A Generalized Fault-Tolerant Sorting Algorithm on a Product Network

Yuh-Shyan Chen [a], Chih-Yung Chang [b], Tsung-Hung Lin [a], Chun-Bo Kuo [a],

[a] *Department of Computer Science and Information Engineering National Chung Cheng University, Chiayi, Taiwan, R.O.C.*

[b] *Department of Computer Science and Information Engineering, Tamkang University, Taipei, Taiwan, R.O.C.*

Abstract

A product network defines a class of topologies that are very often used such as meshes, tori, and hypercubes, etc. This paper proposes a generalized algorithm for fault-tolerant parallel sorting in product networks. To tolerate $r-1$ faulty nodes, an r-dimensional product network containing faulty nodes is partitioned into a number of subgraphs such that each subgraph contains at most one fault. Our generalized sorting algorithm is divided into two steps. First, a single-fault sorting operation is presented to correctly performed on each faulty subgraph containing one fault. Second, each subgraph is considered a supernode, and a fault-tolerant multiway merging operation is presented to recursively merge two sorted subsequences into one sorted sequence. Our generalized sorting algorithm can be applied to any product network only if the factor graph of the product graph can be embedding in a ring. Further, we also show the time complexity of our sorting operations on a grid, hypercube, and Petersen cube. Performance analysis illustrates that our generalized sorting scheme is a truly efficient fault-tolerant algorithm.

Key words: Fault-tolerant, product networks, snake order, odd-even sorting, bitonic sorting.

Email addresses: yschen@cs.ccu.edu.tw (Yuh-Shyan Chen), cychang@cs.tku.edu.tw (Chih-Yung Chang), dust@cs.ccu.edu.tw (Tsung-Hung Lin), cbkuo@wmn.cs.ccu.edu.tw (Chun-Bo Kuo).

[1] A preliminary version of this paper were presented in *Proceedings of HPC 2000: 6th Int'l. Conf. on Applications of High-Performance Computers in Engineering*, Maui, Hawaii, Jan. 26-28, 2000, and was supported by the National Science Council, R.O.C., under contract no. NSC89-2213-E-216-010.

1 Introduction

A product network defines a class of topologies that are very often used. Much research on product networks has been reported in the recent literature [6][8][10][12][13]. These networks have interesting topological properties that make it especially suitable for parallel algorithms. Examples of product networks include hypercubes, grids, and tori. Many other product networks have been recently proposed, such as products of de Bruijn networks [10][18], products of Petersen graphs [14], and mesh-connected trees. A considerable amount of research has been done on product networks. Routing properties of product networks were studied in [3] and [5]. Topological and embedded properties of product networks were analyzed in [10]. Further, a reliable routing problem was proposed in [13]. Optimal fault-tolerant communication in a product network was considered in [12]. In addition, the VLSI complexity of product networks was analyzed in [9].

Many algorithms have been developed for the special case of product networks. Examples can be found in hypercubes and grids. The drawback of these algorithms is that there is no portability for different topologies. For example, a fault-tolerant sorting algorithm developed for a hypercube in [4] and [19] will not work on a grid, even though both hypercubes and grids are product networks. Recently, Fernández and Efe [8] proposed a generalized sorting algorithm for product networks. The main function of their algorithm is to propose a multiway-merging operation. However, their algorithm does not have fault-tolerant capability. The main contribution of this paper is to develop a generalized fault-tolerant sorting algorithm for product networks. Our fault-tolerant sorting algorithm is developed, which is based on Fernández and Efes' sorting algorithm [8]. The fault-tolerant sorting operation is achieved by offering a new fault-tolerant multiway-merging operation. By using this fault-tolerant multiway-merging operation, the fault-tolerant sorting algorithm is thus developed for product networks.

Our generalized sorting algorithm is divided into two steps. First, a single-fault sorting operation is presented to be correctly performed on each faulty subgraphs, each of which contains at most one fault. Second, each subgraph is considered a supernode. A fault-tolerant multiway merging operation is presented to recursively merge two sorted subsequences into one sorted sequence. Our generalized sorting algorithm can be applied to any product network under the constraint that the *factor graph* of the product graph can at least be embedding in a ring. Two basic sorting operations, odd-even and bitonic sorting operations, are used as the primitive operations. Note that using odd-even or bitonic sorting operations as primitive operations depends on the ability of embedding the factor graph of the product graph in a ring or hypercube. Let N be the number of nodes of the factor graph and

L the number of elements that each non-faulty node contains. For any r-dimensional product graph with N^r nodes, the time complexity is bounded by $O(r^2N^2L\log L)$, when using the odd-even sorting as the primitive operation. In the case when each node contains only one key ($L = 1$), the time complexity is $O(r^2N^2)$. When using bitonic sorting as the primitive operation, the time is bounded by $O(r^2L\log L(\log_2 N^2)^2 + r^2N^2 + rNL\log L(\log_2 N)^2)$. In the case when each node contains only one key ($L = 1$), the time complexity is $O(r^2(\log_2 N^2)^2 + r^2N^2 + rN(\log_2 N)^2)$. The performance study on hypercubes and Petersen cubes illustrates that the time complexity of our generalized fault-tolerant sorting algorithm is the same as that of the generalized sorting algorithm proposed by Fernández and Efes [8] when $L = 1$. Observe that Fernández and Efes' sorting algorithm [8] does not provide the fault-tolerant capability. This indicates that our proposed fault-tolerant scheme is a truly efficient algorithm.

The rest of this paper is organized as follows. In Section 2, we describe the definitions and notations used in this paper. In Section 3, we present our fault-tolerant sorting algorithm. In Section 4, the time complexity of the proposed algorithm is analyzed using several well-known product networks. The conclusions of this paper are drawn in Section 5.

2 Preliminary

In this section, we first define some notations. In Section 2.1, we formally define the product network. In Section 2.2 we define the partitioning property of a product network. Finally, we present the snake ordering method for a product network in Section 2.3.

The assumption here logically treats some processors as faulty nodes and assigns no unsorted element to them; the faulty nodes, as a result, can run idle. The fault model can be classified into two types. The most serious fault would be one that completely destroys a processor and all links incident to it. Hastad [11] called such faults *total*. A less-serious fault, named a *partial fault* [11], is one that destroys just the computational portion of a processor, leaving the communication portion of the processor as well as the incident links intact. The faults total properties can be achieved by rewriting a router to handle the fault-tolerant routing of message passing. The execution time will exceed that of the partial fault. Observe that, for simplicity, this paper assumes the partial-fault model.

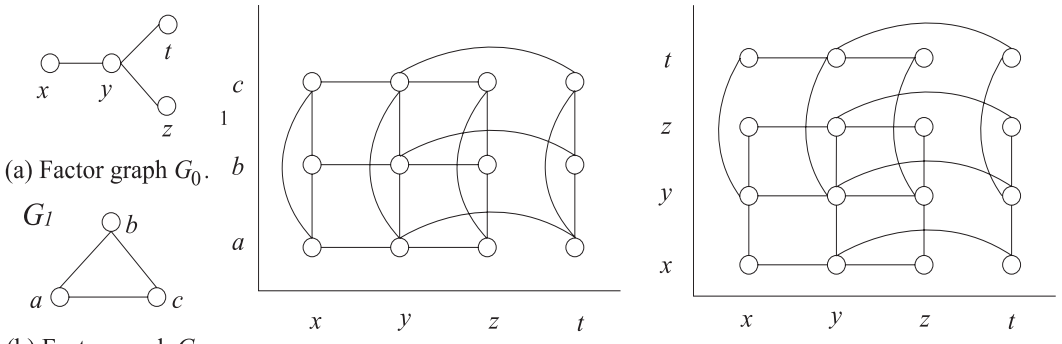

Fig. 1. Examples of product graphs.

2.1 Product Network

An interconnected network is usually modeled as an undirected graph $G = (V, E)$ with the node-set V and edge-set E. $|G|$ (or $|V|$) denotes the number of nodes in G. Let $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$ be two finite undirected graphs. The *product* of G_1 and G_0 is defined as $G = (V, E) = G_1 \times G_0$ with the node-set $V = V_1 \times V_0 = \{(x, y) \mid x \in V_1, y \in V_0\}$. There is an edge $\{(x, y), (u, v)\}$ in E if either $x = u$ and $\{y, u\} \in E_0$, or $\{x, u\} \in E_1$ and $y = v$. The graphs G_1 and G_0 are called the *factors* or component network of G. The product network G consists of $|V_0|$ copies of G_1, namely subgraphs of $G_1 y$ with the node-set$\{(x, y) \mid x \in V_1\}$ and edge-set $\{\{(x, y), (x', y)\} \mid \{x, x'\} \in E_1\}$. Analogously, G has $|V_1|$ copies xG_0 of G_0 induced by the node-set$\{(x, y) \mid y \in V_0\}$. For instance, Fig. 1(a) and Fig. 1(b) illustrate two product networks constructed by the product of G_0 and G_1 and that of G_0 and G_0, respectively.

Definition 1 *[8] The product network $G = G_1 \times G_0$ of two undirected connected graphs $G_1 = (V_1, E_1)$ and $G_0 = (V_0, E_0)$ is the undirected graph $G = (V, E)$, where V and E are given by:*

1) $V = V_1 \times V_0 = \{(x, y) \mid x \in V_1, y \in V_0\}$, and
2) an edge $\{(x, y), (u, v)\}$ in E if either $x = u$ and $\{y, u\} \in E_0$, or $\{x, u\} \in E_1$ and $y = v$.

This definition can be generalized to a product of n graphs as $G = (V, E) = G_{n-1} \times \cdots \times G_0$ where $G_i = (V_i, E_i)$, $0 \leq i \leq n-1$, such that $V = V_{n-1} \times \cdots \times V_0$; $E = \{((x_{n-1} \cdots x_0), (y_{n-1} \cdots y_0)) \mid (x_i, y_i) \in E_i$; and $x_j = y_j$, $\exists i \in \{0, \cdots, n-1\}$, for $i \neq j\}$. The value i is called the *dimension* of the edge $\{(x_{n-1}, \cdots, x_0), (y_{n-1}, \cdots, y_0)\}$. An interconnected topology derived from several factor networks by the product operation will henceforth be called a *product network*. In this paper, we consider only one-factor graphs under the

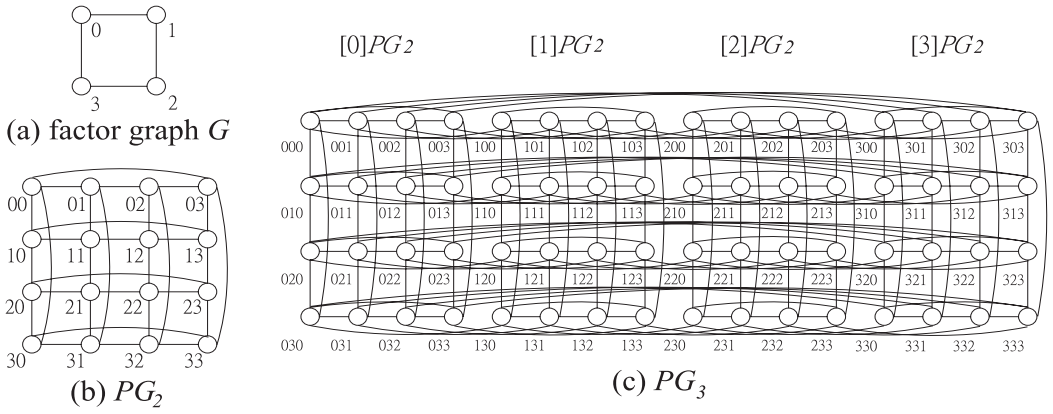

Fig. 2. Recursive construction of a multidimensional product network. (a) Factor graph; (b) two-dimensional product; (c) three-dimensional product.

self-product operation since most popular networks, such as grids, tori, and cubes, are generated by one-factor graphs. This is because the popular interconnected graphs have regular topologies and properties to design efficient parallel algorithms.

Let $PG_1 = G$. We can use the lower-dimensional product graph PG_{r-1} to construct the higher-dimensional product graph PG_r. The construction of PG_r from PG_{r-1}, where $PG_1 = G$, is shown in Fig. 2. Let x be a node of PG_{r-1}, l_x be the label of node x, and N be the number of nodes of PG_1. Symbol $[u]PG_{r-1}$ denotes the product graph obtained by putting an additional digit u before the label l_x of every vertex x in PG_{r-1}, for $u = 0, 1, \cdots, N-1$. The label l_x of every vertex $x \in PG_{r-1}$becomes ul_x. We logically describe the construction of PG_r from PG_{r-1}. First, arrange all vertices of PG_{r-1} one by one along the horizontal (or vertical) direction. Then, make N copies of PG_{r-1} along the vertical (or horizontal) direction such that vertices with identical labels fall in the same column. Next, relabel the uth copy of PG_{r-1} to obtain $[u]PG_{r-1}$, for $u = 0, 1, \cdots, N-1$. Finally, connect the corresponding nodes of $[u]PG_{r-1}$ and $[u']PG_{r-1}$ if $(u, u') \in E_G$. Fig. 2 illustrates this construction process for two- and three-dimensional product graphs. The factor graph G is shown in Fig. 2(a). Nodes in the ith row of Fig. 2(b) are labeled by putting an additional digit i before their labels. Thus, the ith row in Fig. 2(b) can be viewed as $[i]PG_1$. In a similar way, PG_3 is constructed in Fig. 2(c). Since the operations described above are logically the same as the product operation " $\times$ " defined in Definition 1, the PG_r generated by PG_{r-1} is also a product network.

To perform the fault-tolerant sorting operation on PG_r, we begin by describing the partitioning of PG_r into N copies of PG_{r-1}. The j-split operation on PG_r is defined by partitioning PG_r along dimension j into N copies of PG_{r-1}^j. Let $D = (d_{1,}d_{2,\ldots,}d_n), n < r$. The D-split on PG_r is the operation to apply d_1-split, d_2-split, ..., and d_n-split operations on PG_r. For instance, the six-dimensional hypercube is partitioned along dimensions 1, 4, and 5 by a D-split operation, where $D = (1, 4, 5)$.

Theorem 2 *We can obtain N^k copies of $PG_{r-k}^{i_1,\cdots,i_k}$ by partitioning PG_r along k dimensions $i_1, i_2, \cdots, i_k$, where $k < r$, and N is the number of nodes of the factor graph.*

The notation $[u]PG_{r-1}^i$ defines an ordering for subgraphs PG_{r-1}. In general, $[u]PG_{r-1}^i$ is the uth copy of the PG_{r-1} subgraph at dimension i. The subgraph ordering rule can be applied to the general case of $[u_1, \cdots, u_k]PG_{r-k}^{i_1,\cdots,i_k}$ in a number of different ways. We define a particular subgraph ordering method, say *snake ordering*, with certain useful properties for data sorting.

Definition 3 *[8] The snake order for the r-dimensional product graph PG_r is defined as follows:*

1) If $r = 1$, the snake order is the same as the order used for labeling the nodes of G.
2) Assume that the snake order has already been defined for PG_{r-1}, $r > 1$. Then
   (a) $[u]PG_{r-1}^r$has the same order as PG_{r-1} if u is even, and the reverse order if u is odd; and
   (b) if $u < v$ then the order of all vertices in $[u]PG_{r-1}^r$precedes the order of all vertices in $[v]PG_{r-1}^r$.

Example 4 *Let $N = 4$. The snake order sequences Q_r of product graph PG_r, for $r = 1$, 2, and 3 are listed as follows:*

- for $r = 1$, $Q_1 = \{0, 1, 2, 3\}$,
- for $r = 2$, $Q_2 = \{00, 01, 02, 03, 13, 12, 11, 10, 20, 21, 22, 23, 33, 32, 31, 30\}$,
- for $r = 3$, $Q_3 = \{$

$$000, 001, 002, 003, 013, 012, 011, 010, 020, 021, 022, 023, 033, 032, 031, 030,$$

$$130, 131, 132, 133, 123, 122, 121, 120, 110, 111, 112, 113, 103, 102, 101, 100,$$

$$200, 201, 202, 203, 213, 212, 211, 210, 220, 221, 222, 223, 233, 232, 231, 230,$$

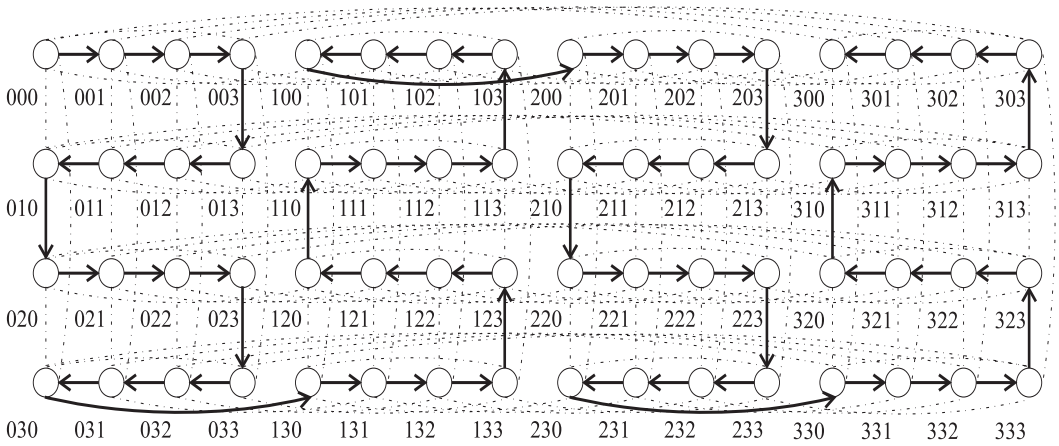

Fig. 3. The snake order of the product network PG_3 whose factor graph is a 4-node ring.

$$330, 331, 332, 333, 323, 322, 321, 320, 310, 311, 312, 313, 303, 302, 301, 300\}.$$

Fig. 3 gives the snake order for product graph PG_3 considered in Fig. 2(c). As we mentioned before, if the factor graph of a product network can be embedded in a ring, the odd-even sorting operation is used as a single-fault algorithm executed on each PG_2. This covers most cases. Further, if the factor graph can be embedded in a hypercube, then a bitonic-like sorting operation is adopted as a single-fault sorting algorithm executed on each PG_2.

3 Generalized Fault-Tolerant Sorting Algorithm

In our algorithm, a faulty product graph PG_r is partitioned into several subgraphs PG_2, where each PG_2 contains at most one faulty node. This is helpful for carrying out executing the single-fault sorting algorithm. In this section, we offer a generalized partition scheme for a faulty PG_r. The partition scheme partitions the faulty PG_r into N^{r-2} copies of PG_2 in which each PG_2 contains at most one faulty node. To tolerate one faulty node, we propose two single-fault sorting operations for each PG_2 to ensure to obtain the correct sorting order for elements on each PG_2. However, we still need to merge all elements node by node. For this purpose, we modified the well-known multiway merging operation [2] which originally had no fault-tolerant capability. By putting together the proposed single-fault sorting operation and the modified multiway merging operation as a basic operation, we developed a generalized multi-fault sorting algorithm for a faulty product network. We outline our generalized fault-tolerant sorting algorithm as follows.

To tolerate up to $r-1$ faults, we partition faulty PG_r into N^{r-2} copies of PG_2 by executing a feasible D-split operation on PG_r such that each PG_2 contains at most one faulty node. Based on a similar partition scheme in a star graph [20], we have the following property.

Lemma 5 *In a $PG_r, r \geq 4$, with $f \leq r-1$ faulty nodes, there always exists a D-split, $|D| = r-2$, such that PG_r can be partitioned into PG_2 by D-split and each partitioned PG_2 contains at most one faulty node.*

The maximum number of faults that can tolerated in this paper is $r-1$. For the condition of $f \neq r-1$, there exist some partitioned PG_2 with no faulty node. Due to the regular operation and balancing of the workload of each PG_2, we determine a *dangling node* [19] in each nonfaulty PG_2. A node is said to be a dangling node if the node is a healthy node but is assigned to no job or data [19]. Nodes in a nonfaulty PG_2 with the same position of most faulty nodes in all other faulty PG_2s will be selected as a dangling node. We logically consider the dangling node as a faulty node and assign no data to it. For example, assume that PG_3, shown in Fig. 2(c), has the faulty set $F = \{023,212\}$. A three-split operation is applied to PG_3 since the digit in dimension three of the address of faulty nodes differs. A D-split with $D = \{3\}$ will partition PG_3 with F into N copies of PG_2, while two $PG_2's$ contain sets $F_1 = \{023\}$ and $F_2 = \{212\}$. The dangling node must be determined for every healthy PG_2.

3.2 ***Distributing Unsorted Keys***

The next step is to distribute unsorted keys into all nonfaulty nodes. Assume that there are $M \gg N^r$ unsorted elements. Since the total number of nonfaulty nodes is $N^r - N^{r-2}$, each nonfaulty node contains $M/(N^r - N^{r-2}) = M/((N^2 - 1)N^{r-2})$ keys. In the next subsection, we present the execution of the single-fault sorting operation for each PG_2.

/* Fault-Tolerant Sorting Algorithm on Product Network PG_r */

Fault_Tolerant_Sorting (G, r, F, M) /* Sorts M keys on PG_r (or r-dimensional product graph G) with F faulty nodes */ {

Partition_PG_r { /* Partitions PG_r into PG_2 */

Step 1. Perform a D-split operation to partition PG_r into N^{r-2} copies of PG_2 such that each PG_2 contains at most one faulty node.

Step 2. Assign one dangling node in each healthy PG_2. The dangling node is logically considered to be a faulty node.}

Distribute_Data{ /* Distributes unsorted keys into all nonfaulty nodes */

Distribute M elements to the nonfaulty nodes.

Each nonfaulty node will thus contain L keys where

$L = M/(N^r - N^{r-2}) = M/((N^2 - 1)N^{r-2})$. }

Single_Fault_Sorting{ /* Applying a single-fault-sorting algorithm to each PG_2 */

For each PG_2,

if (G can be embedded in an n-cube structure) then

{ Step 1. Perform the processor numbering operation according to the original labeling order.

Step 2. Execute the bitonic-like sorting operation. }

else { Step 1. Perform the processor numbering operation according to the snake order.

Step 2. Execute the odd–even-like sorting operation. }}

Fault-Tolerant_Multiway_Merge_Operation{

Recursively performs our fault-tolerant multiway merging operation to merge unmerged keys from PG_i into PG_{i+1}, where $2 \leq i \leq r - 1$. } }

Fig. 4. Generalized fault-tolerant sorting algorithm for r-dimensional product network.

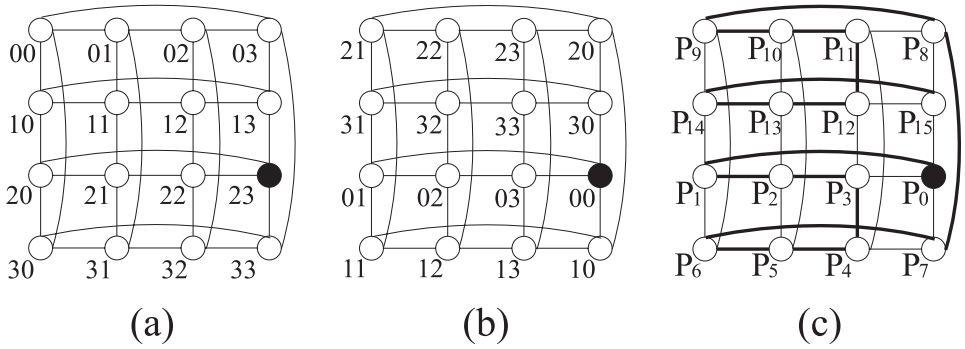

Fig. 5. GRelabeling the processor number for a sub-product network according to the snake order. (a) The original labels of graph; (b) Relabeling the faulty node to position 00; (c) Relabeling the processor number for each node according to snake order.

3.3 Single-Fault Sorting Operation

Two single-fault sorting operation algorithms are given here for PG_2 to sort M/N^{r-2} keys into ASCEND/DESCEND order. If a hypercube, where $n = \log_2 |G|$, can be embedded into factor graph G, we perform the single-fault bitonic sorting operation on each PG_2. Otherwise, we perform the single-fault odd-even sorting operation on each PG_2. For the ease of presentation, we first present the single-fault odd-even sorting operation. The single-fault bitonic sorting operation is then discussed.

Initially, a simple rotation operation is performed based on the address of the faulty node. The purpose of the rotation operation is to reset the logical address of nodes such that the logical address of the faulty node or dangling node can be considered P_0. For example, consider PG_2 containing a faulty node whose label is 23 as shown in Fig. 5(a). After performing the rotation operation, the faulty node's address is 00 as shown in Fig. 5(b). Noted that the addresses of all nodes were changed by the rotation operation shown in Fig. 5(b). According to snake order, we assign each node a processor number as shown in Fig. 5(c). The rotation operation logically makes the faulty node of each PG_2 to be P_0.

Before executing the single-fault sorting operation, all nodes in PG_2 should be assigned a processor number. If the odd-even sorting operation is determined to apply to PG_2, which is dependent on the topology of the factor graph, the processor is numbered according to the snake order. On the other hand, if the bitonic sorting operation is determined to apply to PG_2, the processor

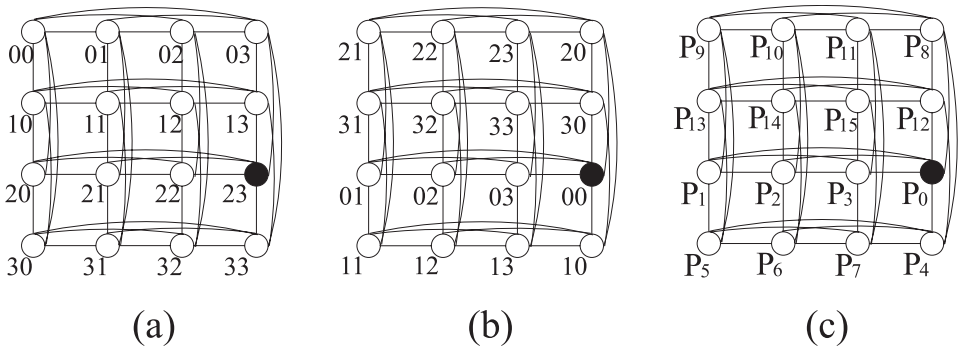

Fig. 6. Relabeling the processor number for a sub-product network according to the original order. (a) The original labels of graph; (b) Relabeling the faulty node to 00; (c) Relabeling the processor number according to the original labels' order.

is numbered according to the original labels' order. Relabeling of the original order is the same as that for the snake order. We illustrate this with an example in Fig. 6. Fig. 6(a) displays the original label sequence. In Fig. 6(b), we treat the faulty node as having the logical address of P_0. The labeling shown in Fig. 6(b) is the same as that shown in Fig. 5(b). Finally, we relabel each node with a processor number according to the original order of their labels. The processor number of each node is shown in Fig. 6(c).

3.3.1 Single-Fault Odd-Even Sorting Operation

The single-fault odd-even sorting operation consists of n comparison-exchange stages for n adjacent elements. As mentioned before, we apply the single-fault odd-even sorting operation to each PG_2 if the factor graph G can be embedded in a Hamiltonian cycle. The proposed odd-even sorting algorithm with one fault is now described as follows. First, we apply the sequential sorting algorithm, e.g., quick sorting or heap sorting, on each node for sorting its $M/((N^2-1)N^{r-2})$ elements. Then, in the odd step of the odd-even sorting algorithm, each pair of nodes, P_n and P_{n-1}, where n is odd, is compared to its sorted sequence element by element. Since P_0 is the faulty node, there is not need for either P_0 or P_1 to undergo any comparison-exchange. In the even step, each pair of nodes P_n and P_{n-1}, where n is even, is compared to its sorted sequence element by element. No node will perform the comparison-exchange with P_0. After each step of the odd-even sorting algorithm, we also need to apply the sequential sorting algorithm to each node. Because the faulty node P_0 is at the first position, we consider that the faulty node does not exist. The odd-even sorting algorithm is only performed for nodes from P_1 to P_{N^2-1}. After the odd-even sorting, the data will be kept in an ASCEND/DESCEND order from P_1 to P_{N^2-1}.

3.3.2 Single-Fault Bitonic Sorting Operation

The bitonic sorting algorithm [1][15][16][17] can also work correctly on PG_2 when the faulty node is at P_0. This result was indicated by Sheu *et al.* [19]. If the number of the factor's node is $N = 2^k$ (k is a constant) and the factor graph contains a $\log_2 N$-dimensional hypercube, then the bitonic sorting algorithm can be applied. The bitonic sorting algorithm consists of $\sum_{i=1}^{log_2 n} i$ comparison-exchange stages for n elements.

First, the M/N^{r-2} unsorted elements are uniformly distributed to $N^2 - 1$ healthy nodes. The faulty node P_0 in PG_2 is treated as a dead node. We apply the sequential sorting algorithm, e.g., quick sorting or heap sorting, on each healthy node for sorting its $M/((N^2 - 1)N^{r-2})$ elements. By applying the bitonic sorting algorithm, all M/N^{r-2} unsorted elements will be sorted at each node of PG_2 in order of their the addresses. Because the factor graph has a hypercube structure, the bitonic sorting algorithm can work correctly. The key concept of the bitonic sorting algorithm is to recursively execute the comparison-exchange operations on each pair of sorted subcubes such that the first half of the elements are located in one subcube and the last half of the elements are located in another subcube. During execution of the bitonic sorting operation, no node needs to perform any operation to P_0. One can assume that elements in PG_1 and PG_1' are now respectively sorted in ascending and descending order after executing the bitonic sorting algorithm.

Now, we have proposed two sorting algorithms which can work correctly on PG_2 when the faulty node is P_0. The odd-even sorting algorithm can be performed when the factor graph contains a ring graph. The bitonic sorting algorithm can also be performed when the number of nodes $N = 2^k$ in the factor graph, and the factor graph has a $\log_2 N$-dimension hypercube structure. However, irregardless of which of the proposed two single fault sorting operations are applied, we can only ensure that elements are sorted in order in each PG_2. In the next step, we perform the fault-tolerant multiway merging operation such that elements can be sorted among all $PG_2 s$.

3.4 Fault-Tolerant Multiway Merging Operation

Fernández and Efe proposed a generalized parallel sorting algorithm in [8]. The kernel function is the multiway merging operation [2]. Before discussing the fault-tolerant multiway merging operation, we first define a fundamental operation, namely the fault-tolerant comparison-exchange operation. Our fault-tolerant merging operation is built based on the fault-tolerant comparison-exchange operation.

3.4.1 Fault-Tolerant Comparison-Exchange Operation

Here we present the fault-tolerant comparison-exchange operation between two adjacent subgraphs, PG_i and PG'_i, where $i < r$. The main function of the fault-tolerant comparison-exchange operation is to perform the comparison-exchange operation between each pair of adjacent nodes, x and y, when $x \in PG_i$ and $y \in PG'_i$, if PG_i and PG'_i are both faulty.

The fault-tolerant comparison-exchange operation is a recursive operation. Let FCE(PG_2) denote the fault-tolerant comparison-exchange operation on any pair of copies of PG_2. We describe the fault-tolerant comparison-exchange operation as follows. Every PG_2 has exactly one faulty or dangling node. Three possible cases are discussed depending on the location of the faulty nodes: $f \in PG_2$ and $f' \in PG'_2$. A column/row of a product network is said to be a *faulty column/row* if it contains a faulty node. Based on the property of the product network, each pair of nodes, x and y, located in the same location can logically connect to each other by a path with length $\lfloor \frac{N}{2} \rfloor$, where $x \in PG_2$ and $y \in PG'_2$. Now we discuss these cases.

Case 1. *Nodes f and f' are located in the same physical location*: Each node $x \neq f$ sends its data to adjacent nodes $y \neq f'$ by a physical link and performs the comparison-exchange operation. The time complexity for sending data to adjacent node is $O(\lfloor \frac{N}{2} \rfloor)$.

Case 2. *Nodes f and f' are located in the same physical row*: Two phases are needed in this case.

1. **Data-moving phase:** Without loss of generality, let P_i in PG_2 and P'_j in PG'_2 be faulty nodes, where $i < j$. Processor sequence $P_{i+1}, P_{i+2}, \ldots, P_j$ sends data to $P'_i, P'_{i+1}, \ldots, P'_{j-1}$ by 2-hop steps as follows. Observe that this work can be correctly performed since our fault model is assumed to be the partial-fault one [19]. That is, a faulty node can still perform its communication operation, and only the computation operation is faulty. Each node P_k in $P_{i+1}, P_{i+2}, \ldots, P_j$ communicates with P'_k, and then every P'_k shifts the received data to the neighboring processor P'_{k-1}. Therefore, $P'_i, P'_{i+1}, \ldots, P'_{j-1}$acquire data. If $i > j$, a similar way can be applied. Nodes P_t not located in the faulty row send data to node P'_t with the same processor number in PG'_2. The time complexity of the data-moving phase is thus $O(\lfloor \frac{N}{2} \rfloor + 1)$. Fig. 7 illustrates this operation. Fig. 7(a) shows the processor numbering of PG_2, and Fig. 7(d) illustrates the same PG_2 with data which have been sorted in PG_2 in an ascending snake order. Fig. 7(b) illustrates the data in nodes of the first row (the row is which the faulty node is located) moving from PG_2 to PG'_2. Fig. 7(e) shows the data layout of PG'_2 after the data-moving operation is performed on each node.
2. **Rotation phase:** All nodes except the faulty node perform a rotation

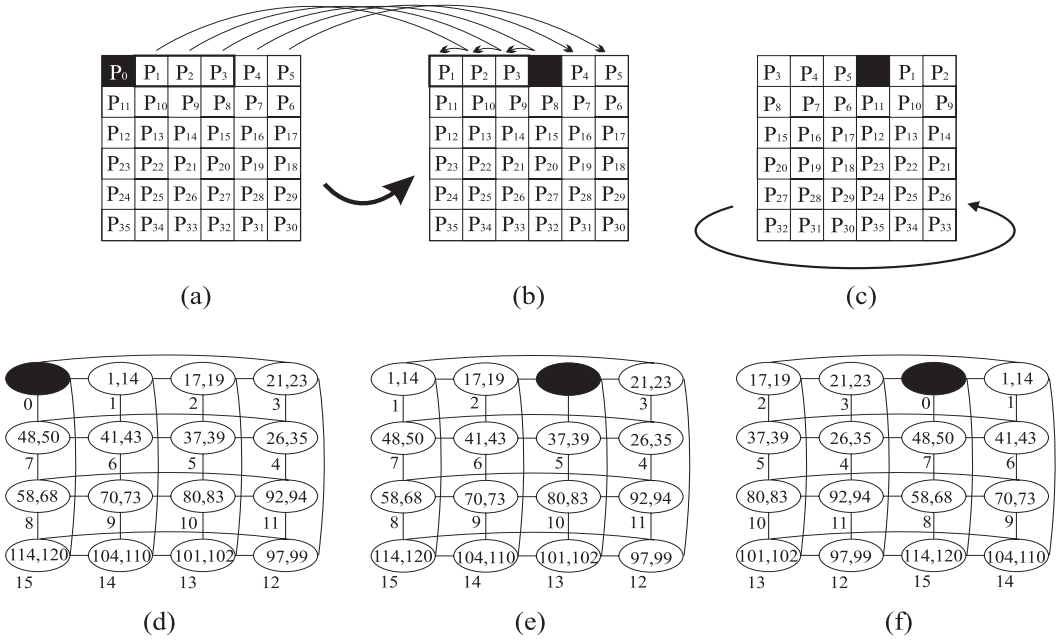

Fig. 7. The FCE(PG_2) operation executed on example of case 2 that faulty nodes of two PG_2 are located at the same row. The processor numbering is in snake order.

operation as follows. All nodes in each row repeatedly shift to the right one position until the node with the smallest processor number arrives at the position $j+1$. The time complexity of the rotation phase is then $O(N)$. Fig. 7(c) illustrates the result of the rotation phase, and Fig. 7(f) illustrates the resultant data layout of Fig. 7(c).

Case 3. *Nodes f and f' are located in different physical rows*: Initially, a similar data-moving phase as in **Case 2** is performed. The only difference is that the number of faulty rows is greater than one. Fig. 8(a) displays the processor number of PG_2, and Fig. 8(b) shows the result of the data-moving operation performed from PG_2 to PG_2'. The next task is to perform a rotation operation. This operation is divided into three phases.

1. **Horizontal-rotation phase:** Assume that the faulty node is located in the jth-column, and the row number of the first row is labeled 0. A horizontal-rotation operation is performed on each row as follows. For row numbers less than the faulty row, all nodes in the row repeatedly shift left/right one position until the node with the maximum processor number arrives in the jth-column. If the faulty row number is odd/even, all nodes in the faulty row except for the faulty node repeatedly shift left/right one position until a node with the largest/smallest address arrives in the $(j+1)$th-column. For the remaining rows, if the row number is odd/even, all nodes in the row repeatedly shift left/right one position until the node with the maximum/minimum address arrives in the jth-column. The time complexity of the horizontal-rotation phase is $O(\lfloor \frac{N}{2} \rfloor)$. Fig. 8(c) shows the result of the

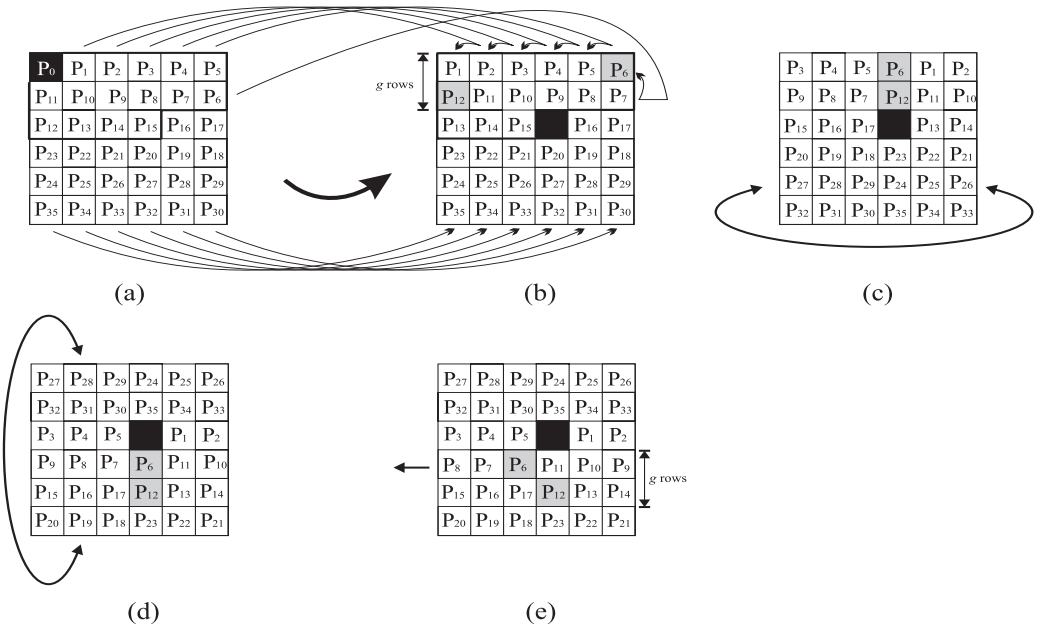

Fig. 8. TThe FCE(PG_2) opeartion executed on example of case 3 that faulty nodes of two PG_2 are not at the same row. The processor numbering is in snake order.

horizontal-rotation operation.

2. **Vertical-rotation phase:** All nodes repeatedly shift up/down one position until nodes in the first row arrive in the faulty row. If all nodes repeatedly shift up one position, then there is no need for nodes in the jth-column to shift up one position in the first step of the shift. The time complexity of the vertical-rotation phase is $O(\lfloor \frac{N}{2} \rfloor)$. An example of a vertical-rotation operation is shown in Fig. 8(d).
3. **Tuning-rotation phase:** Let g denote the gap between the first row and the faulty row. A tuning operation must be performed in the next g rows beginning from the faulty row. The task is performed as follows. Assume that the faulty row is relabeled row 0. If the row number of each row of these g rows is odd, it shifts to the left one position. The time complexity of the tuning-rotation phase is $O(1)$. An example of the tuning-rotation phase is shown in Fig. 8(e).

Lemma 6 *The FCE(PG_2) operation can be correctly executed within* $O(\lfloor \frac{3}{2}N \rfloor + 2)$ *time steps if the processor numbering sequence is in the snake order.*

Proof: The time complexity of one PG_2 sending data to another PG_2 is $O(\lfloor \frac{N}{2} \rfloor + 1)$. In **Case 1**, there is no rotation operation. The time complexities of the rotation phases in **Cases 2** and **3** are respectively $O(N)$ and $O(N+1)$. In total, the time complexity of FCE(PG_2) is $O(\lfloor \frac{3N}{2} \rfloor + 2)$.

Now we consider the other case as follows. If the factor graph has a hypercube structure, the processor numbering operation will use the original order of

each label for each node. In the following, we illustrate how the FCE(PG_2) operation is applied to PG_2 if the processors are numbered in the original order. Similar to the FCE(PG_2) operation using the snake order, we also use three cases to discuss FCE(PG_2) using the original order. Basically, the operation in **Cases 1** and **2** is the same as the operation in the snake order. In **Case 3**, the data-moving phase is the same as that in **Case 2**. Additionally, three rotation operation phases are described as follows.

1. **Horizontal-rotation phase:** Assume that the faulty node is located in the jth-column, and the row number of the first row is labeled 0. A horizontal-rotation operation is performed on each row as follows. For those rows whose row number is less than that of the faulty row, all nodes in the row repeatedly shift left/right one position until the node with the maximum address arrives in the jth-column. All nodes in the faulty row except for the faulty node repeatedly shift left/right one position until the node with the smallest address arrives in the $j+1$th-column. For the remaining rows, all nodes in the row repeatedly shift left/right one position until the node with the minimum address arrives in the jth-column. The time complexity of the horizontal-rotation phase is $O(\lfloor \frac{N}{2} \rfloor)$.
2. **Vertical-rotation phase:** All nodes repeatedly shift up/down one position until the first row arrives in the faulty row. The time complexity of the vertical-rotation phase is $O(\lfloor \frac{N}{2} \rfloor)$.
3. **Tuning-rotation phase:** The tuning rotation phase is the same as that for the snake order. The time complexity of the tuning-rotation phase is $O(1)$.

Lemma 7 *The FCE(PG_2) operation can be correctly executed on PG_2, and its time complexity is bounded by $O(\lfloor \frac{3N}{2} \rfloor + 2)$ if the processor numbering is in the original order.*

The following theorem is derived based on results of Lemmas 2 and 3.

Theorem 8 *The FCE(PG_k) operation can be correctly executed in $O(\lfloor \frac{3N}{2} \rfloor + 2)$ time steps if the processor numbering uses the snake order or the original order, where $k < r$.*

Proof: Recall that the FCE(PG_2) operation can work correctly on PG_2. For the purpose of making this operation correctly run on PG_k, we partition PG_k and PG'_k into a number of PG_2s. Then each PG_2 in PG_k performs FCE(PG_2) with the corresponding PG_2 in PG'_k. Finally, merge PG_2 in PG_k (or PG'_k) into PG_k (or PG'_k). So FCE(PG_k) can also be correctly executed in $O(\lfloor \frac{3N}{2} \rfloor + 2)$ time steps. Fig. 9 illustrates this operation.

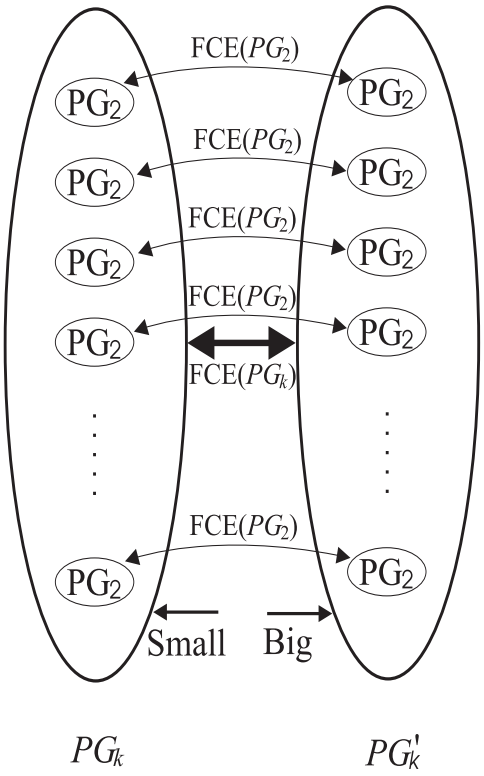

Fig. 9. The FCE(PG_k) operation.

3.4.2 The Fault-Tolerant Multiway Merging Operation

The multiway merging operation was originally used by Fernández and Efe [8] to perform a generalized sorting algorithm on a product network. The correctness can be verified by referring to [8]. However, their multiway merging operation does not have fault-tolerant capability. We present a fault-tolerant multiway merging operation here. By using the proposed fault-tolerant multiway merging operation as a basic operation, we thus develop a generalized fault-tolerant sorting algorithm. The fault-tolerant multiway merging operation is divided into four phases. The proposed multiway merging operation is a recursive algorithm. For ease of presentation, a dimension variable k, $2 < k < r$, is used to denote the current dimension in the recursive process.

We define the terms of a virtual PG_2 and a virtual PG_2 sequence as follows. The *virtual* PG_2 consists of N copies of PG_1. Any two $PG_1 s$ in the virtual PG_2 may not be directly connected. The structure of the virtual PG_2 is similar to that of PG_2 except that communication of each pair of neighboring $PG_1 s$ may require more than one step since a direct link might not exist between them. The *virtual* PG_2 *sequence* is the sequence of a number of virtual $PG_2 s$. Examples of a virtual PG_2 and a virtual PG_2 sequence are shown in Figs. 10(a) and 10(b).

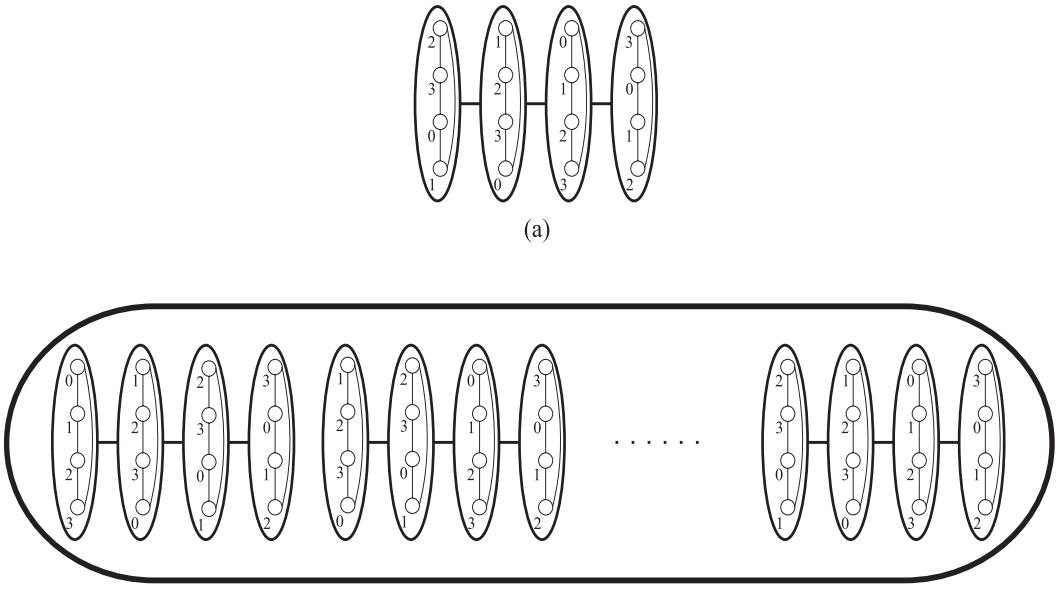

Fig. 10. (a) The virutual PG_2; (b) the virtual PG_2 sequence.

Now we present the fault-tolerant multiway merging operation here.

Fault-Tolerant_Multiway_Merge_Operation (PG_k)

{ /* The PG_k is partitioned into N copies of PG_{k-1} by applying a j-split operation, where $j \in D = (d_1, d_{2,\ldots,} d_n)$. */

1. **Redistribution step:** Basically, the redistribution process is the same as the function of the redistribution phase in [8] except for the operation of $k = 2$. The goal of the redistribution step is to collect unmerged data from different dimensions. In the case of $k = 2$, we are not only collecting unmerged data from different dimensions, but also collecting data from all of the faulty columns of every original PG_2 to organize a virtual PG_2. All the remaining virtual PG_2s are constructed according to order of the faulty columns. For example, in Fig. 11(a), the data layout shows that a PG_3 has been partitioned into four PG_2s and a single-fault sorting operation has been performed on each PG_2. We partition each PG_2 into N copies of PG_1, and collect PG_1 in each PG_2 into a virtual PG_2 as shown in Fig. 11(b).

2. **Merging step:** If $3 \leq k < r$, for each PG_{k-1} among N copies of PG_{k-1}, perform the **Fault-Tolerant_Multiway_Merge_Operation** (PG_{k-1}). Note that if $k = 2$, single-fault sorting and FCE(PG_2) are performed on the virtual PG_2 sequence. Fig. 12(a) illustrates the merging operation of PG_k when $3 \leq k < r$. Fig. 12(b) displays the construction of a number of virtual

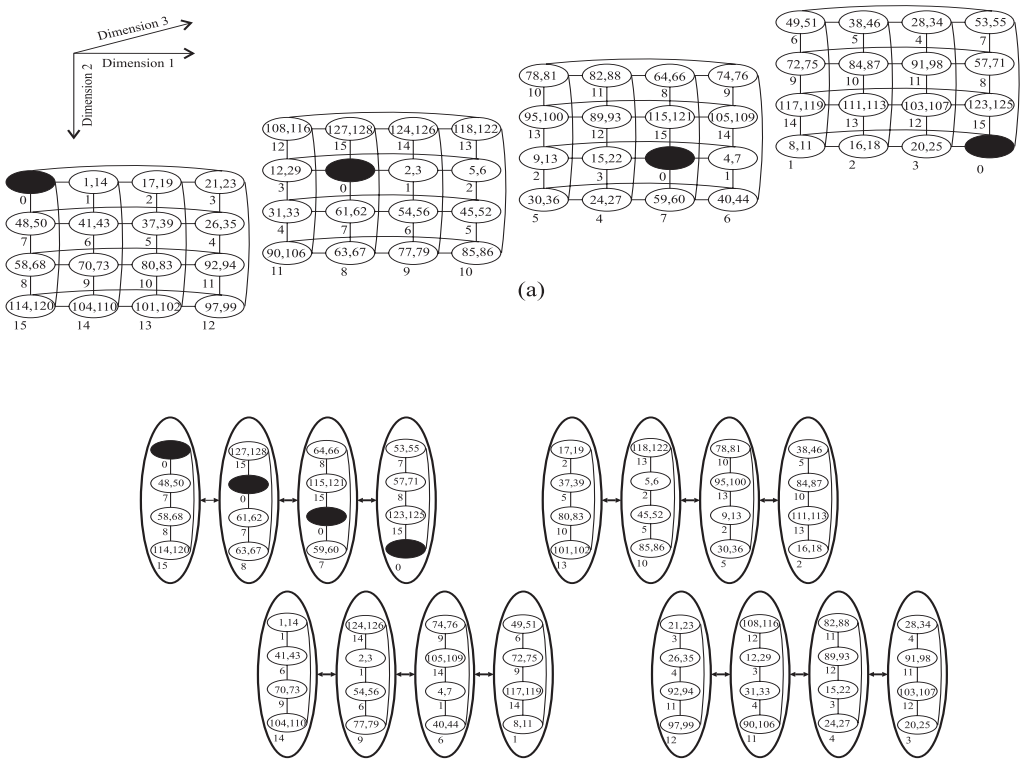

Fig. 11. Redistribution step. (a) Each PG_2 performing a single-fault sorting operation; (b) Construction of virtual PG_2 from PG_2.

PG_2s when $k = 2$. Fig. 12(c) displays the virtual PG_2 after executing the single-fault sorting and FCE(PG_2).

3. **Interleaving step:** This task is a restoration operation which is opposite to the **Redistribution step**. After executing this operation on PG_2 as shown in Fig. 13(a), we have the result shown in Fig. 13(b).

4. **Clear-dirty step:** There are three parts of the clear-dirty step. (1) For each PG_2, sort its keys. (2) Perform two odd-even transpositions among the PG_2 sequences. (3) For each PG_2, sort the keys again. For the correctness of the clear-dirty step, refer to [8].

}

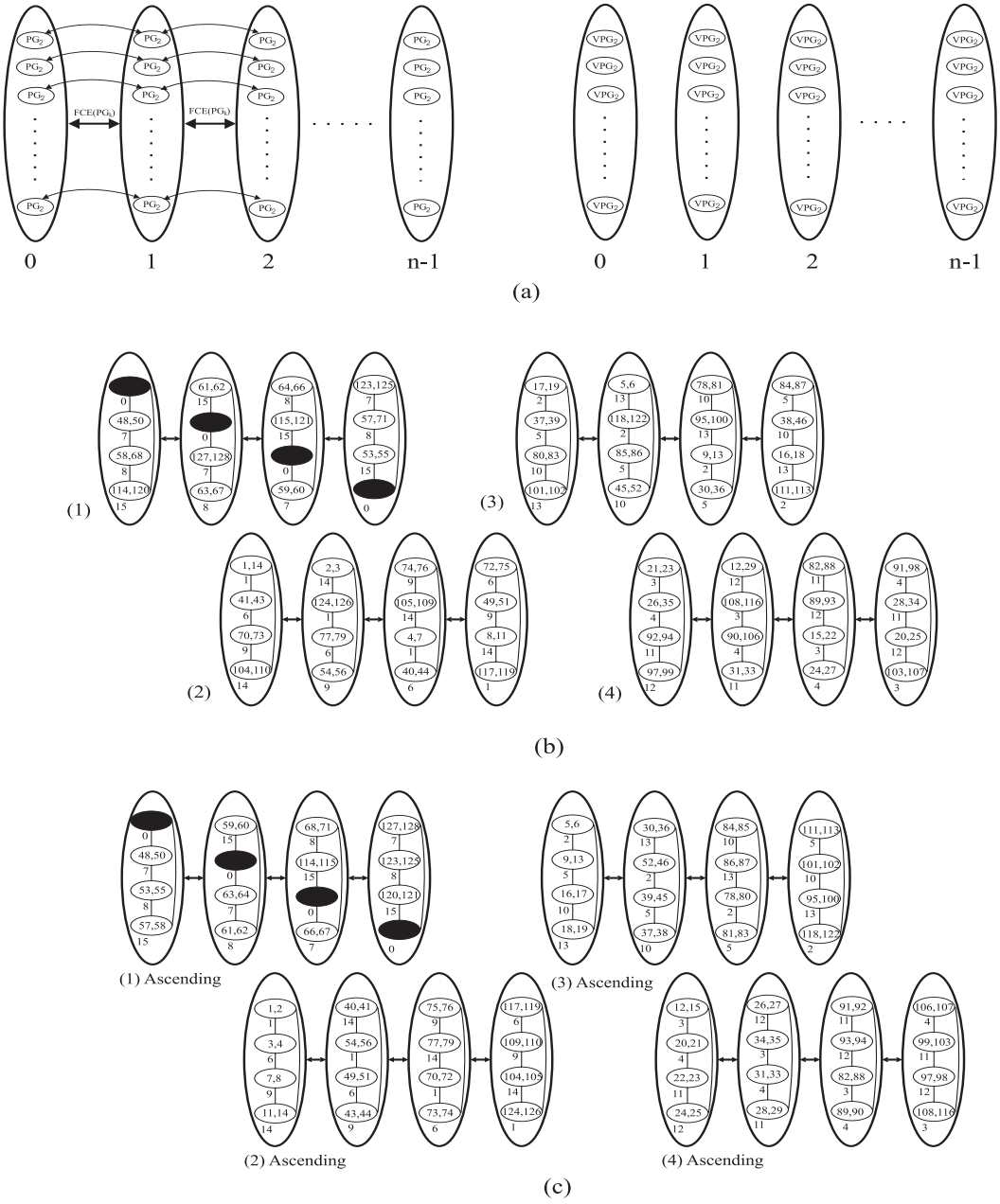

Fig. 12. The merge operation. (a) The merge operation performed on PG_k; (b) Construction of the virtual PG_2 sequence from PG_k; (c) Sorting and FCE(PG_2) operations performed on each virtual PG_2.

4 Analysis of the Time Complexity of the Generalized Fault-Tolerant Sorting Algorithm

In this section, the time complexity of the generalized fault-tolerant sorting algorithm is given. Furthermore, we discuss the time complexity of a torus, grid, hypercube, and the Petersen cube using our generalized fault-tolerant

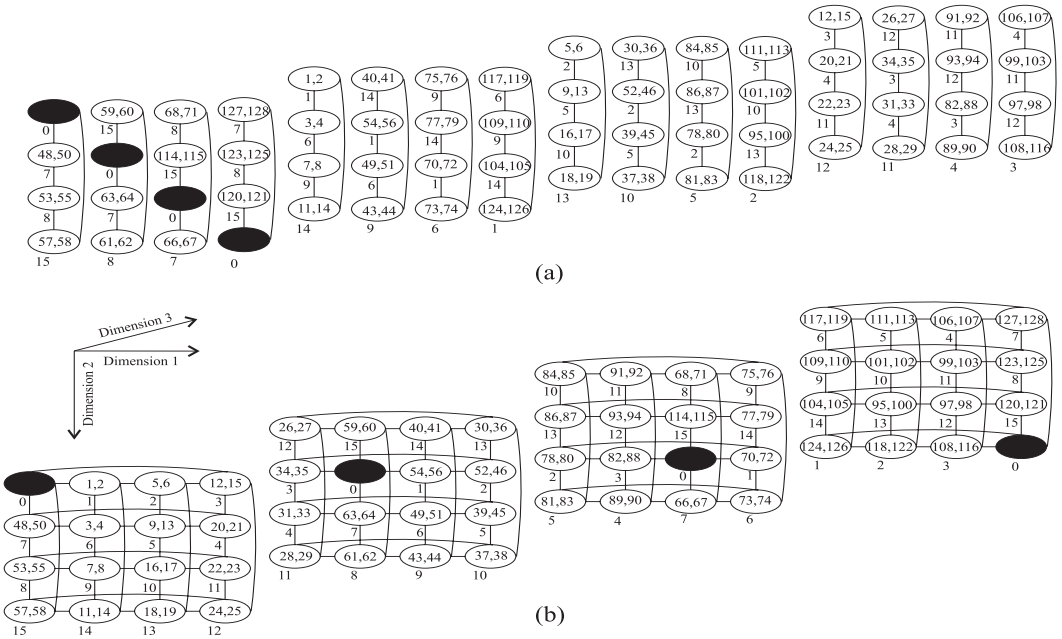

Fig. 13. Interleave step. (a) Snapshot of the virtual PG_2 after executing merge step; (b) Snapshot of the virtual PG_2 after executing the interleave step.

sorting algorithm.

4.1 Generalized Time Complexity

To analyze the time complexity of generalized fault-tolerant sorting algorithm, we first study the time complexity of the sequential sorting algorithm, the communication operation, and the merging process for a k-dimensional product network. We assume that each nonfaulty node contains L keys, where $L = M/(N^r - N^{r-2}) = M/\left((N^2-1)N^{r-2}\right)$. The time cost for the sequential sorting algorithm to run on a node with L keys is denoted T_{ss}. Let T_{s_2} denote the time complexity required for sorting PG_2, T_{s_1} represent the time complexity required for sorting PG in the virtual PG_2, and T_{M_k} be the multiway merging process on a k-dimensional product network. We derive the following Lemma.

Lemma 9 *Merging N sorted sequences of N^{k-1} nodes on PG_k takes $T_{M_k} = T_{s_1}T_{ss}(N+1) + (\left\lfloor \frac{3N}{2} \right\rfloor + 2) \times N + 2(k-2)\left(T_{s_2}T_{ss} + \left\lfloor \frac{3N}{2} \right\rfloor + 2\right)$ time steps.*

Proof: Step 1 of the multi-fault sorting operation takes no computation time. Step 2 is a recursive call to the merging operation for $k-1$ dimensions, and hence requires a time cost of $T_{M_{k-1}}$. Step 3 takes no computation time. Finally, step 4 requires the time for one sorting operation on PG_2, two communication operations for PG_2 (the time for FCE(PG_2)), and one more sorting operation

for PG_2. Every time the keys are sorted, we need to perform a sequential sorting algorithm which takes T_{ss} time steps. Therefore, the value of T_{M_k} can be recursively expressed as:

$$T_{M_k} = T_{M_{k-1}} + 2\left(T_{s_2}T_{ss} + (\left\lfloor \frac{3N}{2} \right\rfloor + 2)\right).$$

In the initial condition, for the two-dimensional PG_2, we perform the sorting operation in PG $N+1$ times, and comparison-exchange in the virtual PG_2 N times. Therefore, T_{M_2} will be

$$T_{M_2} = T_{s_1}T_{ss}(N+1) + (\left\lfloor \frac{3N}{2} \right\rfloor + 2) \times N.$$

This yields

$$T_{M_k} = T_{s_1}T_{ss}(N+1) + (\left\lfloor \frac{3N}{2} \right\rfloor + 2) \times N + 2(k-2)\left(T_{s_2}T_{ss} + (\left\lfloor \frac{3N}{2} \right\rfloor + 2)\right).$$

The time complexity of the fault-tolerant sorting algorithm is $FS_r(N)$.

Theorem 10 *For any factor graph G, the time complexity of the proposed fault-tolerant sorting algorithm on PG_r with $f \leq r-1$ faulty nodes is $FS_r(N) = O(r^2T_{s_2}L\log L + r^2N^2 + rNT_{s_1}L\log L)$, where L is the number of elements distributed on each node.*

Proof: By the algorithm of Section 3.3.2, the time complexity for sorting PG_r with $f \leq r-1$ faulty nodes is the sum of the time complexities for sorting a two-dimensional subgraph and the recursive merging of N sorted sequences into a higher-dimensional product network in PG_r. The derivation of time complexity is as follows.

$$\begin{aligned}
FS_r(N) &= T_{s_2}T_{ss} + T_{M_3} + T_{M_4} + \cdots + T_{M_{r-1}} + T_{M_r} \\
&= T_{s_2}T_{ss} + (r-2)(T_{s_1}T_{ss}(N+1) + (\left\lfloor \frac{3N}{2} \right\rfloor + 2) \times N) + \\
&\quad 2(T_{s_2}T_{ss} + (\left\lfloor \frac{3N}{2} \right\rfloor + 2))\sum_{i=3}^{r}(i-2) \\
&= ((r-1)(r-2)+1)\,T_{s_2}T_{ss} + \\
&\quad (r-2)(r+N-1)(\left\lfloor \frac{3N}{2} \right\rfloor + 2) + (r-2)(N+1)T_{s_1}T_{ss}.
\end{aligned}$$

Since the heap sorting algorithm in the worst case takes $(L-1)\log L + 1$ time steps, the time complexity of $S_r(N)$ becomes

$$
\begin{aligned}
FS_r(N) = & ((r-1)(r-2)+1)T_{s_2}((L-1)\log L+1) + \\
& (r-2)(r+N-1)(\lfloor\frac{3N}{2}\rfloor+2) + \\
& (r-2)(N+1)S(N)((L-1)\log L+1) \\
= & O(r^2T_{s_2}L\log L + r^2N^2 + rNT_{s_1}L\log L).
\end{aligned}
$$

Corollary 11 *The time complexity of odd-even sorting is $O(r^2N^2L\log L)$. If each non-faulty node contains only one key, the complexity is $O(r^2N^2)$.*

Proof: In Theorem 10, we know that the time complexity of our algorithm is $O(r^2T_{s_2}L\log L + r^2N^2 + rNT_{s_1}L\log L)$. We spent $T_{s_2} = O(N^2)$ time steps to perform odd-even sorting in PG_2 with the snake order, and $T_{s_1} = O(N)$ time steps to perform odd-even sorting in PG. Therefore, the time complexity is bounded by $O(r^2N^2L\log L)$. Note that if $L = 1$, the time cost becomes $O(r^2N^2)$.

Corollary 12 *The time complexity of bitonic sorting is $O(r^2L\log L(\log_2 N^2)^2 + r^2N^2 + rNL\log L(\log_2 N)^2)$. If each non-faulty node contains only one key, the complexity is $O(r^2(\log_2 N^2)^2 + r^2N^2 + rN(\log_2 N)^2)$.*

Proof: To perform a bitonic sorting on PG_2, we need

$$
T_{s_2} = \sum_{i=1}^{\log_2 N^2} i \text{ steps, and } T_{s_1} = \sum_{i=1}^{\log_2 N} i
$$

time steps to perform bitonic sorting in PG. Therefore, the time complexity is

$$
O(r^2L\log L(\log_2 N^2)^2 + r^2N^2 + rNL\log L(\log_2 N)^2).
$$

Note that if $L = 1$, the time complexity is bounded by $O(r^2(\log_2 N^2)^2 + r^2N^2 + rN(\log_2 N)^2)$.

4.2 Time Complexity of a Torus

From corollary 11, we know that the complexity of our fault-tolerant sorting on a torus is $O(r^2N^2L\log L)$. Note that if $L = 1$, the time complexity on a torus is $FS_r(N) = O(r^2N^2)$.

4.3 Time Complexity of a Grid

The following corollary measures the time complexity of our fault-tolerant sorting algorithm applied to a grid.

Corollary 13 *If PG_r is a grid, the time complexity of sorting on PG_r is at most $O(r^2N^2L\log L)$, where L is the number of elements each node contains.*

Proof: We calculate the time complexity of our fault-tolerant sorting algorithm on an r-dimensional torus. Then, we refer to the result proposed in [7] which points out that if G is a connected graph, PG_r can emulate any computation on the N^r-node r-dimensional torus by embedding the torus into PG_r with a dilation of three and a congestion of two. Since this embedding undergoes constant dilation and congestion, the emulation has a constant slowdown. (In fact, the slowdown is no greater than six). We use the slowdown value to compute the exact running time for PG_r. The complexity of sorting on r-dimensional torus was previously proposed as $FS_r(N) = O(r^2N^2L\log L)$. Since the emulation of our algorithm by PG_r requires a slowdown factor of at most six, elements of the grid can be sorted in a time complexity $6\times$ $S_r(N) = 6 \times O(r^2N^2L\log L) = O(r^2N^2L\log L)$. Note that if $L = 1$, the time complexity for the grid is bounded by $FS_r(N) = O\left(r^2N^2\right)$.

4.4 Time Complexity of a Hypercube

A hypercube has a constant $N = 2$. We are using the bitonic sorting operation in the single-fault sorting algorithm. From Corollary 13, we can measure the complexity $FS_r(N)$ of the fault-tolerant sorting of a hypercube:

$$FS_r(N) = O(r^2L\log L + r^2 + rL\log L) = O(r^2L\log L).$$

Note that if $L = 1$, the time complexity on the hypercube becomes $FS_r(N) = O\left(r^2\right)$.

4.5 Time Complexity of a Petersen Cube

The Petersen cube is the r-dimensional product network of a Petersen graph, as shown in Fig. 14. The product graphs obtained from the Petersen graph are studied in [14]. Similar to a hypercube, the product of a Petersen graph has a constant N. Since the Petersen graph is Hamiltonian, its two-dimensional product network contains the 10×10 two-dimensional grid as a subgraph.

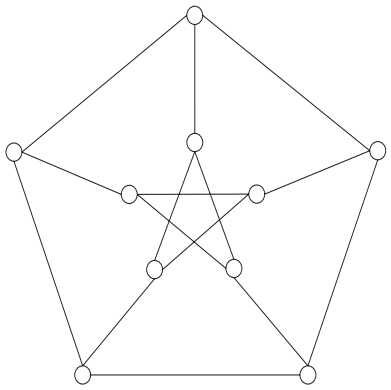

Fig. 14. Petersen graph.

Thus, we can use a grid algorithm for sorting 100 nodes on the two-dimensional product of a Petersen graph in constant time. Consequently, data in the r-dimensional product of a Petersen graph with 10^r nodes can be sorted in a time complexity of $O(r^2 L \log L)$. Note that if $L = 1$, the time complexity for executing the generalized fault-tolerant sorting algorithm on a Petersen cube is $FS_r(N) = O(r^2)$. Table 1 shows a comparison of time complexity of Fernández and Efes' sorting algorithm [8], Sheu *et al.'s* fault-tolerant sorting on a hypercube [19], Chen*'s* fault-tolerant sorting on a hypercube [4], and our generalized fault-tolerant sorting algorithm on a product network. The proposed sorting algorithm is portable for a number of popular product networks. Note that if $L = 1$, the time complexity is bounded by $O(r^2 N^2)$ if the graph is a grid, and by $O(r^2)$ if the graph is a hypercube or a Petersen cube. Moreover, in the case of $L = 1$, the time complexities of hypercube and Petersen cube are the same with the result in Fernández and Efes' algorithm. From Table 1, Fernández and Efes' approach is more efficient when no faults are present. However, our generalized fault-tolerant sorting algorithm is developed to tolerate faults in the product network. Observe that, Fernández and Efes' approach cannot work even if only one fault is occurred. This characteristic of the performance analysis illustrates the performance achievement of the generalized fault-tolerant sorting algorithm.

5 Conclusions

In this paper, we present the fault-tolerant sorting algorithm on an r-dimensional product network when the number of faulty nodes is $f \leq r - 1$. The proposed algorithm is generalized and portable for executing sorting operations on faulty product networks. We first presented the D-split partitioning scheme for partitioning PG_r into a number of $PG_2 s$ such that each PG_2 contains at most one faulty node. To tolerate up to one faulty node, we proposed two single-fault sorting operations executed on each PG_2. We combined the proposed

Table 1
Time complexity comparison of existing sorting and fault-tolerant sorting algorithms.

<table>
<tr><td></td><td colspan="4">Existing Fault-Tolerant Sorting Algorithms</td><td>Fernández and Efes [8](without fault-tolerant)</td><td colspan="2">Our Scheme ($f \leq r-1$)</td></tr>
<tr><td></td><td colspan="4"></td><td>$f=0,\ L=1$</td><td>$f>0,$ $L=1$</td><td>$f>0, L>1$</td></tr>
<tr><td>Torus</td><td colspan="4">unknown</td><td>$O(r^2N)$</td><td>$O\left(r^2N^2\right)$</td><td>$O(r^2N^2L\log L)$</td></tr>
<tr><td>Grid</td><td colspan="4">unknown</td><td>$O(r^2N)$</td><td>$O\left(r^2N^2\right)$</td><td>$O(r^2N^2L\log L)$</td></tr>
<tr><td rowspan="4">Hyper-cube</td><td colspan="2">Sheu et al. [19] ($f \leq r-1$)</td><td colspan="2">Chen [4] ($f \leq \lfloor \frac{3r}{2} \rfloor - 1$)</td><td rowspan="4">$O\left(r^2\right)$</td><td rowspan="4">$O\left(r^2\right)$</td><td rowspan="4">$O(r^2L\log L)$</td></tr>
<tr><td>$f>0,$</td><td>$f>0,$</td><td>$f>0,$</td><td>$f>0,$</td></tr>
<tr><td>$L=1$</td><td>$L>1$</td><td>$L=1$</td><td>$L>1$</td></tr>
<tr><td>$O(\log^2 2^r)$</td><td>$O(L\log L+ L\log^2 2^r)$</td><td>$O(\log^2 2^r)$</td><td>$O(L\log L+ L\log^2 2^r)$</td></tr>
<tr><td>Petersen cube</td><td colspan="4">unknown</td><td>$O\left(r^2\right)$</td><td>$O\left(r^2\right)$</td><td>$O(r^2L\log L)$</td></tr>
</table>

where, f is the number of faulty nodes, L is the number of elements on each node, r is the dimension, and N is the number of nodes of the factor graph.

single-fault sorting operations with the modified multi-way merging operation as the basic operation for tolerating multiple faults. The time complexity of the proposed fault-tolerant sorting algorithm is $O(r^2L\log L(\log_2 N^2)^2+r^2N^2+ rNL\log L(\log_2 N)^2)$ when using bitonic sorting and is $O(r^2N^2L\log L)$ when using odd-even sorting, where L is the number of data distributed on each node and $f \leq r-1$. For particular networks, the time complexity for the grid is $O(r^2N^2L\log L)$ and for a hypercube and Petersen cube is $O(r^2L\log L)$. Note that if $L=1$, the time complexities of hypercube, and Petersen cube are the same as the result in Fernández and Efes' approach. From Table 1, Fernández and Efes' approach is more efficient when no faults are present. However, our generalized fault-tolerant sorting algorithm is developed to tolerate faults in the product network. Fernández and Efes' approach cannot work even if one fault is occurred. Consequently, the performance analysis indicates that our proposed generalized sorting scheme is a truly efficient fault-tolerant scheme.

References

[1] K. E. Bacher. On bitonic sorting networks. *Proc. 1990 Int'l Conf. Parallel Processing*, I:376–379, 1990.

[2] K. E. Bacher. Sorting Networks and their Applications. *Proc. AFIPS Spring Joint Computing Conf.*, 307–314, 1968.

[3] M. Baumslag and F. Annexstein. A unified framework for offline permutation routing in parallel networks. *Math. Systems Theory*, 24(4):233–251, 1991.

[4] Y. W. Chen. The design and analysis of fault-tolerant prefix computation, sorting and embedding algorithms on hypercube. *PhD thesis, Graduate*

School of Management, National Taiwan Univ. of Science and Technology, Taipei, 1999.

[5] T. E. Ghazawi and A. Youssef. A general framework for developing adaptive fault-tolerant routing algorithms. *IEEE Transactions on Reliability*, 42:250–258, June 1993.

[6] A. Fernández. Homogeneous product networks for processor interconnection. *PhD thesis, Univ. of Southwestern Louisiana, Lafayette*, Oct. 1994.

[7] A. Fernández and K. Efe. Mesh-connected trees: a bridge between grids and meshes of trees. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1281–1291, Dec. 1996.

[8] A. Fernández and K. Efe. Generalized algorithm for parallel sorting on product networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1211–1225, Dec. 1997.

[9] A. Fernández and K. Efe. Efficient vlsi layouts for homogeneous product networks. *IEEE Transactions on Computers*, 46(10):1070–1082, Oct. 1997.

[10] A. Fernández and K. Efe. Product networks with logarithmic diameter and fixed degree. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):963–975, Sept. 1995.

[11] J. Hastad, T. Leighton and M. Newman. Reconfiguring a hybercube in the presence of faults. *19th Annual ACM Symposium on Theory of Computing*, 274–284, 1987.

[12] D. R. Őhring and D. H. Hohndel. Optimal fault-tolerant communication algorithms on product netowrks using spanning trees. *Proc. of sixth IEEE Symposium on Parallel and Distributed Processing*, pages 188–195, Jan. 1994.

[13] D. R. Őhring, M. Ibel, and S. K. Das. Reliable broadcasting in product networks in the presence of faulty nodes. *Proc. of seventh IEEE Symposium on Parallel and Distributed Processing*, pages 711–718, 1995.

[14] S. R. Őhring and S.K. Das. The folded petersen cube network: New competitiors for the hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):151–168, Feb. 1996.

[15] D. L. Lee and K. E. Batcher. On sorting multiple bitonic sequences. *Proc. 1994 Int'l Conf. Parallel Processing*, I:121–125, 1994.

[16] K. J. Liszka and K. E. Batcher. A generalized bitonic sorting network. *Proc. 1993 Int'l Conf. Parallel Processing*, I:105–108, 1993.

[17] T. Nakatani, S.-T. Hung, B. W. Arden, and S. K. Tripathi. K-way bitonic sort. *IEEE Transactions on Computers*, 38(2):283–288, Feb. 1989.

[18] A. L. Rosenberg. Product-shuffle network: Toward reconciling shuffles and butterflies. *Discrete Applied Math.*, 37/38:465–488, 1992.

[19] J. P. Sheu, Y. S. Chen, and C. Y. Chang. Fault-tolerant sorting algorithm on hypercube multicomputers. *Journal of Parallel and Distributed Computing*, 16:185–197, 1992.

[20] S.-H. Chang Y.-C. Tseng and J.-P. Sheu. Fault-tolerant ring embedding in a star graph with both link and node failures. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1285–1295, 1997.

Yuh-Shyan Chen received the B.S. degree in computer science from Tamkang University, Taiwan, Republic of China, in June 1988 and the M.S. and Ph.D. degrees in Computer Science and Information Engineering from the National Central University, Taiwan, Republic of China, in June 1991 and January 1996, respectively. He joined the faculty of Department of Computer Science and Information Engineering at Chung-Hua University, Taiwan, Republic of China, as an associate professor in February 1996. He joined the Department of Statistic, National Taipei University in August 2000, and then joined the Department of Computer Science and Information Engineering, National Chung Cheng University in August 2002. Dr. Chen served as Editor-in-Chief of International Journal of Ad Hoc and Ubiquitous Computing (IJAHUC), Editorial Board Member of International Journal of Internet Protocol Technology (IJIPT) and The Journal of Information, Technology and Society (JITAS), Guest Editor of Telecommunication Systems, special issue on "Wireless Sensor Networks" (2004), and Guest Editor of Journal of Internet Technology, special issue on "Wireless Internet Applications and Systems" (2002) and special issue on "Wireless Ad Hoc Network and Sensor Networks" (2004). He was a Workshop Co-Chair of the 2001 Mobile Computing Workshop, IASTED Technical Committee on Telecommunications for 2002 2005, Program Committee Member of IEEE ICPP'2003, IEEE ICDCS'2004, IEEE ICPADS'2001, IEEE ICCCN'2001 2004, IASTED CCN'2002 2004, IASTED CSA'2004, IASTED NCS'2005, and MSEAT'2003 2004. His paper wins the 2001 IEEE 15th ICOIN-15 Best Paper Award. His recent research topics include mobile ad-hoc network, wireless sensor network, Bluetooth WPANs, mobile computing, mobile learning system, and mobile P2P communication. Dr. Chen is a member of the IEEE Computer Society, IEICE Society, and Phi Tau Phi Society.

Chih-Yung Chang received the Ph.D. degree in Computer Science and Information Engineering from National Central University, Taiwan, in 1995. He joined the faculty of the Department of Computer and Information Science at Aletheia University, Taiwan, as an assistant professor in 1997. He was the Chair of the Department of Computer and Information Science, Aletheia University, from August 2000 to July 2002. He is currently an associate professor of Department of Computer Science and Information Engineering at Tamkang University, Taiwan. Dr. Chang is a member of Editorial Board of Tamsui Oxford Journal of Mathematical Sciences and a member of the IEEE Computer Society and IEICE society. His current research interests include wireless sensor networks, mobile learning, Bluetooth radio systems, Ad Hoc wireless networks, and mobile computing.

Tsung-Hung Lin received the B.S. degree in computer science from Tamkang University, Taiwan, R.O.C., in June 1988 and the M.S. degree in computer science and information engineering from National Chung Cheng University, Taiwan, in 1993. He is currently a graduate student for the Ph.D. degree in the Department of Computer Science and Information Engineering, National

Chung Cheng University. His research interests include mobile computing, wireless sensor network, and WCDMA systems.

Chun-Bo Kuo received the M.S. degrees in Computer Science and Information Engineering from Chung-Hua University, Taiwan, R.O.C., in June 1999 . His research interests include parallel and distributed processing and collective communication.