A Reliable Sorting Algorithm on Hypercube Multicomputers

Yuh-Shyan Chen and Jang-Ping Sheu

Department of Computer Science & Information Engineering
National Central University, Chung-Li 32054, Taiwan, R.O.C.
sheujp@mbox.ee.ncu.edu.tw

Correspondence Address: Prof. Jang-Ping Sheu
Department of Computer Science & Information Engineering
National Central University, Chung-Li 32054, Taiwan, R.O.C.

November 10, 1993

Abstract

In this paper, we present an algorithm-based fault-tolerant technique, namely the median-splitting strategy, for designing a reliable sorting algorithm. Combining the median-splitting strategy with bitonic sorting algorithm, a reliable sorting algorithm is proposed on the hypercube multicomputers. By the strategies of duplicating data and rollback, the proposed algorithm can detect transient faults and automatically correct errors without any hardware modification. We also implement our algorithm on NCUBE/7 MIMD hypercube machines with 64 processors. The simulation results show that our sorting algorithm is reliable and cost-effective.

Keywords: Bitonic sort, fault tolerance, hypercubes, parallel processing, transient faults.

1. Introduction

With the advent of VLSI technology, hundreds even thousands of processors can be built in a multicomputers system. The systems with a large number of processors will incur faults more frequently than the single-processor systems. Designing reliable algorithms to ensure the correctness of output results and keep the degree of performance are important topics for the multicomputers in the presence of faults. Efficient sorting algorithms have generally been the fundamental components and factors of many scientific algorithms. Moreover, hypercube multicomputers recently become commercially available parallel machines. Thus, it is more meritorious and interesting to design a reliable sorting algorithm on the hypercubes.

Recently, many fault-tolerant schemes which comprise hardware and software strategies have been addressed under the assumption of permanent fault model [2] [5] [6] [15] [18] [19]. Particularly in the hardware strategies, notable is the result by Bruck, Cypher, and Ho [6]. They proposed the efficient fault-tolerant hypercube and mesh architectures with adding minimum redundancy processors and links based on the graph model. In contrast to hardware strategies, Özgüner proposed the maximum dimensional fault-free subcubes [14] method for tolerating two or more faulty processors. Bruck, Cypher, and Soroker [5] also proposed the tolerating faults technique in n-dimensional hypercubes by using the subcube partitioning. The approach is to partition a faulty hypercube into subcubes within each of them contains less faults. Sheu, Chen, and Chang [19] proposed similar subcube partitioning method for a fault-tolerant sorting algorithm on n-dimensional hypercubes that can tolerate up at most n - 1 faults. However, compared to permanent faults, it is more difficult to tolerate up transient faults since the transient faults occur more unpredictable and frequent. Tolerating transient faults thus is a more meaningful and practical research topic in the multicomputers.

The algorithm-based fault-tolerant encoded scheme has been first proposed by Huang [9] for the purpose of solving the transient faults. Banerjee [3] continually proposed many algorithm-based error detection schemes that detect transient faults and replace the faulty processors by spare processors in many applications. It is unreasonable to detect a transient fault and then replace it by a spare processor because that many faulty processors may produce correct results after those transient faults disappear. Recently, Yeh and Feng [20] proposed an algorithm-based fault-tolerant approach for the algorithm of matrix inversion. However, this approach just can correct a single fault and detect multiple faults. Conventional encoding scheme is difficult to handle the sorting problem on multicomputers when occurring transient faults in hardware.

Therefore, designing a reliable sorting algorithm that can tolerate up transient faults is our main focus on this study.

We assume that the communication channels are reliable. Huang and Abraham assumed the similar fault-model in [9]. This assumption is reasonable since effective error correcting schemes such as coding theory [7] [13] and alternative retry method [17] are proposed for tolerating faults in the communication lines and memories. The input/output latch registers of a processor are considered as parts of the communication circuitry since a fault in the latch register affects the data transfer but not the computation itself. Therefore, we only focus on fault tolerance of the computations of processors in the multicomputers.

In this paper, we propose a reliable sorting algorithm that can detect transient faults without any hardware modification and can correct errors by low degree of data duplication and replicated computations. The reliable sorting algorithm can rapidly detect faults occurring, skillfully recover from the computation failures and continually perform the sorting operations. Algorithm presented here thus is not the fail-stop fashion [11]. That is, the result of calculation is either completely correct, or the entire algorithm halts with an error condition. Batcher's [4] bitonic sorting algorithm has two fundamental operations, compare-exchange and merge-compare-exchange operations, without the fault-tolerant capability. We propose a strategy, namely the median-splitting strategy, to additionally achieve the fault tolerance. On the basis of the compare-exchange and merge-compare-exchange operations with the median-splitting strategy, the reliable sorting algorithm can be applied on the hypercube multicomputers. Besides, we implement our algorithm on NCUBE/7 MIMD hypercube machines with 64 processors. The simulation results show that our reliable sorting algorithm running on hypercubes has lower extra-overhead.

The rest of this paper is organized as follows. Some basic properties of bitonic sorting algorithm will be reviewed and a median-splitting strategy is proposed in Section 2. A reliable sorting algorithm is presented in Section 3. The implementation and performance analysis of the proposed algorithm are discussed in Section 4. The conclusions will be finally presented in Section 5.

2. Median-Splitting Strategy

In this section, we first describe the compare-exchange and merge-compare-exchange operations of the bitonic sorting algorithm [4] in Section 2.1. To achieve the fault tolerance on these two operations, a median-splitting method is then proposed to tolerate the transient faults. The reliable compare-exchange and merge-compare-exchange operations are then presented in Section 2.2.

2.1 Median-Splitting Method

We first recall the bitonic sorting algorithm [10] [16]. The key concept of the algorithm is recursively executing the compare-exchange and merge-compare-exchange operations on each pair of neighboring processors P and P' such that the first half smallest and sorted elements are located in processor P and the last half largest and sorted elements are located in processor P'. For example, given a bitonic sequence $\{b_1, b_2, \ldots, b_m\}$ such that there exists ascending subsequence $\{b_1, b_2, \ldots, b_{m/2}\}$ and descending subsequence $\{b_{m/2+1}, b_{m/2+2}, \ldots, b_m\}$, the two subsequences are located in processors P and P', respectively. After performing the compare-exchange operation, sequences CE_l and CE_h are then respectively located in processors P and P' as follows. Let

$$\begin{aligned} CE_l &= \{\min(b_1, b_{m/2+1}), \min(b_2, b_{m/2+2}), \ldots, \min(b_{m/2}, b_m)\} \\ &= \{ce_1, ce_2, \ldots, ce_{m/2}\} \\ &= \{\underbrace{ce_1 \le ce_2 \le \cdots \le ce_i}_{A=\{ce_1, ce_2, \ldots, ce_i\}} \le \underbrace{ce_{i+1} \ge ce_{i+2} \ge \cdots \ge ce_{m/2}}_{B=\{ce_{i+1}, ce_{i+2}, \ldots, ce_{m/2}\}}\} \end{aligned}$$

and

$$\begin{aligned} CE_h &= \{\max(b_1, b_{m/2+1}), \max(b_2, b_{m/2+2}), \ldots, \max(b_{m/2}, b_m)\} \\ &= \{ce_{m/2+1}, ce_{m/2+2}, \ldots, ce_m\} \\ &= \{\underbrace{ce_{m/2+1} \ge ce_{m/2+2} \ge \cdots \ge ce_{m/2+i}}_{A'=\{ce_{m/2+1}, ce_{m/2+2}, \ldots, ce_{m/2+i}\}} \le \underbrace{ce_{m/2+i+1} \le ce_{m/2+i+2} \le \cdots \le ce_m}_{B'=\{ce_{m/2+i+1}, ce_{m/2+i+2}, \ldots, ce_m\}}\}, \end{aligned}$$

where $i \le m/2$. It has been shown [4] that sequences CE_l and CE_h are also bitonic sequences. Let A and B, B' and A' respectively denote the ascending and descending subsequences from sequence CE_l and CE_h. Processor P then merges subsequences A and B to obtain sequential subsequence $\{b'_1, b'_2, \ldots, b'_{m/2}\}$. Similarly, processor P' merges subsequences A' and B' to obtain sequential subsequence $\{b'_{m/2+1}, b'_{m/2+2}, \ldots, b'_m\}$.

Unfortunately, these two operations will result in a faulty execution on multicomputers as soon as the processors occurred transient faults. It leads to a serious problem

that there exists an element $x \in CE_l$ in processor P' and an element $y \in CE_h$ in processor P. What is worse, the problem can further lead to the more serious condition of error propagation as long as processor P repeatedly executes these two operations on different neighboring processors. For the reason of preventing of error propagation and achieving the fault tolerance, the two operations should be further improved with fault tolerant capability. The reliable compare-exchange operation must be ensured to correctly separate the original sequence into smaller and larger subsequences which are respectively located in processors P and P'. In addition, the reliable merge-compare-exchange operation must be ensured to correctly merge each pair of subsequences into sequential sequence. In this paper, the median-splitting process is presented to achieve the reliable capability of these two operations.

In the following, we first describe how to find the $\lceil m/2 \rceil$th element of the original sequence and then separate the original sequence $\{b_1, b_2, \ldots, b_m\}$ into smaller and larger subsequences. The process of finding the median element for splitting the original sequence is called the median-splitting process in this paper. To introduce the median-splitting process, we first define the notations of sequence, sequence pair, and subsequence pair. Assume there exist sequences $G = \{g_1, g_2, \ldots, g_r\}$ and $H = \{h_1, h_2, \ldots, h_s\}$, where $g_1 \leq g_2 \leq \cdots \leq g_r$ and $h_1 \leq h_2 \leq \cdots \leq h_s$. Let $S[1,r]$ denote the *sequence* G and *sequence pair* $S([1,r],[1,s])$ denote the pair of sequence ($\{g_1, g_2, \ldots, g_r\}$,$\{h_1, h_2, \ldots, h_s\}$). More, let $S([i,j],[i',j'])$ denote the *subsequence pair* ($\{g_i, g_{i+1}, \ldots, g_j\}$, $\{h_{i'}, h_{i'+1}, \ldots, h_{j'}\}$) which respectively extract subsequence from sequence pair ($\{g_1, g_2, \ldots, g_r\}$,$\{h_1, h_2, \ldots, h_s\}$), for $1 \leq i \leq j \leq r$ and $1 \leq i' \leq j' \leq s$. For example, consider the sequence $G = \{g_1, g_2, g_3, g_4\} = \{1, 3, 7, 9\}$ and sequence $H = \{h_1, h_2, h_3, h_4, h_5, h_6\} = \{2, 4, 5, 6, 8, 10\}$. Sequence $S[1,4] = \{1, 3, 7, 9\}$ and sequence pair $S([1,4],[1,6]) = (\{g_1, g_2, g_3, g_4\}, \{h_1, h_2, h_3, h_4, h_5, h_6\}) = (\{1, 3, 7, 9\}, \{2, 4, 5, 6, 8, 10\})$ exist. Subsequence pair $S([2,3],[2,4])$ is $(\{g_2, g_3\}, \{h_2, h_3, h_4\}) = (\{3, 7\}, \{4, 5, 6\})$.

Splitting the original sequence pair into smaller disjoint subsequence pairs is the basis step of our median-splitting process. Without loss of generality, we assume that $i = 1, j = r, i' = 1$, and $j' = s$ for the original sequence pair $S([1,r],[1,s])$. It is straightforward that the original sequence pair $S([1,r], [1,s])$ can be recursively split into many disjoint subsequence pairs. We first consider the condition of splitting the sequence pair $S([1,r], [1,s])$ into two disjoint subsequence pairs. We may obtain two fully disjoint subsequence pair; one is $S([1,p], [1,q])$ and another is $S([p+1,r], [q+1,s])$ where $1 \leq p \leq r$ and $1 \leq q \leq s$. In other words, the sequence pair ($\{g_1, g_2, \ldots, g_r\}$, $\{h_1, h_2, \ldots, h_s\}$) can be split into two disjoint subsequence pairs ($\{g_1$,

g_2, ..., g_p}, {h_1, h_2, ..., h_q}) and ({g_{p+1}, g_{p+2}, ..., g_r}, {h_{q+1}, h_{q+2}, ..., h_s}). There possibly exist many disjoint subsequence pairs. In what follows, we state how to find the feasible subsequence pairs. Let $G.H$ denote the merged sequence resulting from merging sequences G and H, and (x, y) denote the *median-index pair* of $S([1, r], [1, s])$ such that all elements in sequence $S([1, x], [1, y])$ are the first $\lceil (r + s)/2 \rceil$ smallest elements of $G.H$, and all elements in $S([x + 1, r], [y + 1, s])$ are the remainder larger elements of $G.H$ for $1 \leq x \leq r$ and $1 \leq y \leq s$.

Recalling the above example, we obtain the merged sequence $G.H$ = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. The *median-index pair* (x, y) is (2, 3). This is because that elements g_1, g_2, h_1, h_2, and h_3 are the first half smallest elements in sequence $G.H$ as shown in Fig. 1. Hence, one of the disjoint subsequence pairs is $S([1, 2], [1, 3])$ = ({g_1, g_2}, {h_1, h_2, h_3}) = ({1, 3}, {2, 4, 5}) and the other one is $S([3, 4], [4, 6])$ = ({g_3, g_4}, {h_4, h_5, h_6}) = ({7, 9}, {6, 8, 10}).

In general, if we have known the valid *median-index pair* (x, y) of $S([i, j], [i', j'])$, we may correctly separate the original sequence into two parts $S([i, x], [i', y])$ and $S([x + 1, j], [y + 1, j'])$. For the presentation of this scheme, we formally introduce the following definition.

Definition 1 : *Median-splitting* and *median-index pair*

The *median-splitting* is the process of splitting the $S([i, j], [i', j'])$ into two disjoint subsequence pairs $S([i, x], [i', y])$ and $S([x+1, j], [y+1, j'])$, where (x, y) is the *median-index pair* of sequences {g_i, g_{i+1}, ..., g_j} and {$h_{i'}$, $h_{i'+1}$, ..., $h_{j'}$} such that the elements of sequence $S([i, x], [i', y])$ are smaller than the elements of sequence pair $S([x+1, j], [y+1, j'])$ and $(x - i + 1) + (y - i' + 1) = \lceil \frac{j-i+j'-i'}{2} + 1 \rceil$.

□

In order to successfully separate $S([i, j], [i', j'])$ into two parts and detect whether the obtained *median-index pair* (x, y) is valid or not, an invoked self-checking routine is described as follows. By definition 1, we know that if the number of elements in $S([i, x], [i', y])$ is half of the number of elements in $S([i, j], [i', j'])$ and each element in $S([i, x], [i', y])$ is smaller than elements in $S([x + 1, j], [y + 1, j'])$, the *median-index pair* (x, y) is valid. In special, because that all elements in sequence pair $S([i, x], [i', y])$ are smaller than elements in $S([x+1, j], [y+1, j'])$, we just need to check whether the minimum value in $S([x+1, j], [y+1, j'])$ is no less than the maximum value in $S([i, x], [i', y])$ or not. That is, whether the criteria $g_x \leq h_{y+1}$ and $h_x \leq g_{y+1}$ hold or not. Therefore, provided that sequences $S[i, j]$ and $S'[i', j']$ in sequence pair $S([i, j], [i', j'])$ are the correct sequences, we can successfully check a given (x, y) is valid *median-index pair* or

not by following self-checking routine.

/* Checking the value (x, y) is the valid *median-index pair* of $S([i, j], [i', j'])$ or not.
If the calculated (x, y) is valid then return TRUE. Otherwise, return FALSE. */
Function self-checking-routine((x, y), $S([i, j], [i', j'])$)
if $(x + y - (i + i') + 2 = \lceil \frac{j-i+j'-i'}{2} + 1 \rceil$ **and** $g_x \leq h_{y+1}$ **and** $h_y \leq g_{x+1})$ **return** TRUE;
else return FALSE;

The alternative retry statements are used here to guarantee the self-checking-routine function with high reliability. That is, the recalculation of the condition ($\lceil \frac{j-i+j'-i'}{2} + 1 \rceil = x + y - (i + i') + 2$ **and** $h_{y+1} \geq g_x$ **and** $g_{x+1} \geq h_y$) is necessary for the sake of reliability. It is assumed throughout this paper that the period time between two of consecutive transient faults is larger than the computation time of the alternative retry statements. Therefore, we have enough capability to check whether there exists fault or not by using the replicated calculation method. According to the assumption, the self-checking routine is considered as reliable. After applying the reliable self-checking routine, it can be known that whether the calculated *median-index pair* is valid or invalid. Referring the above example, the valid *median-index pair* $(2, 3)$ is detected by applying the self-checking-routine($(2, 3)$, $S([1, 4], [1, 6])$). This is because that the conditions $(x + y - (i + i') + 2 = \lceil \frac{j-i+j'-i'}{2} + 1 \rceil = \lceil \frac{4-1+6-1}{2} + 1 \rceil = 5$ and $g_2 = 3 < h_4 = 6$ and $h_3 = 5 < g_3 = 7)$ hold. Once the correct *median-index pair* (x, y) found, the (x, y) is used to achieve the reliable compare-exchange and merge-compare-exchange operations with median-splitting technique in a recursive manner which will be introduced in the next subsection.

2.2 Reliable Compare-Exchange and Merge-Compare-Exchange Operations

In the previous subsection, a median-splitting method is proposed to separate an original sequence pair into two subsequence pairs by a valid *median-index pair*. Based on the median-splitting technique, the reliable compare-exchange and merge-compare-exchange operations will be presented in this subsection.

Consider an n-dimensional hypercube Q_n that has 2^n processors with address space $\{b_{n-1}b_{n-2} \ldots b_0\}$. Let processors P and P' be a pair of neighboring processors in hypercube Q_n and perform the reliable compare-exchange operation along dimension k, where $0 \leq k \leq n - 1$. The compare-exchange operation with median-splitting technique can correctly separate the original sequence into smaller and larger subsequences and are respectively located in P and P' even if some faulty computations occurring.

The proposed technique has the fault-tolerant capabilities of detecting transient faults, correcting incorrect values, and preventing error propagation.

In the following, we describe how our compare-exchange operation achieves the fault-tolerant capabilities. Assume that neighboring processors P and P' have sequence $S[1, m/2]$ and $S'[1, m/2]$, respectively. Processor P then sends its allocated sequence $S[1, m/2]$ to processor P' and receives sequence $S'[1, m/2]$ from processor P'. Similarly, processor P' also sends its allocated sequence $S'[1, m/2]$ to processor P and receives sequence $S[1, m/2]$ from processor P. As a result, both processors P and P' have the same sequence pair $S([1, m/2], [1, m/2])$. Processors P and P' independently calculate its own *median-index pair* (x, y) of the sequence pair $S([1, m/2], [1, m/2])$ and then detect the validation of the (x, y) for preventing any transient fault occurring during the period of finding the *median-index pair* (x, y). The operation of automatic detecting transient faults is achieved by performing the function self-checking-routine((x, y), $S([1, m/2], [1, m/2])$). The strategy of correcting faults is to compare each *median-index pair* with another one calculated by the neighboring node. Three cases are discussed in follows. First, if both the *median-index pairs* are valid, nothing needs to do. Second, if one of the *median-index pairs* is invalid, the valid *median-index pair* can mask the invalid *median-index pair*. The third, when both the *median-index pairs* are invalid, the routine of finding the *median-index pair* (x, y) will be performed again. After finding the correct *median-index pair* (x, y), processor P correctly reserves the smaller sequence pair $S([1, x], [1, y])$ and P' reserves the larger subsequence pair $S([x + 1, m/2], [y + 1, m/2])$. The purpose of prevention of error propagation is also achieved by correctly performing each reliable compare-exchange operation. The procedure of the compare-exchange operation with median-splitting technique (CEMS) is outlined as follows.

Procedure CEMS(P, P', k)

Input: Assume processors P and P' are neighboring processors along dimension k in n-dimensional hypercubes, where $0 \leq k \leq n - 1$. Processors P and P' have sequences $S[1, m/2]$ and $S'[1, m/2]$, respectively.

Output: The smaller subsequence pair $S([1, x], [1, y])$ is located in processor P and the larger subsequence pair $S([x+1, m/2], [y+1, m/2])$ is located in processor P', where (x, y) is the correct *median-index pair* of the sequence pair $S([1, m/2], [1, m/2])$.

Step 1: /* Duplicating data */

Processor P (P') sends its allocated sequence $S[1, m/2]$ ($S'[1, m/2]$) to its neighboring processor P' (P) and receives sequence $S'[1, m/2]$ ($S[1, m/2]$) from the neighboring processor P' (P).

Step 2: /* Finding *median-index pair* */
Both processors P and P' calculate a *median-index pair* (x, y) of the sequence pair $S([1, m/2], [1, m/2])$ independently.

Step 3: /* Detecting faults */
Each processor sets a variable *flag* by performing the operation

$$flag = \text{self-checking-routine}((x, y), S([1, m/2], [1, m/2])).$$

/* If *flag* = TRUE, the (x, y) is valid; else (x, y) is invalid. */

Step 4: /* Correcting faults */

4-1: Processor with *flag* = TRUE (*flag* = FALSE) sends the valid *median-index pair* (x, y) (the "Invalid" message) to its neighboring processor.

4-2: **If** (processor with *flag* = FALSE and receives valid (x, y) from neighboring processor) **then** corrects its invalid *median-index pair* and then sets variable *flag* to be TRUE; **else** do nothing.
If (processor with *flag* = TRUE) **then** discard the received message.

4-3: **If** (the processor's *flag* is still equal to FALSE) **then goto Step 2**.

Step 5: /* Splitting sequence pair */
Processor P keeps the index pairs $(1, x)$ and $(1, y)$ for reserving the smaller sequence pair $S([1, x], [1, y])$ and processor P' keeps the index pairs $(x+1, m/2)$ and $(y+1, m/2)$ for reserving the larger subsequence pair $S([x+1, m/2], [y+1, m/2])$.

Example 1 illustrates how processors P and P' in Q_n perform the CEMS procedure along dimension k. Initially, processors P and P' respectively have sequences $S[1, 4]$ = {2, 4, 13, 22} and $S'[1, 4]$ = {3, 7, 24, 31} as shown in Fig. 2(a). After applying step 1 of the CEMS procedure, processors P and P' duplicate the same sequence pair $S([1, 4], [1, 4])$ = ({2, 4, 13, 22}, {3, 7, 24, 31}) as shown in Fig. 2(b). In step 2, processor P and P' respectively find its own *median-index pairs* $(2, 2)$ and $(2, 3)$. In the step 3, only the processor P finds the valid *median-index pair* $(2, 2)$ as shown in Fig. 2(c). After applying the step 4, the correct *median-index pair* $(2, 2)$ can mask the invalid *median-index pair* of processor P' as shown in Fig. 2(d). Finally, applying the step 5,

processor P reserves the smaller subsequence pair $S([1,2],[1,2]) = \{(2,4),(3,7)\}$ and processor P' reserves the larger subsequence pair $S([3,4],[3,4]) = \{(13,22),(24,31)\}$ as shown in Fig. 2(e).

The following lemma states that the proposed CEMS procedure is reliable.

Lemma 1 : The CEMS procedure is reliable.

Proof: In the step 1, the operation of duplicating the sequence $S[1,m/2]$ or $S'[1,m/2]$ between neighboring processors is reliable since the communication lines and memories are reliable under our assumption. Transient faults possibly occur in step 2. In the step 3, the reliable self-checking-routine((x,y), $S([1,m/2],[1,m/2])$) function is applied to detect whether the error exist or not. Since that the communication lines and memories are reliable, the correct *median-index pair* (x,y) can be successfully sent to neighboring processor for updating the invalid *median-index pair* $(\hat{x},\hat{y})$. Hence, transient faults occurring in step 2 can be corrected by the reliable operations of the step 3 and step 4. It can successfully split the sequence pair into two subsequence pairs in step 5 because that the correct *median-index pair* has been obtained. This completes the proof that the CEMS procedure is reliable since all steps are reliable.

□

We now analyze the time complexity of the CEMS procedure. Let symbol $t_{s/r}$ denote the cost of sending or receiving an element between two neighboring processors and symbol t_c denote the time cost of a unit computation. In step 1, the time cost of processors P and P' respective sending its allocated sequences $S[1,m/2]$ and $S'[1,m/2]$ to each other is $O(m)t_{s/r}$. Akl proposed a *two-sequence-median* algorithm [1] to find the indices of the median pair of sequences $\{a_1, a_2, \ldots, a_r\}$ and $\{b_1, b_2, \ldots, b_s\}$ with time complexity $O(c_1 + c_2\log(\min\{r,s\}))$. Thus, we need time $O(\log m/2)t_c$ for each processor to find a *median-index pair* (x,y) of $S([1,m/2],[1,m/2])$ in step 2. The time cost of steps 3, 4-1, 4-2, and 5 is constant time. If the **goto** statement in step 4-3 of CEMS procedure performs r loops of steps 2, 3, 4-1, and 4-2, the total time cost T_{CEMS} of the compare-exchange operation with median-splitting technique is

$$T_{CEMS} = O(m)t_{s/r} + O(r\log m/2)t_c.$$

In the case that a constant number of transient faults occurs in total, the time complexity of our reliable compare-exchange operation is $O(m)$.

After applying the reliable compare-exchange operation between neighboring processors P and P', the smaller subsequence pair $S([1,x],[1,y])$ is located in processor P and the larger subsequence pair $S([x+1,m/2],[y+1,m/2])$ is located in processor P'.

Continually, subsequence pairs $S([1,x],\,[1,y])$ and $S([x+1,m/2],\,[y+1,m/2])$ must be respectively merged into sequential sequences $S'[1,x+y]$ and $S'[x+y-1,m]$. In the following, we describe our reliable merge-compare-exchange operation with median-splitting technique. Initially, the sequence pair $S([i,j],[i',j'])$ can be split into $S([i,x],[i',y])$ and $S([x+1,j],[y+1,j'])$ by *median-index pair* (x,y) of sequence pair $S([i,j],[i',j'])$. The smaller sequence pair $S([i,x],[i',y])$ will be recursively split into two disjoint subsequence pairs $S([i,x'],[i',y'])$ and $S([x'+1,x],[y'+1,y])$ by a valid *median-index pair* (x',y'). For the larger sequence pair $S([x+1,j],[y+1,j'])$, if the largest element g_j in sequence $\{g_i,\,g_{i+1},\,\ldots,\,g_j\}$ is equal to the g_x, we move all elements of $\{h_{y+1},\,\ldots,\,h_{j'}\}$ onto the resultant sequence $S'[x+y+1,j+j']$. Similarly, if the largest element $h_{j'}$ in sequence $\{h_{i'},\,h_{i'+1},\,\ldots,\,h_{j'}\}$ is equal to the h_y, all elements of $\{g_{x+1},\,\ldots,\,g_j\}$ are moved onto the $S'[x+y+1,j+j']$. Otherwise, the larger sequence pair $S([x+1,j],[y+1,j'])$ will also be recursively split into two disjoint subsequence pairs $S([x+1,x''],[y+1,y''])$ and $S([x''+1,j],[y''+1,j'])$ by a valid *median-index pair* (x'',y''). Recursively applying the above median-splitting process on each new split subsequence pair, in final, the length of each sequence of the split subsequence pair is equal to 1. Let $S([k,k],[l,l])$ be the final split subsequence pair, where $i \le k \le j$ and $i' \le l \le j'$. Then, if condition $g_k \le h_l$ is satisfied, we move $\{g_k,\,h_l\}$ to $S'[k+l-1,k+l]$; else move $\{h_l,\,g_k\}$ to $S'[k+l-1,k+l]$. The merge-compare-exchange operation with median-splitting technique presented here is a divide-and-conquer scheme. We recursively partition the original sequence pair into a number of disjoint subsequence pairs with high reliability and then merge each subsequence pair successively and independently into a ascending sequence. After recursively splitting and merging the subsequence pairs into ascending subsequences, our reliable merge-compare-exchange operation will obtain different disjoint ascending subsequences. Note that, the operation of combining the subsequence pairs into sequential subsequences is naturally finished because we place the merged sequences into the suitable locations during the merging operation. Therefore, these disjoint subsequences can constitute the final merging result.

Our reliable merge-compare-exchange operation presented here has the capabilities of automatic detecting and correcting the transient faults. The operation of automatic detecting transient faults is achieved by the self-checking routine. Then, the strategy of correcting the fault is to recalculate the routine of finding the *median-index pair* (x,y) again when an invalid *median-index pair* $(\hat{x},\hat{y})$ is detected. A formal procedure for the reliable merge-compare-exchange operation with median-splitting technique (MCEMS) is given as follows.

Procedure MCEMS($S([i,j],[i',j'])$)

Input: Sequence pair $S([i,j],[i',j']) = (\{g_i, g_{i+1}, \ldots, g_j\}, \{h_{i'}, h_{i'+1}, \ldots, h_{j'}\})$.
Output: A merged sequential sequence $S'[i+i'-1,\ j+j']$ from the sequence pair $S([i,j],[i',j'])$.

Step 1:

(a) /* The length of each sequence of current sequence pair $S([i,j],[i',j'])$ is equal to 1 */
If $((i = j$ **and** $j = i)$ **and** $(i' = j'$ **and** $j' = i'))$ **then**

if $((g_i \leq h_{i'})$ **and** $(h_{i'} \geq g_i))$ **then** $S'[i+i'-1, j+j'] = \{g_i, h_{i'}\}$;
else $S'[i+i'-1, j+j'] = \{h_{i'}, g_i\}$;
return;

(b) /* Alternative retry statements */
If $((i \neq j$ **or** $j \neq i)$ **or** $(i' \neq j'$ **or** $j' \neq i'))$ **then goto Step 2**,
else goto Step 1.

Step 2: /* Finding *median-index pair* */
Find the *median-index pair* (x, y) of sequence pair $S([i,j],[i',j'])$.

Step 3: /* Detecting faults */
Check its own *median-index pair* (x, y) is valid or invalid by performing the self-checking-routine((x, y), $S([i,j],[i',j'])$). If the *median-index pair* (x, y) is invalid then **goto Step 2.**

Step 4: /* Merging for the smaller subsequence pair */
Recursively perform procedure MCEMS($S([i,x],[i',y])$) for the smaller subsequence pair $S([i,x],[i',y])$.

Step 5: /* Manipulating for the case when the largest element in sequences $\{g_i, g_{i+1}, \ldots, g_j\}$ or $\{h_{i'}, h_{i'+1}, \ldots, h_{j'}\}$ is smaller than or equal to the median element of sequence pair $S([i,j],[i',j'])$ */
If $(x = j$ **and** $j = x)$ **then** $S'[x+y+1, j+j'] = \{h_{y+1}, \ldots, h_{j'}\}$; **return**;
If $(y = j'$ **and** $j' = y)$ **then** $S'[x+y+1, j+j'] = \{g_{x+1}, \ldots, g_j\}$; **return**;

Step 6: /* Merging for the larger subsequence pair */
If $((x \neq j$ **and** $j \neq x)$ **and** $(y \neq j'$ **and** $j' \neq y))$ **then**

Recursively perform procedure MCEMS($S([x+1, j],[y+1, j'])$) for the larger subsequence pair $S([x+1, j],[y+1, j'])$,

else goto Step 5.

Example 2 illustrates the operation of the procedure MCEMS($S([1,5], [1,6])$). Let sequences $G = \{g_1, g_2, g_3, g_4, g_5\} = \{1, 3, 5, 6, 10\}$ and $H = \{h_1, h_2, h_3, h_4, h_5, h_6\} = \{2, 4, 7, 8, 9, 11\}$. First, we can find the correct *median-index pair* $(3, 2)$ of $S([1,5],[1,6])$ such that $S([1,5], [1,6])$ can be split into smaller subsequence pair $S([1,3],[1,2])$ and larger subsequence pair $S([4,5],[3,6])$. Recursively perform procedure MCEMS($S([1,3],[1,2])$) for the smaller subsequence pair such that sequence pair $S([1,3],[1,2])$ can be split again into $S([1,2],[1,1])$ and $S([3,3],[2,2])$. Finally, we can reliably merge each subsequence pair $S([1,1],[1,1])$, $S([2,2],[1,1])$, and $S([3,3],[2,2])$ into sequential sequence and locate in $S'[1,2] = \{S'_1, S'_2\} = \{1,2\}$, $S'[3,3] = \{S'_3\} = \{3\}$, and $S'[4,5] = \{S'_4, S'_5\} = \{4,5\}$, respectively. Similarly, $S'[6,7] = \{S'_6, S'_7\} = \{6,7\}$, $S'[8,8] = \{S'_8\} = \{8\}$, $S'[9,10] = \{S'_9, S'_{10}\} = \{9,10\}$, and $S'[11,11] = \{S'_{11}\} = \{11\}$ can be also obtained. Consequently, all of the disjoint subsequences $S'[1,2]$, $S'[3,3]$, $S'[4,5]$, $S'[6,7]$, $S'[8,8]$, $S'[9,10]$, and $S'[11,11]$ constitute a sequential sequence $S'[1,11] = \{1,2,3,4,5,6,7,8,9,10,11\}$.

The following lemma states that the proposed MCEMS procedure is reliable.

Lemma 2 : The procedure of the merge-compare-exchange operation with median-splitting technique (MCEMS) is reliable.

Proof: In the step 1 of MCEMS($S([i, j], [i', j'])$), if the length of sequences $S[i,j]$ and $S'[i',j']$ is equal to 1, we can reliably decide the smaller element of $S[i,j]$ and $S'[i',j']$ by the replicated calculation statement. Transient faults occurring in step 2, if exist, can be detected and corrected by the reliable operations of the step 3. In the step 3, the reliable self-checking routine is applied to detect whether the error exist or not. Hence, steps 2 and 3 assure that *median-index pair* (x, y) of the sequence pair $S([i,j],[i',j'])$ is valid. If the condition in the step 5 is satisfied, all elements of the larger subsequence pair need to move to the suitable locations. Since the communication channels are reliable, the operations of data movements can be done successfully. Otherwise, procedure MCEMS still need to recursively merge the larger subsequence pair $S([x+1, j], [y+1, j'])$ into the sequential sequence. Next, we prove that the steps 4 and 6 are reliable by the induction. The expansion of the MCEMS($S([i,j],[i',j'])$) procedure is a binary tree, which is called the *expansion tree* of the original MCEMS($S([i,j],[i',j'])$) procedure. Let n denote the level of the expansion tree.

Basis $n = 1$: Leaves in level 1 of the expansion tree will obtain a sequential subsequence by reliable merging the sequence pair $S([i,j],[i',j'])$ since the length of each sequence of $S([i,j],[i',j'])$ is equal to 1.

Hypothesis $n = k-1$: The MCEMS($S([i,x],[i',y])$) and MCEMS($S([x+1,j],[y+1,j'])$) are hypothesized reliable in the level $k-1$ of the expansion tree, where (x,y) is the valid *median-index pair* of sequence pair $S([i,j],[i',j'])$.

Induction $n = k$: In level k of the expansion tree, we claim that MCEMS($S([i,\ j],\ [i',j'])$) is reliable. MCEMS($S([i,\ j],\ [i',j'])$) is reliable by the hypothesis step. This is because that the MCEMS ($S([i,\ x],\ [i',y])$) and MCEMS($S([x+1,j],[y+1,j'])$) are the only two subtrees of MCEMS($S([i,\ j],\ [i',j'])$) in level k of the expansion tree.

We conclude that the merge-compare-exchange operation with median-splitting technique (MCEMS) is reliable. Hence, the Lemma 3 is proved by the induction hypothesis.

□

We now analyze the time complexity of the MCEMS procedure. The time cost of steps 1 and 3 is constant. In step 2, we need time $O(c_1 + c_2 \log(m/2))$ for finding a *median-index pair* (x,y) of $S([i,j],[i',j'])$ since $m/2 = i' - i + j' - j + 2$. Sequence pairs $S([i,x],[j,y])$ and $S([x+1,i'],[y+1,j'])$ of steps 4 and 6 respectively need time $O(c_1 + c_2\log(\min\{x',y'\}))$ and $O(c_1 + c_2\log(\min\{x'',y''\}))$ to find *median-index pairs* (x',y') and (x'',y''). It is obvious that both time cost are smaller or equal to $O(c_1 + c_2 \log(m/2^2))$ since $m/2^2 = x' + y' = x'' + y''$. Repeatedly split the new sequence pairs until the number of disjoint subsequence pairs is $2^{\log(m/2)-1}$. During the execution of our algorithm, if condition in step 5 holds, the execution time of step 5 is bounded in $O(m/2)$. Thus, the total time cost T_{MCEMS} can be measured by the following equation.

$$\begin{aligned} T_{MCEMS} &\leq O(\log(m/2) + 2\log(m/2^2) + \cdots + 2^{\log(m/2)-1}\log(m/2^{\log(m/2)})) + O(m/2) \\ &\leq O((m/2)\log(m/2) - \log(m/2) - (m/2)\log(m/2) + m) + O(m/2) \\ &\leq O(m - \log(m/2)) + O(m/2) \\ &= O(m) \end{aligned}$$

If constant number of transient faults occurs in total, the time complexity of our reliable merge-compare-exchange operation with median-splitting technique is $O(m)$.

3. Reliable Sorting Algorithm

In this section, we will describe our reliable sorting algorithm with median-splitting technique on the hypercube Q_n. The reliable compare-exchange and merge-compare-exchange operations with median-splitting technique have been developed in Section 2.2. The key concept of our reliable sorting algorithm proposed here is based on recursively performing these two reliable operations.

We now explain how the reliable sorting algorithm performs on the n-dimensional hypercube Q_n with 2^n processors. Assume each processor has $m/2 = \lceil M/2^n \rceil$ unsorted elements, where M is the number of total unsorted elements. First, each processor sorts its allocated elements independently to be sequential sequence $S[1, m/2]$ by using merge-sort algorithm. To prevent the transient fault occurring, each merge stage is achieved by performing the reliable MCEMS procedure. Next, each pair of neighboring processors P and P' recursively performs the compare-exchange and merge-compare-exchange operations with median-splitting technique along dimension k in hypercube Q_n, where $0 \leq k \leq n-1$. The reliable compare-exchange operation is achieved by performing the CEMS(P, P', k) procedure. After applying the CEMS procedure, each processor can successfully split the current sequence pair $S[1, m/2], [1, m/2])$ into two subsequence pairs $S([1, x], [1, y])$ and $S([x+1, m/2], [y+1, m/2])$ by the valid *median-index pair* (x, y). In addition, processors P and P' respective perform the procedures MCEMS($S([1, x], [1, y])$) and MCEMS($S([x+1, m/2], [y+1, m/2])$. The reliable merge-compare-exchange operation is thus achieved. Therefore, each processor can obtain the sequential sequence in the ascending order. These two reliable operations can be recursively applied and, in final, we obtain the sorted elements located on hypercube Q_n in the the processors' address order.

Our reliable sorting algorithm on hypercube (RSH) is introduced as follows.

Reliable Sorting Algorithm on Hypercube:

Input: A hypercube Q_n contains M unsorted elements. Each processor of Q_n has $m/2 = \lceil M/2^n \rceil$ elements. The address of each processor is $b_{n-1}b_{n-2}\ldots b_0$.

Output: Elements are sorted in ascending order and located on processors of Q_n by the processors' address order.

Step 1: /* Reliable merging-sort operation */

Each processor of Q_n sorts its elements to a sequential sequence $S[1, m/2]$ in ascending order by applying the merging-sort algorithm, which is based on the reliable MCEMS procedure.

Step 2: **For** $i = 0, 1, \ldots, n-1$ **do Steps 3** through **5**.

Step 3: /* Assume $b_n = 0$ */
For each processor, let variable *mask* be equal to the value of bit b_{i+1} of the processor's address.
If ($mask \neq b_{i+1}$ **or** $b_{i+1} \neq mask$) **then goto Step 3**.

Step 4: For $k = i, i-1, \ldots, 0$ do **Step 5**.

Step 5: For each pair of neighboring processors P and P' along dimension k:

(a) /* Reliable compare-exchange operation */
If (processor P satisfies the condition ($mask = v_k$ **and** $v_k = mask$) **and** processor P' satisfies the condition ($mask \neq v_k$ **and** $v_k \neq mask$)) **then** perform procedure CEMS(P, P', k) such that processor P reserves the smaller sequence pair $S([1,x],[1,y])$ and processor P' reserves the larger subsequence pair $S([x+1,m/2],[y+1,m/2])$, where the (x,y) is a valid *median-index pair*; **else goto Step 5(a)**.

(b) /* Reliable merge-compare-exchange operation */
Processors P and P' respectively perform procedures MCEMS($S([1,x],[1,y])$) and MCEMS ($S([x+1,m/2],[y+1,m/2])$) to obtain an ascending sequence.

Example 3 is considered here to illustrate the operations of the reliable sorting algorithm running on the hypercube Q_3. The host evenly distribute 32 unsorted elements to 8 processors of Q_3. By applying step 1 of the proposed algorithm, each processor sorts the assigned 4 unsorted elements to sequential sequence $S[1,4]$ in ascending order by the reliable merge-sort algorithm as shown in Fig. 3(a). In Fig. 3(b), each processor duplicates 4 elements to its neighboring processor along dimension 0. Then, both the neighboring processors have the same sequence pair $S([1,4],[1,4])$. Processors 0, 1, 3, and 4 find the valid *median-index pairs* $(2,2)$, $(2,2)$, $(2,2)$, and $(3,1)$, respectively. Processors 2, 5, 6, and 7 find the invalid *median-index pairs* $(2,1)$, $(2,3)$, $(1,3)$, and $(2,3)$, respectively. Then, processors 0, 1, 3, and 4 respectively send its valid *median-index pair* to its neighboring processors 1, 0, 2, and 5 along dimension 0. Similarly, processors 2, 5, 6, and 7 send the "Invalid" message to its neighboring processors along dimension 0. Processors 2 and 5 thus respective mask its invalid *median-index pairs* $(2,3)$ and $(2,1)$ by the valid *median-index pair* $(2,2)$ and $(1,3)$ as shown Fig 3(c). Fig. 3(d) depicts that processors 0, 3, 4, and 7 (1, 2, 5, and 6) reserve the smaller (larger) subsequence pair. Finally, each processor merges its subsequence pair into one sequence as shown in Fig. 3(e). Repeatedly performing the step 2 through step 5, all the elements can be reliably sorted in an ascending order according to the processor's

address order as shown in Fig. 3(f).

Theorem 1 : The sorting algorithm on hypercube (RSH) is reliable.

Proof: The merge-sorting algorithm of step 1 is reliable because that each merging step is reliable proved in the Lemma 2. The *For* statements in steps 2 and 4 are reliable. There exist register store operation and register addition operations in the *For* statements. The register addition operation can be achieved reliable by the alternative retry method and the register store operation is reliable by the previous assumption. Thus, the step 3 is correct after performing the alternative retry statements. Lemma 1 proves the procedure of the compare-exchange operation with median-splitting technique (CEMS) of step 5(a) is reliable. Similarly, Lemma 2 shows the procedure of the merge-compare-exchange operation with median-splitting technique (MCEMS) of step 5(b) is reliable. Consequently, the sorting algorithm on hypercube is reliable since all steps of the algorithm are reliable.

□

The derivation of total time cost T_{RSH} of the proposed reliable sorting algorithm is described as follows. The worst case of time cost for reliable merging-sort operations in step 1 is $O(m\log\lceil m/2\rceil)t_c$. The total time cost of step 5(a) for performing the CEMS(P, P', k) procedure is $O(m)t_{s/r} + O(\log m/2)t_c$. The time cost of step 5(b) in the worst case for performing the procedure MCEMS($S([1, m/2], [1, m/2])$) is $O(m)t_c$. Steps 2 and 4 of the proposed algorithm perform $\frac{n(n+3)}{2}$ loops of steps 5(a) and 5(b). The total time cost T_{RSH} of the proposed reliable sorting algorithm without fault occurring in the worst case is

$$\begin{aligned} T_{RSH} &= O((m\log\lceil m/2\rceil)\ t_c + \tfrac{n(n+3)}{2}\ (O(m)t_{s/r} + O(\log m/2)t_c + O(m)t_c) \\ &= O(\max(m\log m,\ n^2 m)). \end{aligned}$$

If the constant number, in total, of transient faults occur during running the RSH algorithm, the time complexities of the reliable compare-exchange and merge-compare-exchange operations are $O(m)t_{s/r} + O(r\log m/2)t_c$ and $O(m)t_c$, respectively. Thus, the time cost of our reliable sorting algorithm is still $O(\max(m\log m,\ n^2 m))$. The time complexity of our reliable sorting algorithm is the same as the time complexity of the bitonic sorting algorithm on hypercube [10] [16].

Note that, some modifications are needed to apply this reliable algorithm to the PRAM model. Each processor of the PRAM model can read the $m/2$ elements simultaneously and avoid the operation of the duplicating data in the CEMS procedure. Thus, we can omit the operation of duplicating data in the CMES procedure. The time cost of the CEMS and MCEMS procedures are thus $O(r\log m/2)t_c$ and $O(m)t_c$,

respectively. Therefore, the time complexity of the reliable sorting algorithm on the PRAM model is also $O(\max(m\log m, n^2 m))$.

4. Experimental Results

In this section, we describe the implementation of the proposed reliable sorting algorithm on an NCUBE/ 7 MIMD hypercube machine with 64 processors each contains 512 K bytes of local memory. Simulation here mainly compares the execution time of our sorting algorithms with and without fault tolerance.

The sorting algorithm without fault tolerance can be achieved by omitting steps 3 and 4 of the CEMS procedure, step 3 of the MCEMS procedure, and the replicated statements in alternate-retry scheme. Our proposed sorting algorithm without fault tolerance is an novel algorithm and is simulated on the 6-dimensional hypercubes. In our simulation, the number of data elements is ranging from 2048 to 20480. The execution result of our algorithm without fault tolerance is depicted in Fig. 4 by thick line with solid points. The reliable sorting algorithm has been simulated on the 6-dimensional hypercubes. For illustrating the capability of fault tolerance, we assume that the number of repeatedly executing the steps 3 and 4 of CEMS procedure and step 3 of MCEMS procedure presents the degree of occurring transient faults. Here, we denote the number of recalculation by r. The larger value r is, the more times occurring transient faults will be. In our simulation, we estimate the extra execution time under the value of r are 1, 2, 3, and 4. The simulation results of our reliable sorting algorithm with different value of r, ranging from 1 to 4, are depicted in Fig. 4 by thin lines. In Fig. 4, the execution time of our reliable algorithm with $r = 1$, $r = 2$, $r = 3$, and $r = 4$ is shown to be slightly larger than sorting algorithm without fault tolerance running on the Q_6. The percentage overhead of the execution time in Fig. 5. is used to illustrate the degree of extra execution time of our reliable sorting algorithm. Let the execution time of our reliable sorting algorithm be A. Let the execution time of our sorting algorithm without fault tolerance be B. Then, the value of the percentage overhead is evaluated by $(A-B)/A$. The percentage overhead of our reliable algorithm with $r = 1$, $r = 2$, $r = 3$, and $r = 4$ is illustrated in Fig. 5. As a result, more the number of data elements have been sorted, the lower percentage overhead is obtained by our reliable sorting algorithm with fixed value of r. For example, when the number of data elements is 20480, the percentage overhead of our reliable algorithm with $r = 1$ is near 20%. This concludes that our method is a truly low overhead reliable sorting algorithm.

5. Conclusions

In this paper, an algorithm-based fault-tolerant technique, namely the median-splitting strategy, is presented for designing a reliable sorting algorithm. Based on the median-splitting strategy, the reliable compare-exchange/merge-compare-exchange operations with median-splitting technique are proposed for the purpose of tolerating the transient faults. Combining these two reliable operations with the bitonic sorting algorithm, a reliable sorting algorithm is then proposed on the hypercube multicomputers. The algorithm presented here has the capabilities of detecting transient faults, correcting incorrect values, and preventing error propagation. The reliable algorithm can be easily extended to apply to other distributed memory systems such as the mesh multiprocessors or the PRAM model. Besides, we present experimental results on NCUBE/7 MIMD hypercube machines with 64 processors indicating that our fault-tolerant sorting algorithm has low extra-overhead.

References

[1] S. G. Akl, The Design and Analysis of Parallel Algorithms, *Prentice-Hall International Editions,* 1989, pp. 74-76.

[2] M. S. Alam and R. G. Melhem, "An Efficient Modular Spare Allocation Scheme and Its Application to Fault Tolerant Binary Hypercubes," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 1, Jan. 1991, pp. 117-126.

[3] P. Banerjee and J. T. Rahmeh, "Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor," *IEEE Transactions on Computers*, Vol. 39, No. 9, Sep. 1990, pp. 1132-1145.

[4] K. Batcher, "Sorting Networks and Their Applications," *Proc. 1968 Spring Joint Comput. Conf.*, Vol. 32. Reston, VA: AFIPS Press, 1968, pp. 307-314.

[5] J. Bruck, R. Cypher, and D. Soroker, "Tolerating Faults in Hypercubes Using Subcube Partitioning," *IEEE Transactions on Computers*, Vol. 41, No. 5, May 1992, pp. 599-605.

[6] J. Bruck, R. Cypher, and C. T. Ho, "Efficient Fault-Tolerant Mesh and Hypercube Architectures," *Int. Symp. Fault-Tolerant Computing*, 1992, pp. 162-169.

[7] C. L. Chen, "Symbol Error-Correcting Codes for Computer Memory Systems," *IEEE Transactions on Computers*, Vol. 41, No. 2, Feb. 1992, pp. 252-256.

[8] P. F. Corbett and Isaac D. Scherson, "Sorting in Mesh Connected Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, Sep. 1992, pp. 626-632.

[9] K. H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, Vol. C-33, No. 6, June 1984, pp. 518-528.

[10] S. L. Johnsson, "Combining Parallel and Sequential Sorting on a Boolean n-cube," *Proc. 1984 International Conference on Parallel Processing*, 1984, pp. 21-24.

[11] B. M. McMillin and Lionel M. Ni, "Reliable Distributed Sorting Through the Application-Oriented Fault Tolerance Paradigm," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 4, July 1992, pp. 411-420.

[12] V. P. Nelson "Fault-Tolerant Computing: Fundamental Concepts," *Computer*, July 1990, pp. 19-25.

[13] D. Nikolos and A. Krokos, "Theory and Design of t-Error Correcting, k-Error Detecting and d-Unidirectional Error Detecting Codes with $d > k > t$," *IEEE Transactions on Computers*, Vol. 41, No. 4, Apr. 1992, pp. 411-419.

[14] F. Özgüner and C. Aykanat, "A Reconfiguration Algorithm for Fault Tolerance in a Hypercube Multiprocessor," *Information Processing Letters*, Vol. 29, No. 5, Nov. 1988, pp. 247-254.

[15] D. A. Rennels, "On Implementing Fault-Tolerance in Binary Hypercubes," *Int. Symp. Fault-Tolerant Computing*, 1986, pp. 344-349.

[16] S. R. Seidel and L. R. Ziegler, "Sorting on Hypercubes," *Proc. of the Second Conference on Hypercube Multiprocessors*, 1987, pp. 285-291.

[17] J. J. Shedletsky, "Error correction by Alternate-data Retry," *IEEE Transactions on Computers*, Vol. C-27, Feb. 1978, pp. 106-114.

[18] J. P. Sheu, "Fault-Tolerant Parallel k Selection Algorithm in n-cube networks," *Information Processing Letters*, Vol. 39, No. 2, July 1992, pp. 93-97.

[19] J. P. Sheu, Y. S. Chen, and C. Y. Chang, "Fault-Tolerant Sorting Algorithm on Hypercube Multicomputers," *Journal of Parallel and Distributed Computing*, Vol. 16, No. 2, Oct. 1992, pp. 185-197.

[20] Y. M. Yeh and T. Y. Feng, "Algorithm-Based Fault Tolerance for Matrix Inversion with Maximum Pivoting," *Journal of Parallel and Distributed Computing*, Vol. 14, No. 4, Apr. 1992, pp. 373-389.