A Generalized Fault-Tolerant Sorting Algorithm on Product Network*

Chih-Yung Chang
Department Computer and Information Science
Aletheia University, Tamsui, Taipei, Taiwan
Email: changcy@email.au.edu.tw

Yuh-Shyan Chen and Chun-Bo Kuo
Department of Computer Science and Information Engineering
Chung-Hua University, Hsin-Chu, Taiwan
Email: chenys@csie.chu.edu.tw

Abstract

Product networks define a class of topologies that are used very often such as mesh, tori, and hypercube, etc. This paper proposes a generalized algorithm for fault-tolerant parallel sorting in the product networks. To tolerate multiple faulty nodes, product network is partitioned into a number of subgraphs such that each subgraph contains at most one fault. Our generalized sort algorithm is divided into two steps. First, a single-fault sorting operation is presented to correctly execute on each faulty subgraph containing one fault. Second, each subgraph is considered as a supernode, and a fault-tolerant multiway merge operation is presented to recursively merge two sorted subsequences into one sorted sequence. Our generalized sort algorithm can be applied on any product network if the factor graph of product graph can at least embed a ring structure. Further, we also show the time complexity of our sorting operations on the grid, hypercube, and petersen cube. The performance analysis illustrates that our generalized fault-tolerant sort algorithm is near-optimal.

1 Introduction

Product network defines a class of topologies that are used very often. A lot of researches on product networks have been done in recent literature[4][5][6]. This network has interesting topological properties that make them especially suitable for parallel algorithms. Examples of product networks include hypercubes, grids, and tori.

There are many algorithms developed for special case of the 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 [3][7] cannot work on a grid, even though both hypercube and grid are product network. Recently, A. Fernández and K. Efe [4] proposed a generalized sorting algorithm on product network. The main function of their algorithm is to propose a multiway-merge operation. However, their algorithm is without fault-tolerant capability. This paper presents a generalized fault-tolerant sorting algorithm on product network. Our fault-tolerant sorting algorithm is based on a A. Fernández and K. Efes' algorithm. We modify the multiway-merge operation as a fault-tolerant multiway-merge operation. By using this fault-tolerant multiway-merge operation, we present the fault-tolerant sorting algorithm on product network.

Our generalized sort algorithm is divided into two steps. Firstly, a single-fault sorting operation is presented to correctly execute on faulty subgraphs that each contains at most one fault. Secondly, each subgraph is considered as a supernode. A fault-tolerant multiway merge operation is presented to recursively merge two sorted subsequences into one sorted sequence. Our generalized sort algorithm can be applied on any product network under the constraint the factor graph of product graph can at least embed a ring structure. The performance study on grid, hypercube, and person cube illustrates that our generalized fault-tolerant sort algorithm is near-optimal.

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

2 Preliminaries

In this section, we define the product network and derive the partitioning property of product

*This work was supported by National Science Council of the Republic of China under grant NSC 89-2213-E-216-010

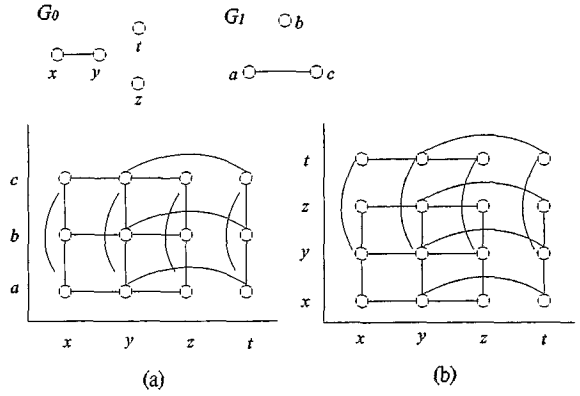

Figure 1: Examples of product network:(a) product network of G_0 and G_1; (b) product network of G_0 and G_0.

network. Finally, we present the snake ordering method for product network in Subsection 2.2.

2.1 Product Network

An interconnection 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 iff 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. Figure 1 shows an example of product network.

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 Figure 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 $u = 0, 1, \cdots, N-1$. Thus, the label l_x of every vertex $x \in PG_{r-1}$becomes ul_x. We describe the construction of PG_r from PG_{r-1} in logical view. First, arrange all the vertices of PG_{r-1} one by one along a horizontal (or vertical) direction. Then, make N copies of PG_{r-1} along vertical (or horizontal) direction such that the vertices with identical label fall in the same column. Next, relabeling 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 $(u, u') \in E_G$. Figure 2 illustrates this construction process for two and three dimensional products graph. The factor graph G is shown in Figure 2(a). Nodes in the ith row of Figure 2(b) are labeled by putting an additional digit i before their labels. Thus, the ith row in Figure 2(b) can be viewed as $[i]PG_1$. By similar way, we may con-

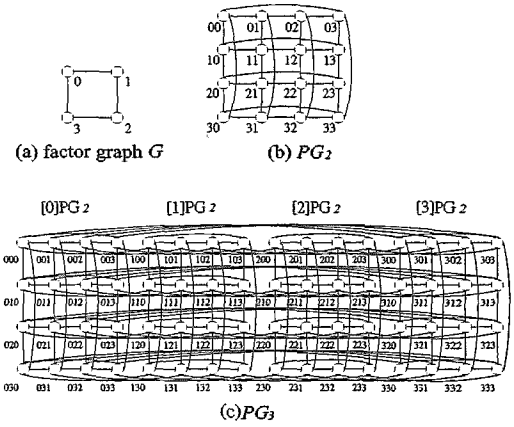

Figure 2: Recursive construction of multidimensional product network. (a) The factor graph; (b) Two-dimensional product; (c) Three-dimensional product.

struct PG_3 as shown in Figure 2(c). Since the operations described above is 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.

2.2 Network Partitioning

To perform the fault-tolerant sorting operation on PG_r, we first partition PG_r into a number of PG_2. The jth-split operation on PG_r is defined by partitioning PG_r along dimension-j into N copies of PG^j_{r-1}. Let $D = (d_1, d_2, ..., d_n), n < r$. The D-split on PG_r is the operation to apply d_1th-split, d_2 th-split, ..., and d_n th-split on PG_r.

Theorem 1: We can obtain N^k copies of $PG^{i_1,\cdots,i_k}_{r-k}$ by partitioning PG_r along k dimensions $i_1, i_2, \cdots, i_k$, where $k < r$, N is the number of nodes of factor graph.
Proof: The proof can be referred to [2].

The notation $[u]PG^i_{r-1}$ naturally defines an ordering for subgraphs PG_{r-1}. In general, $[u]PG^i_{r-1}$ is the uth copy of the PG_{r-1}subgraph at dimension i. The subgraph ordering has a number of different ways. We define a particular subgraph ordering method, say *snake ordering*, with certain useful properties for data sorting.

Definition 1: The *snake order* for the product graph PG_r is defined by:

1) If $r = 1$, the snake order is the same as the order used for labeling the nodes of G.

2) Assume the snake order has been defined for PG_{r-1}, $r > 1$. Then

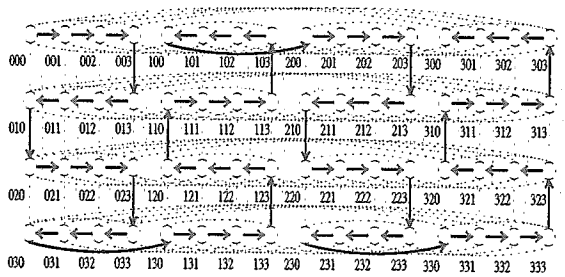

Figure 3: The snake order of the product network PG_3 whose factor graph is a 4-node ring.

(a) $[u]PG_{r-1}^r$ has the same order as PG_{r-1} if u is even, and reverse order if u is odd.

(b) if $u < v$ then the order of all vertices in $[u]PG_{r-1}^r$ precedes the order of all vertices in $[v]PG_{r-1}^r$.

Figure 3 shows the snake order for a product graph PG_3 shown in Figure 2(c). As discussed in latter, if the factor graph of product network is a ring structure, we use the odd-even like sorting operation as single-fault algorithm executing on each PG_2. In this case, each node in PG_2 will be assigned a processor number according to the snake order. On the other hand, if the factor graph is a hypercube structure, we use the bitonic-like sorting operation as single-fault sorting algorithm executing on each PG_2. The processor numbering for each node in PG_2 is according to the original order.

3 Generalized Algorithm of Fault-Tolerant Sorting

For a faulty product graph PG_r, we will first partition it into several subgraphs PG_2 such that each PG_2 contains at most one faulty node. To tolerate up one faulty node, we propose two single-fault sorting operations for each PG_2. This ensures that we can obtain a 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 merge operation [1] which is originally without fault-tolerate capability.

We outline our generalized fault-tolerant sorting algorithm as follows. The detailed operations are presented in the following subsections.

/* Fault-Tolerant Sorting Algorithm on PG_r */
Generalized Fault_Tolerant_Sorting (G, r, F, M)
/* Sorting M keys on PG_r with F faults */
{
Partition_PG_r:
/* Partition 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 as a faulty node.
Distribute_Data:
Distribute M elements to the nonfaulty nodes.
Single_Fault_Sorting:
/* Apply single-fault-sorting algorithm on each PG_2 */
For each PG_2,
if (G can be embedded to 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:
Recursively perform our fault-tolerant multiway merge operation to merge unmerged keys from PG_i into PG_{i+1}, where $2 \leq i \leq r-1$.
}

3.1 Partitioning Scheme for Faulty Product Network

To tolerate up $r-1$ faults in faulty PG_r, we partition original 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. In the condition of $f \neq r-1$, there exist some partitioned PG_2 without faulty node. For the reasons of regular operation and balancing the workload of each PG_2, we determine a dangling node in each nonfaulty PG_2. Node in nonfaulty PG_2 with the same position of most faulty node in all of faulty PG_2 will be selected as a dangling node. We will logically consider the dangling node as faulty node and will not assign any data to it.

3.2 Distribute Unsorted Keys

The next step is to distribute unsorted keys into all nonfautly nodes. Assume that there are $M \gg N^r$ unsorted elements. Since the total number of nonfaulty node is $N^r - N^{r-2}$, each nonfaulty node contains $M/(N^r - N^{r-2}) = M/((N^2-1)N^{r-2})$ keys.

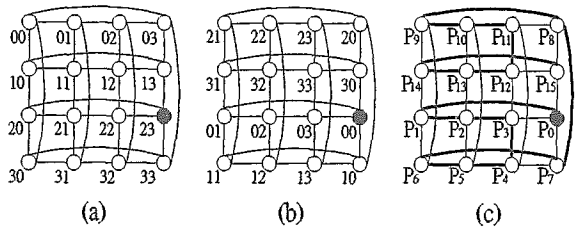

Figure 4: Relabeling the processor number for subproduct network according to snake order. (a) The original labels of graph; (b) Relabeling the faulty node to position 00; (c) Relabeling the processor number for each node according to snake order.

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 ASCENT/DESCEND order. As mentioned before, we apply single-fault odd-even sort operation on each PG_2 if the factor graph G can embed a hamintian cycle. If an n-cube structure can be embedded into the original factor graph G, we perform the single-fault bitonic sorting operation on each PG_2.

Before applying one of these two sort operations, a simple rotation operation is performed based on the address of faulty node. The purpose of rotation operation is to reset the logical address of nodes such that the logical address of faulty node and dangling node can be considered as P_0. For example, consider a PG_2 containing a faulty node whose label is 23 as shown in Figure 4(a). After rotation, the faulty node can be considered as P_0 as shown in Figure 4(b). Noted that the label of each node has changed by rotation operation as shown in Figure 4(b). According to snake order, we assign each node a processor number as shown in Figure 4(c).

Before executing the single-fault sorting operation, all nodes in PG_2 should be assigned a processor number. If odd-even sort operation is determined to apply on PG_2, the processor numbering is according to snake order. On the other hand, if bitonic sorting operation is determined to apply on PG_2, the processor numbering is according to original labels' order. The relabeling of original order is the same as one in snake order.

After applying the relabeling operation, we perform single fault sorting operation on each PG_2. The single-fault odd-even sort operation is applied on each PG_2 if the factor graph G can embed a hamintian cycle. The odd-even sorting alrogithm which deals with one fault is now described. First, we apply the sequential sorting algorithm, such as quick sort or heap sort, on $M/((N^2-1)N^{r-2})$ elements of each nonfaulty node. Then, in the odd step, each pair of nodes P_n and P_{n-1}, for n is odd, compares each other element by element. In the even step, each pair of nodes P_n and P_{n-1}, for n is even, applies the compare-and-exchange operation element by element. After each step of odd-even sorting algorithm, we also need to perform the sequential sorting algorithm to each node. Since P_0 is a faulty node, P_0 and P_1 do nothing in both odd and even steps.

If an n-cube structure can be embedded into the original factor graph G, we perform the single-fault bitonic sorting operation on each PG_2. The bitonic sorting algorithm consists of $\sum_{i=1}^{log_2 n} i$ compare-exchange stages for n elements. The key concept of the bitonic sorting algorithm is recursively executing the comparison-exchange operations on subcube such that the first half of elements are located in one subcube and the last half of elements are located in another subcube. Noted that node P_0 performs no operation during the execution of compare-and-exchange operations. No matter the odd-even sort or bitonic sort algorithm is applied to perform the single fault sorting operation, the next step of our generalized sorting algorithm is to perform the fault-tolerant multiway merge operation, as described in the next subsection.

3.4 Fault-Tolerant Multiway Merge Operation

A. Fernández and K. Efe [4] proposed a generalized parallel sorting algorithm. The kernel function is the multiway merge operation[1]. Before discussing the fault-tolerant multiway merge operation, we firstly define a fundamental operation, namely the fault-tolerant compare-exchange operation. Our fault-tolerant merge operation is built based on the fault-tolerant compare-exchange operation.

3.4.1 Fault-Tolerant Compare-Exchange Operation

The main function of the fault-tolerant compare-exchange operation is to perform the compare-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 compare-exchange operation is a recursive operation. Let FCE(PG_2) denote the fault-tolerant compare-exchange operation on any pair of two copies of PG_2. We describe the fault-tolerant compare-exchange operation in follows. Every PG_2 exactly has one faulty or dangling node. Three possible cases are discussed depending on the location of 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 product network, each pair of nodes x and y located in same location can logical 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' located in the same physical location.) Each node $x \neq f$ sends its data to adjacent node $y \neq f'$ by physical link and performs the compare-exchange operation. The time complexity for sending data to adjacent node is $O(\lfloor \frac{N}{2} \rfloor)$.

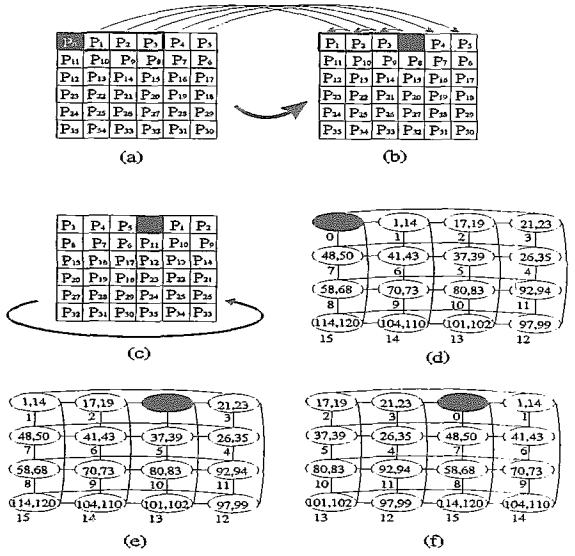

Figure 5: The FCE(PG_2) operation executed on example of case 2 that faulty nodes of two PG_2 are located at the same row. The processor numbering is in snake order.

Case 2. (Nodes f and f' 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 are faulty nodes, where $i < j$. Processor sequence $P_{i+1}, P_{i+2}, \ldots, P_j$ will send data to $P'_i, P'_{i+1}, \ldots, P'_{j-1}$ by 2-hops as follows. First, 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 neighbor 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. For nodes P_t that are not located in the faulty row, they will send data to node P'_t with same processor number in PG'_2. The time complexity of data moving phase is thus $O(\lfloor \frac{N}{2} \rfloor + 1)$. Figure 5 illustrates this operation. Figure 5(a) shows the processor numbering of PG_2 and Figure 5(d) illustrates the same PG_2 with data which have been sorted in PG_2 in an ascending snake order. Figure 5(b) illustrates the data in nodes of first row (the row the faulty node located) moving from PG_2 to PG'_2. Figure 5(e) shows the data layout of PG'_2 after data moving operation performed on each node.

2. **Rotation phase:**All nodes except the faulty node perform a rotation operation as follows. All nodes in each row repeatedly shift right a position until a node with smallest processor number arrives the position of $j+1$. The time complexity of rotation phase is then $O(N)$. Figure 5(c) illustrates the result of rotation phase and Figure 5(f) illustrates the resultant data layout of Figure 5(c).

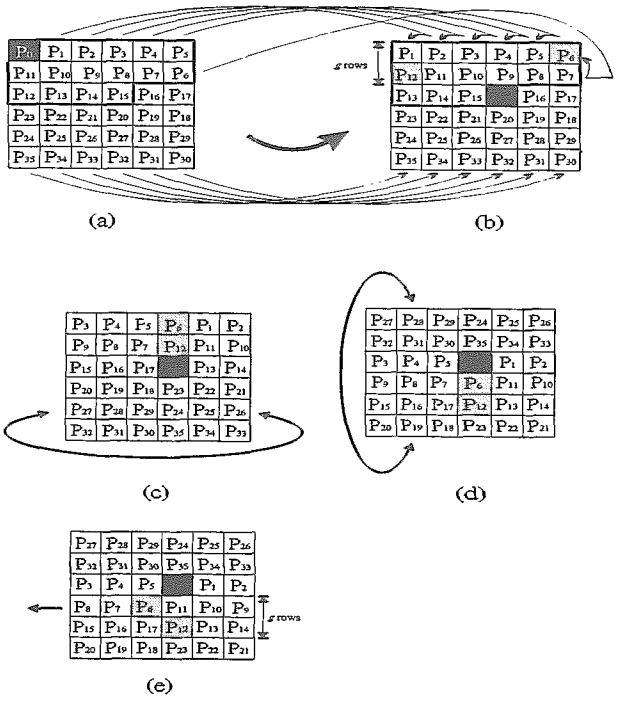

Figure 6: The snapshot of operation of redistribution step.

Case 3. (Nodes f and f' located in different physical row.) Initially, a similar data moving phase in **Case 2** is done. Figures 6(a) and 6(b) display the operation of moving phase. The only difference is that the number of faulty rows is more than one. We perform the rotation operations. The rotation operation can be divided into three phases as described in what follows.

1. **Horizontal rotation phase:** Each row performs a horizontal rotation operation as follows. Assume that the faulty node is located in the jth-column. Let row number of the first row be labeled by 0. For that row whose row number is less than the faulty row, all nodes shift left/right a position until a node with maximum address arrives at the jth-column. If the faulty row with an odd/even number, all the nodes in the faulty row except the faulty node repeatedly shift left/right a position until a node with largest/smallest address arrives at the $(j+1)$th-column. For the rest rows whose row number is odd/even, all nodes shift left/right a position until a node with maximum/minimum address arrives at the jth-column. The time complexity of horizontal rotation phase is $O(\lfloor \frac{N}{2} \rfloor)$ time steps. Operation of the horizontal rotation is shown in Figure 6(c).

2. **Vertical rotation phase:** All nodes repeatedly shift up/down one position until the first row arrives at the faulty row. The jth-column don't need to shift up one position at first step. The time complexity of vertical rotation phase is then $O(\lfloor \frac{N}{2} \rfloor)$ time steps. Operation of vertical rotation is shown in Figure 6(d).

3. **Tuning rotation phase:** Assume the distance between the first row and the faulty row is g. A tuning operation is needed to perform in the next g rows beginning from the faulty row. The task is described as follows. Assume that the faulty row is relabled by row 0. For every row of these g rows, if its row number is odd, nodes on this row shift one position to the left. The time complexity of tuning rotation phase is $O(1)$. Operation of the tuning rotation phase is shown in Figure 6(e).

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

Proof: The proof can be referred to [2].

3.4.2 The Fault-Tolerant Multiway Merge Operation

The multiway merge operation was originally used by A. Fernández and K. Efe [4]. However, their multiway merge operation is without fault-tolerant capability. The proposed multiway merge 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 virtual PG_2 and virtual PG_2 sequence in follows. The *virtual* PG_2 is consists of N copies of PG_1. Any two PG_1 in the virtual PG_2 may not connected directly. The structure of virtual PG_2 is similar to PG_2 except that the communication of each pair of neighboring PG_1 may need more than one step since there may not exist direct link between them. The *virtual* PG_2 *sequence* is the sequence of a number of virtual PG_2.

Similar to [4], the fault-tolerant multiway merge operation is consist of redistribution step, merge step, interleave step, and clear-dirty step. The only difference is at the opertion of $k = 2$ in redistribution step and merge step. In case of $k = 2$ of redistribution step, we are not only collecting unmerge data from different dimensions, but also collecting data from all of faulty columns of every original PG_2 to organize a virtual PG_2. Figure 7 displays the snapshot of redistribution step. In case of $k = 2$ of merge step, the single-fault sorting and FCE(PG_2) are performed on the virtual PG_2 sequence.

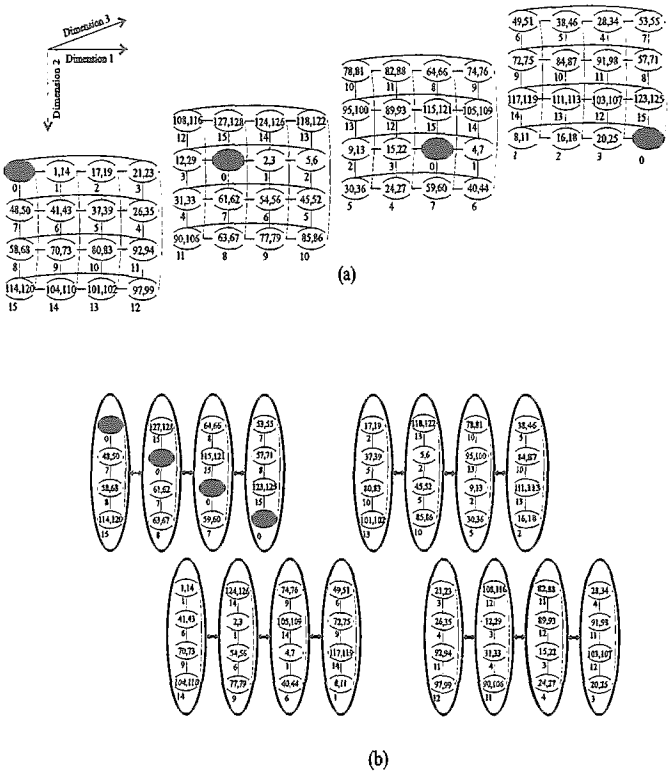

Figure 7: The snapshot of operation of redistribution step.

4 Analysis of Time Complexity of Generalized Sorting Algorithm

In this section, we give the time complexity of the generalized sorting algorithm. We discuss the time complexity of torus, grid, hypercube, and petersen cube by applying our algorithm.

4.1 Generalized Time Complexity

To analyze the time complexity of the sorting algorithm, we will study the time complexity of sequential sorting algorithm, communication operation and the merge process on a k-dimensional product network. We assume that each nonfaulty node contains L keys. We have $L = M/(N^r - N^{r-2}) = M/\left((N^2 - 1)N^{r-2}\right)$. The time cost of sorting L keys on a node is denoted as T_{ss}. Let T_{s_2} denote the time complexity required for sorting operation performed on PG_2, T_{s_1} denote the time complexity required for sorting operation performed on PG in virtual PG_2, and T_{M_k} denote the time complexity of the multiway merge process performed on a k-dimensional product network.

Lemma 1: Merging N sorted sequences of N^{k-1} nodes on PG_k takes $T_{M_k} = T_{s_1}T_{ss}(N + 1) +$

$(\lfloor \frac{3N}{2} \rfloor + 2) \times N + 2(k-2)\left(T_{s_2}T_{ss} + \lfloor \frac{3N}{2} \rfloor + 2\right)$ time steps.

Proof: Step 1 of multi-fault sorting operation takes only constant computation time. Step 2 is a recursive call to merge operation for $k-1$ dimensions, and hence takes $T_{M_{k-1}}$ time cost. Step 3 takes only constant computation time. Finally, step 4 takes the time of one sorting operation performed on PG_2, two communication operations performed on PG_2 (the time for FCE(PG_2)), and another sorting operation performed on PG_2. Whenever we sort the keys of a node, we need to perform a sequential sorting algorithm. This implies that we pay time complexity T_{ss} for each step of merging operation.

The value of T_{M_k}, therefore, can be recursively expressed as:

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

In total, we perform $N+1$ times of sorting operation in PG, and N times of compare-exchange in virtual PG_2. Therefore, T_{M_2} will be

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

This yields

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

Now we can derive the value of fault-tolerant sorting algorithm $FS_r(N)$ by the following Theorem.

Theorem 3: For any factor graph G, the time complexity of fault-tolerant sorting on PG_r 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 keys contained in each node and G contains $f \le r-1$ faulty nodes.

Proof: By the algorithm of Section 3, the time complexity taken to sort on PG_r, with $f \le r-1$ faulty nodes, is the time complexity taken to sort a two dimensional subgraph and then recursively merge N sorted sequences into an increasing order in PG_r. The expression of the time complexity is measured as follow:

$$\begin{aligned} FS_r(N) &= T_{s_2}T_{ss} + T_{M_3} + T_{M_4} + \cdots + T_{M_{r-1}} + T_{M_r} \\ &= T_{s_2}T_{ss} + (r-2)(T_{s_1}T_{ss}(N+1) + (\lfloor \tfrac{3N}{2} \rfloor + 2) \times N) + 2(T_{s_2}T_{ss} + (\lfloor \tfrac{3N}{2} \rfloor + 2)) \textstyle\sum_r^{i=3}(i-2) \\ &= ((r-1)(r-2)+1)T_{s_2}T_{ss} + (r-2)(r+N-1)(\lfloor \tfrac{3N}{2} \rfloor + 2) + (r-2)(N+1)T_{s_1}T_{ss}. \end{aligned}$$

Since complexity of heap sort algorithm, in the worst case, is $(L-1)\log L + 1$ time steps, the time complexity of $FS_r(N)$ becomes:

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

Corollary 1: The time complexity of odd-even-like sorting is $O(r^2N^2L\log L)$. If each non-faulty node contains only one key, the complexity will become $O(r^2N^2)$.

Proof: In Theorem 3, we get the generalized time complexity of our algorithm is $O(r^2T_{s_2}L\log L + r^2N^2 + rNT_{s_1}L\log L)$. We have paid $T_{s_2} = O(N^2)$ time steps to perform odd-even sorting in a PG_2 with snake order, and $T_{s_1} = O(N)$ time steps to perform odd-even sorting in a PG. Therefore, the time complexity can be bounded to $O(r^2N^2L\log L)$. Note that if $L = 1$, the time cost becomes $O(r^2N^2)$.

Corollary 2: The time complexity of bitonic-like 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 will become $O(r^2(\log_2 N^2)^2 + r^2N^2 + rN(\log_2 N)^2)$.

Proof: From Theorem 3, we know that the time complexity of the generalized algorithm. When we perform the bitonic sorting on a PG_2, we need $T_{s_2} = \sum_{i=1}^{\log_2 N^2} i$ time steps, and $T_{s_1} = \sum_{i=1}^{\log_2 N} i$ time steps to perform bitonic sorting in a 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 will be bounded to $O(r^2(\log_2 N^2)^2 + r^2N^2 + rN(\log_2 N)^2)$.

4.2 Time Complexity of Torus and Grid

From Corollary 2, we have that the complexity of our fault-tolerant sorting on torus is $O(r^2N^2L\log L)$. Note that if $L = 1$, the time complexity on tours is $FS_r(N) = O\left(r^2N^2\right)$. To analyze the time complexity of grid, we propose the follow corollary to explain the time complexity by using our faulty-tolerant sorting algorithm.

Corollary 3: If PG_r is a grid, the time complexity of sorting operation performed on PG_r is at most $O(r^2N^2L\log L)$.

Proof: To prove, we first propose the complexity of our faulty-tolerant sorting algorithm on the r-dimensional tours. Then, we refer to a result in [8] which indicates that, if G is a connected graph, PG_r can emulate any computation on the N^r-node r-dimensional torus by embedding the tours into PG_r with dilation three and congestion two. Since this embedding has constant dilation and congestion, the emulation has constant slowdown. (In fact, the slowdown is no more than six). We use the slowdown values to compute the exact running time for PG_r.

The complexity of sorting on r-dimensional torus had been proposed in pervious. That is, $S_r(N) = O(r^2N^2L\log L)$. Since the emulation of our algorithm by PG_r requires a slowdown factor of, at most, six, grid can sort with 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 on grid is bounded on $FS_r(N) = O\left(r^2N^2\right)$.

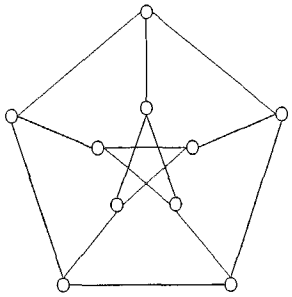

Figure 8: Peterson graph.

4.3 Time Complexity of Hypercube

The hypercube has fixed $N = 2$. We use the bitonic sorting operation in single-fault sorting operation. From Corollery 3, we can compute the complexity of fault-tolerant sorting on hypercube. That is, $FS_r(N) = O(r^2 L \log L + r^2 + rL \log L) = O(r^2 L \log L)$. Note that if $L = 1$, the time complexity on hypercube becomes $FS_r(N) = O\left(r^2\right)$.

4.4 Time Complexity of Petersen Cube

The Petersen cube is the r-dimensional product of the Petersen graph, as shown in Figure 8. Product graph obtained from the Petersen graph is studied in [9]. Like the hypercube, the product of Petersen graph has fixed N. Since the Petersen graph is Hamiltonian, its 2-dimensional product contains the 10×10 2-dimensional grid as a subgraph. Thus, we can use grid algorithm for sorting 100 nodes on the 2-dimensional product of Petersen graph in constant time. Consequently, the r-dimensional product of Petersen graph can sort 10^r nodes in $O(r^2 L \log L)$. Note that if $L = 1$, the time complexity on petersen cube will be $FS_r(N) = O\left(r^2\right)$.

5 Conclusions

In this paper, we present the fault-tolerant sorting algorithm on $r-$dimensional product network with the number of faulty nodes $f \leq r - 1$. The proposed algorithm is generalized and portable for executing sorting operations on faulty product network. The time complexity of the proposed fault-tolerant sorting algorithm is $O(r^2 L \log L(\log_2 N^2)^2 + r^2 N^2 + rNL \log L(\log_2 N)^2)$ in case of using bitonic-like sorting and is $O(r^2 N^2 L \log L)$ in case of using odd-even-like sorting, where L is the number of data distributed on each node and $f \leq r - 1$. In case of L=1, the time complexity of hypercube and petersen cube are the same as K. Efe *et al.* sorting algorithm which is without fault-tolerant capability. Complexity study shows that the proposed fault-tolerant sorting algorithm is near optimal.

References

[1] K. Batcher. "Sorting Networks and Their Applications," *Proc. AFIPS Spring Joint Computing Conf.*, 32:307–314, 1968.

[2] Chih-Yung Chang, Yuh-Shyan Chen, and Chun-Bo Kuo. "A Generalized Sorting Algorithm for Faulty Product Networks," *Technical Report, AU-TR9908, Department of Computer Information Science, Aletheia University*, 1999.

[3] Y. W. Chen. "The Design and Analysis of Fault-Tolerant Prefix Computation, Sorting and Embedding Algorithms on Hypercube," *PhD thesis, Graduate School of Management in National Taiwan University of Science and Technology*, 1999.

[4] 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.

[5] 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.

[6] D. R. Őhring and Dirk H. Hohndel. "Optimal Fault-Tolerant Communication Algorithms on Product Netowrks using Spanning Trees," *Proc. of sixth IEEE Symposium on Parallel and Distributed Processing*, pp. 188–195, Jan. 1994.

[7] J. P. Sheu, Y. S. Chen, and C. Y. Chang. "Fualt-Tolerant Sorting Algorithm on Hypercube Multicomputers," *Journal of Parallel and Distributed Computing*, 16:185–197, 1992.

[8] 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, 1996.

[9] S. R. Ohring and S. K. Das. "The Folded Peterson Cube Network: New Competitiors for The Hypercubes," *IEEE Transactions on Parallel and Distributed Systems*, 7(2):151–168, Feb. 1996.