(T)$ are two nodes in a square in the same level, we can find two node-disjoint paths between $P^{l_1-1}(S)$ and $P^{l_2-1}(T)$ in that level with the longest one no more than three. Since only A and B are faulty, we can find a new path L' with length no more than $l_1 + l_2 + 1$.

If $k < l_1$, there are two node-disjoint paths which does not traverse A beween $P^{k-1}(S)$ and $P^{k+1}(S)$ with length four. Since only A and B are faulty, we can find a new path L' with length no more than $l_1 + l_2 + 2$.

Thus, the new path L' is no more than $2n + 2$.

Case 2: $A \in L$ and $B \in L$. By the symmetric property, we only need to consider the following three cases (see Fig. 2b).

Subcase 2.1: $A = P^{l_1}(S)$ and $B = P^{l_1-1}(S)$. There is a path which does not traverse A and B between $P^{l_1-2}(S)$ and $P^{l_2-1}(T)$ with length at most five. Since only A and B are faulty, we can find a new path L' with length no more than $l_1 + l_2 + 2$.

Subcase 2.2: $A = P^{l_1}(S)$ and $B = P^{l_1-k}(S)$ with $k > 1$. There is a path which does not traverse A and B between $P^{l_1-1}(S)$ and $P^{l_2-1}(T)$ with length at most two, and a path which does not traverse A and B between $P^{l_1-k_1-1}(S)$ and $P^{l_1-k_1+1}(S)$ with length at most four. Since only A and B are faulty, we can find a new path L' with length no more than $l_1 + l_2 + 2$.

Subcase 2.3: $A = P^{l_1-k_1}(S)$ and $B = P^{l_2-k_2}(T)$ with $k_1, k_2 \geq 1$. There is a path which does not traverse A and B between $P^{l_1-k_1-1}(S)$ and $P^{l_1-k_1+1}(S)$ with length at most four, and a path which does not traverse A and B between $P^{l_2-k_2-1}(T)$ and $P^{l_2-k_2+1}(T)$ with length at most four. Since only A and B are faulty, we can find a new path L' with length no more than $l_1 + l_2 + 4$.

By simple case analysis, we can show it is true for $PM[2]$ and $PM[3]$.

Thus, the new path L' is no more than $2n + 4$. □

The proof for the above theorem gives a constructive algorithm for routing a path between any two nodes in a pyramid network with two faulty nodes.

4 Containers

Lemma 4.1. *If there are two node-disjoint paths, l_1 and l_2, between $S = (k_1; x, x)$ and $T = (k_2; y, y)$, then there are two node-disjoint paths, l_1' and l_2', between S and T, satisfying*

1) *the length of l_i' is equal to the length of l_i for $i = 1, 2$;*
2) $\forall A = (k; x, y) \in l_1', x \leq y; \forall B = (m; z, w) \in l_2', z \geq w.$

Proof. For any node $A = (k; x, y) \in l_1$ and $x > y$, replace A by $(k; y, x)$. Connecting the nodes in the original order, we get a new path l_1' between S and T. For any node $B = (m; z, w) \in l_2$ and $z < w$, replace B by $(m; w, z)$. We also get a new path l_2' between S and T.

It is easy to see that l_1' and l_2' are what we want. □

Lemma 4.2. *The sum of lengths of two node-disjoint paths between $S = (n; 1, 1)$ and $T = (n; 2^n, 2^n)$ is at least $6n - 2$ in $PM[n]$.*

Proof. It is easy to verify it is true for $n = 1, 2$.

Suppose it is true for $PM[n-1]$ and the sum of lengths of two node-disjoint paths, l_1 and l_2, between $S = (n; 1, 1)$ and $T = (n; 2^n, 2^n)$ is less than $6n - 2$ in $PM[n]$. Let $S' = (n-1; 1, 1)$ and $T' = (n-1; 2^{n-1}, 2^{n-1})$. By the previous lemma, we assume that, for any node $A = (k; x, y) \in l_1$, $x \leq y$; for any node $B = (m; z, w) \in l_2$, $z \geq w$.

Let $A = (n-1; c, d) \in l_1$ be the node nearest to S in l_1, reroute a path from A to S' in the level n-1 with the node $C = (n-1; s, t)$ satisfying $x < y$. We get a new path l_1' between S' and T' in $PM[n-1]$. Similarly, let $B = (m; p, q) \in l_2$ be the node nearest to T, reroute a path from B to S' in the level n-1 with the node $C = (n-1; s, t)$ satisfying $x > y$. We get a new path l_2' between S' and T' in $PM[n-1]$. Because of the choice of l_1 and l_2, l_1' and l_2' are also node-disjoint.

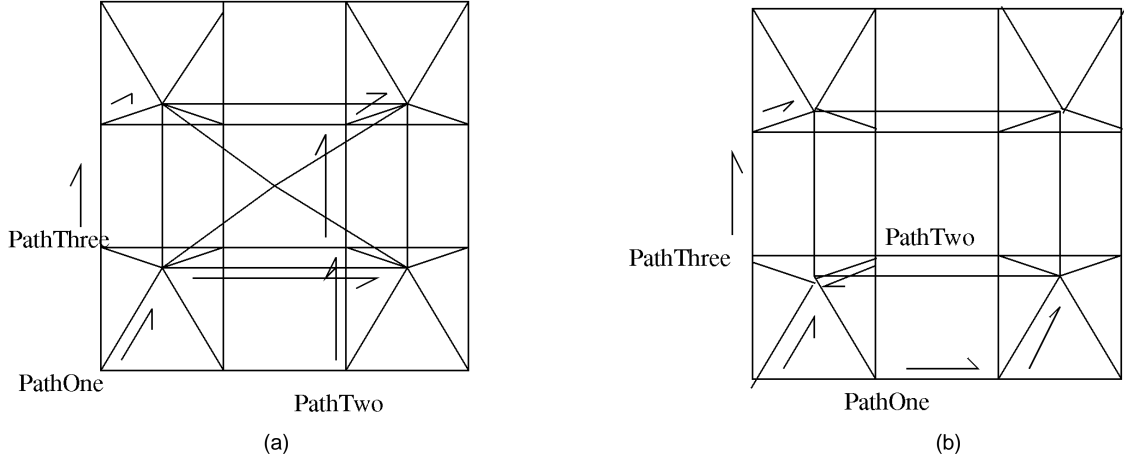

Fig. 4. Part of Step 2 and part of Step 3.

Since l_1 and l_2 are node-disjoint, the sum of the lengths of l_1' and l_2' is at least (1 + 2 + 1 + 2) less than the sum of the lengths of l_1 and l_2.

Thus, we get two node-disjoint paths l_1' and l_2' between S' and T' in $PM[n-1]$, and the sum of their length is at most $6n - 3 - 6 = 6(n-1) - 3$. This contradicts the hypothesis. □

By this lemma, we have

COROLLARY 4.3. $d_2(PM[n]) \geq 3n - 1$.

THEOREM 4.4. $d_2(PM[n]) = 3n - 1$.

PROOF. We only need to show that $d_2(PM[n]) \leq 3n - 1$.

Based on induction on n. It is true for $n = 1, 2$.

Suppose it is true for $PM[n-1]$. Let $A = (k_1; x, y)$ and $B = (k_2; s, t)$ be two nodes in $PM[n]$. If both k_1 and k_2 are less than n, there exist two node-disjoint paths between A and B in $PM[n-1]$ with length less than $3(n-1)$. Assume $k_1 = n$. Define

$$s_1 = min\{l \mid \exists k, P^l(A) = P^k(B)\}$$
$$s_2 = min\{l \mid \exists k, P^k(A) = P^l(B)\}.$$

If $n > k_2 \geq 0$ and $s_1 = 1$, it is easy to see that there are two paths between A and B with length 2. If $n > k_2 \geq 0$ and $s_1 \geq 2$, there are two node-disjoint paths, l_1 and l_2, between B and $P(A)$. We can assume that there exists a neighboring node of $P(A)$ in l_2. For l_1, we add the edge between A and $P(A)$ to form a new path l_1' between A and B. For l_2, we can add one edge from the neighboring node of P(A) to the node in level n and two other edges in level n to form another new path l_2' between A and B. By our construction, l_1' and l_2' are node-disjoint. The lengths of l_1' and l_2' are at most $3(n-1) + 2 = 3n - 1$.

If $k_2 = n$ and $P(A) = P(B)$, it is easy to see it is true. If $k_2 = n$ and $P(A) \neq P(B)$, there exist two node-disjoint paths, l_1 and l_2, between $P(A)$ and $P(B)$ in $PM[n-1]$ with lengths no more than $3(n-1) - 1$. If there exists a neighboring node of $P(A)$ in l_2 and a neighboring node of $P(B)$ in l_1, we add the edge between A and $P(A)$, the edge connecting the neighboring node of $P(B)$ in l_1 to level n, and two other edges to form a new path l_1' between A and B. Similarly, we construct another new path l_2' between A and B. l_1' and l_2' are node-disjoint. The lengths of l_1' and l_2' are at most $3(n-1) + 3 = 3n - 1$.

If there are no neighboring nodes of $P(A)$ and $P(B)$ in l_1, $P^2(A)$ is in l_1 and there is a neighboring node of P(A) and a neighboring node of $P(B)$ in l_2. We can assume that there is only one neighboring node of $P(A)$ in l_2. So, we add the edge between B and $P(B)$, the edge between $P^2(A)$ to the other neighboring node of $P(A)$, and tree edges from the other neighboring node of $P(A)$ to A to form a new path l_1' between A and B. We also add the edge between A and $P(A)$ and three edges connecting a neighboring node of $P(B)$ to B to form another new path l_2' between A and B. l_1' and l_2' are node-disjoint. The lengths of l_1' and l_2' are at most $3(n-1) + 3 = 3n - 1$.

Thus, it is true for $PM[n]$. □

From the proof of the above theorem, we actually give a recursive polynomial-time algorithm for constructing a container between any two nodes in $PM[n]$ with width two and length no more than $3n - 1$.

COROLLARY 4.5. *A recursive* $O(n^2)$ *algorithm is given for constructing a container between any two nodes in* $PM[n]$ *with width two and length no more than* $3n - 1$.

In the following, we give an algorithm for constructing a container between any two nodes in $PM[n]$ with width three and length no more than $\frac{10}{3}n + 2$.

Constructing a container between a pair of nodes in $PM[n]$ depends on the relative positions of the two nodes, the conditions such as two nodes are already connected by an edge or they share the same parent node can make the construction easier. But in most cases, it is not obvious. We provide a general algorithm for all different relative positions between the two nodes. Let $A = (k_1; x, y)$ and $B = (k_2; x, y)$ be two nodes in $PM[n]$. Define

$$l_1 = min\{l \mid \exists k, P^l(A) = P^k(B)\}$$
$$l_2 = min\{l \mid \exists k, P^k(A) = P^l(B)\}.$$

If $l_1 = 3k_1 + 2$ and $l_2 = 3k_2 + 2$:

Step 1: (See Fig. 3a)
for $i = 0$ to $k_1 - 2$ do
PathOne: the edge between $P^i(A)$ and $P^{i+1}(A)$
PathTwo: The two edges between one neighboring of $P^i(A)$ and one neighboring node of$P^{i+1}(A)$.
PathThree: The two edges between the other neighboring node of $P^i(A)$ and the other neighboring node of $P^{i+1}(A)$.
endfor

Step 2:
For PathOne, PathTwo and PathThree, we do routing as Fig. 3b and Fig. 4a.
Step 3:
Do routing as Fig. 4b,
for $i = k_1 + 1$ to $2k_1 - 2$ do
PathOne: The two edges between one neighboring of $P^i(A)$ and one neighboring of $P^{i+1}(A)$.
PathTwo: the edge between $P^i(A)$ and $P^{i+1}(A)$
PathThree: The two edges between the other neighboring node of $P^i(A)$ and the other neighboring node of $P^{i+1}(A)$.
endfor
Step 4: It is the same as Step 2 except the changes of path names, i.e., Pathone here is PathThree in Step 2, PathTwo here is PathOne in Step 2, and PathThree here is PathTwo in Step 2.
Step 5: (same as Step 3 except the path names changes)
Do routing similar to Fig. 4b.
for $i = 2k_1 + 1$ to $3k_1$ do
PathOne: The two edges between one neighboring of $P^i(A)$ and one neighboring node of$P^{i+1}(A)$.
PathTwo: The two edges between the other neighboring node of $P^i(A)$ and the other neighboring node of $P^{i+1}(A)$.
PathThree: the edge between $P^i(A)$ and $P^{i+1}(A)$
Step 6:
Similarly, we construct three paths from B to $P^{3k_2}(B)$ and its two neighboring nodes. Since $P^{3k_2}(B)$ and $P^{3k_1}(A)$ are in the same square in the same level, we can connect the six paths into three node-disjoint paths from A to B.

For other cases of l_1 and l_2 with k_1 and k_2 no less than two, it is similar except that we can do one or two more iterations in Step 1. If k_1 and k_2 are both less than two, A and B are in a $PM[5]$. It is easy to construct a container between A and B with width three and length no more than 15. If $k_1 \geq 2$ and $k_2 < 2$, or $k_1 < 2$ and $k_2 \geq 2$, we construct three paths from A or B according to Step 1 to Step 5, and connect them into B or A.

THEOREM 4.6. $d_3(PM[n]) \leq \frac{10}{3}n + 6$.

PROOF. Consider the container obtained by the above algorithm.

If k_1 and k_2 are no less than two, for Step 1, the length of PathOne is at most $k_1 - 1$, the length of PathTwo is at most $2k_1 - 1$, the length of PathThree is at most $2k_1 - 1$. In Step 2, the length of PathOne is three, the length of PathTwo is at most seven, the length of PathThree is four. Similarly, we also count the rest parts of the three paths.

The maximum length among the three paths are at most $5(k_1 + k_2) + 24$. Thus, $d_3(A, B) \leq 5(k_1 + k_2) + 13 \leq \frac{10}{3}n + 6$.

If $k_1 \leq 2$ or $k_2 \leq 2$, similarly, the inequality also holds. □

From the above, we actually have:

COROLLARY 4.7. *An* $O(n^2)$ *algorithm is given for constructing a container between any two nodes in* $PM[n]$ *with width three and length no more than* $\frac{10}{3}n + 6$.

5 CONCLUSION

In this paper, we study the pyramid network, one of the important architectures in parallel computing, network computing, and image processing. We determine the line connectivity and the fault diameters in pyramid networks. We show how to construct a path between two nodes in the faulty pyramid networks in polynomial time. We also give a polynomial-time algorithm for generating the containers in pyramid networks. The above results show that pyramid networks have very good fault tolerance properties.

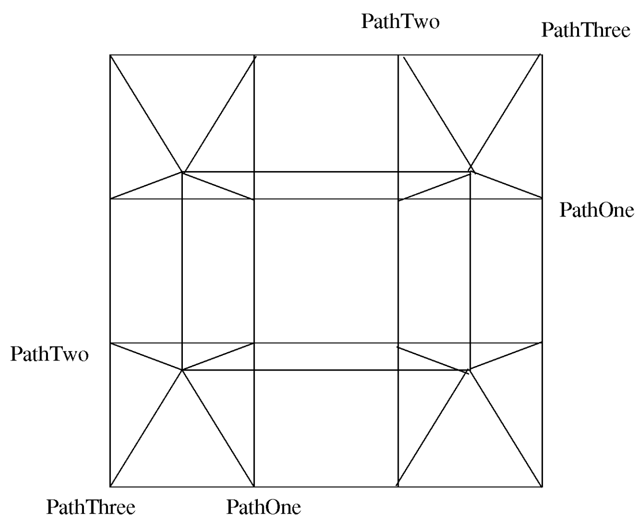

Fig. 5. Step 6.

REFERENCES

[1] F. Cao, "Reliability Analysis of Partitioned Optical Passive Stars Networks," *Proc. 22nd IEEE Local Computer Networks (LCN '97)*, Minneapolis, July 1997.
[2] F. Cao, D.Z. Du, and D.F. Hsu, "Fault-Tolerant Routing in Butterfly Networks," Technical Report TR 95-073, Dept. of Computer Science, Univ. of Minnesota, 1995.
[3] F. Cao, D.Z. Du, and D.F Hsu, "On the Fault-Tolerant Diameters and Containers of Large Bipartite Digraphs," *Proc. Int'l Symp. Combinatorial Algorithms*, Tianjin, China, June 1996.
[4] S.E. Deering, "Multicast Routing in a Datagram Internetwork," STAN-CS-92-1415, Stanford Univ., Palo Alto, Calif. **year?**
[5] D.Z. Du, D.F. Hsu, and Y.D. Lyuu, "On the Diameter Vulnerability of Kautz Digraphs," *Discrete Mathematics*, to appear. **appeared yet?**
[6] D.Z. Du and F.K. Hwang, "Generalized de Bruijn Digraphs," *Networks*, vol. 18, pp. 27-38, 1988.
[7] D.Z. Du, D.F. Hsu, F.K. Hwang, and X.M. Zhang, "The Hamiltonian Property of Generalized de Bruijn Digragh," *J. Combinatorial Theory*, Series B, vol. 52, pp. 1-8, 1991.
[8] A.H. Esfahanian and L. Hakimi, "Fault-Tolerant Routing in de Bruijn Communications Networks," *IEEE Trans. Computers*, vol. 34, pp. 777-788, 1985.
[9] D.F. Hsu, "On Container Width and Length in Graphs, Groups and Networks," *IEICE Trans. Fundamentals of Electronics, Comm., and Computer Sciences*, vol. 77A, pp. 668-680, 1994.
[10] D.F. Hsu and Y.D. Lyuu, "A Graph Theoretical Study of Transmission Delay and Fault Tolerance," *Proc. Fourth ISSM Int'l Conf. Parallel and Distributed Computing and System*, pp. 20-24, 1991.
[11] M. Imase, T. Soneoka, and K. Okada, "Fault-Tolerant Processor Interconnection," *Systems and Computers*, vol. 30, pp. 21-30, 1986.
[12] M. Imase, T. Soneoka, and K. Okada, "Connectivity of Regular Directed Graphs with Small Diameters," *IEEE Trans. Computers*, vol. 34, pp. 267-273, 1985.
[13] M.S. Krithnamoorthy and B. Krithnamurthy, "Fault Diameter of Interconnection Networks," *Computers and Mathematics with Applications*, vol. 13, no. 516, pp. 577-582, 1987.
[14] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, Calif.: Morgan Kaufmann.
[15] S. Levialdi, "A Pyramid Project Using Integrated Technology," *Integrated Technology for Parallel Image Processing*, S. Levialdi, ed. New York: Academic Press, 1985.
[16] J.B. Postel, Internet Protocol, RFC 791.
[17] D.K. Pradhan and S.M. Reddy, "A Fault-Tolerant Communication Architecture for Distributed Systems," *IEEE Trans. Computers*, vol. 31, pp. 863-870, 1982.
[18] M.J. Quinn, "Designing Efficient Algorithms for Parallel Computers," K. Hwang ed. McGraw-Hill, 1987.