# Fault-Tolerant Sorting in SIMD Hypercubes

Amitabh Mishra Y. Chang L. Bhuyan F. Lombardi

Department of Computer Science
Texas A&M University
College Station, TX 77843

## Abstract

*This paper considers sorting in SIMD hypercube multiprocessors in the presence of node failures. The proposed algorithm correctly sorts up to $2^n = N$ keys in a faulty SIMD hypercube of dimension n containing up to $n-1$ faulty nodes. The proposed fault-tolerant algorithm employs radix sort. We use a pair of flood dimensions which helps to route data around the faulty processors during the movement of data. If all the key values to be sorted belong to the range 0 to $M-1$, sorting can be accomplished efficiently in $O(log\ M * log\ N) + O(log^2 N)$ time.*

**Index Terms**: SIMD, Fault-tolerant, Parallel Sorting, Radix Sort.

## 1 Introduction

The hypercube architecture has received much attention in the literature [1, 2, 3]. The nCUBE/2 from Ncube, iPSC/2 and iPSC/860 from Intel and CM-2 from Thinking Machine are some of the commercial machines based on a hypercube structure [4, 5]. The bigger the size of the system, the higher is the probability that some processors and/or links may fail. Being able to operate in the presence of faults, therefore, is of paramount importance. Sorting is one of the most basic algorithms in computer science. Applications of sorting have been listed in [6]. To the best of our knowledge, there is no practical fault-tolerant sorting algorithm for SIMD hypercubes, and the best known MIMD reliable sorting algorithm tolerates just one node/ link failure in an $n$-cube, and has a $O(\log^2 N)$ complexity [7]. In [8], Leighton analyzes the fault-tolerance properties of several bounded-degree networks that are commonly used for parallel computation. In the proposed algorithm, we have focused on SIMD hypercubes with *activity control* [9]. Processors belonging to SIMD machines of this type can participate in a computation step or abstain from it based on a local condition. Our algorithm tolerates upto $n-1$ faults in an $n$-cube. In our fault-tolerant sorting algorithm, we have employed radix sort, which has been implemented with high performance for data sets with unknown (non-uniform) distributions [10]. Parallel sorting in hypercubes involves efficient data movement among processors. In the event of a failure of a processing element, the required information is not sent to its neighbor. In our fault-tolerant algorithm, we ensure that the neighboring node receives proper data by performing a so-called *flood* operation. After finding out two valid flood dimensions, we rearrange the order of the dimensions as follows. We assign dimension numbers 0 and 1 to the two selected flood dimensions. The remaining dimensions may be assigned numbers from 2 to $n-1$ in any order. We refer to the newly-numbered dimensions as *logical* dimensions. After the execution of the sorting algorithm, the sorted key values reside on PE's whose indices are from 0 to $N-1$ based on the logical dimensions.

[1] This research has been supported by TATP grant 999903-025.

## 2 The Hypercube Model - Notations and Definitions

We refer to a processing element as a $PE$. An $n$-dimension hypercube SIMD computer consists of $N = 2^n$ PE's, each of which has a local memory associated with it. A PE, together with its memory, is referred to as a *node*. Each node is connected directly to $n$ other nodes by virtue of links, i.e., each node has $n$ *neighbors*. Two nodes lying on opposite ends of a link have their addresses differing in exactly one bit. We refer to the $i$th PE as PE($i$). We specify a node PE($i$) by its binary representation, $i_{n-1}i_{n-2}...i_0$. We define $i^{(q)}$ as the number represented by $i_{n-1}i_{n-2}...i_{q+1}\bar{i}_q i_{q-1}...i_0$, where $\bar{i}_q$ is the complement of $i_q$. Hence, PE($i$) is directly connected to PE($i^{(q)}$).

The fault model that we have assumed for the faulty hypercube is as follows. There might be up to $n-1$ randomly located faulty nodes in the hypercube. All failures are fail-stop; that is, once failure occurs in a PE, it does not communicate with its neighbors in any way; it is totally disabled. In addition, it is assumed that a diagnostic process has been successfully executed so that information concerning the location of faults may be used when reconfiguring the network in order to circumvent them.

We define a *pseudo faulty* node as follows. SIMD multiprocessors carry out data movements along one dimension at a time. A faulty PE fails to send data to its neighbor. The neighboring PE now contains spurious information, and is called a pseudo faulty node. $f$ denotes the number of faulty PE's in the hypercube. The maximum number of faulty PE's that the proposed faul-tolerant algorithm tolerates is $(n-1)$, so $f \leq (n-1)$.

The notations employed in the algorithms are similar to those in [11].

## 3 A Sketch of the Algorithm

The input to the proposed algorithm is the key values stored in the $V$ registers of the PE's. By using the most-significant-bit (MSB) radix sort, $\log M$ steps are needed to complete the sorting process. Each step, corresponding to one bit of the values, takes $O(\log N)$ time. First, values with the MSB equal to 0 (1) are *ranked*, and *concentrated* into the front (rear) segment of the set of PE's. As a result, the set of PE's is partitioned into *regions*. At the end of each of the $\log M$ steps, each region is further partitioned into at most two subregions. Each region has an *st* (*en*) flag to indicate whether it is the starting (ending) PE of a region. Before executing the same procedure for subsequent bits, the *starting/ending* flag is set for each region starting/ending PE. Next, the index/rank of the region starting/ending PE is *broadcast* to all PE's in that region. This step is necessary for the so-called *rank adjustment* in different regions. Each of these steps is performed on the hypercube in $O(\log N)$ time, which causes the main sorting algorithm to be of $O(\log M * \log N)$ time complexity.

An example of how our proposed fault-tolerant algorithm operates is illustrated in Figure 1. We have a 3-cube having two faulty nodes, PE(2) and PE(4). The key values to be sorted are in the $V$ registers of the PE's. We assume that $M = 3$. First, we rank the values with the MSB equal to 0 (1), and concentrate them at the rear (front) of the set of

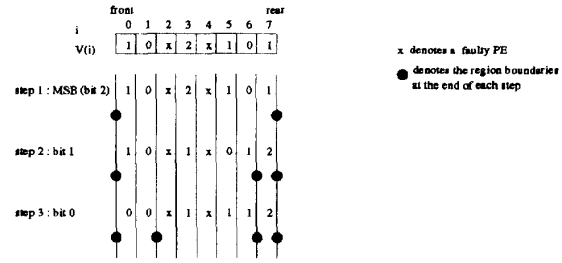


Figure 1: An example of the fault-tolerant algorithm for a 3-cube

PE's. In our algorithm, this step is performed by the $RANK$ procedure, the $BROADCAST$ procedure which broadcasts information to each record within a region about its final destination within that region, and the $CONCENTRATE$ procedure, which actually moves records to their final destinations within the region. Then we repeat the process for bit 1 and bit 0 of the key values. At the end of each step, two new regions are created, as illustrated by the new region boundaries in Figure 1. New boundaries are recognized by the $SET\text{-}FLAGS$ procedure. At the end of step three, we see that the key values have been sorted.

## 4 Flood Dimensions

A *flood dimension* (hereafter referred to as $FD$) is a dimension along which messages are transmitted from fault-free nodes to pseudo-faulty nodes in order for all pseudo-faulty nodes to receive the required information. Having one flood dimension, however, is not always enough. This is because, prior to flooding, there is at least one pair of nodes along the flood dimension in which the transmitting node is a faulty node and the receiving node is a pseudo-faulty one. This scenario is illustrated by the following example. We denote faulty nodes by 'x' and pseudo-faulty ones by 'y'. We wish to implement the algorithm given in Figure 2 on a 4-cube. Algorithm *First-example* is a small part of the *rank* algorithm we describe later, and serves to illustrate the inefficacy of having one flood dimension. The initial values of $S(i)$ are as indicated in Figure 3. Dimension 0 is the flood dimension for the example. The steps constituting the execution of algorithm *First-example* are shown in Figure 3. We observe that although flooding has been successful at each intermediate step, we finally reach a situation where there are two pairs of nodes, each of which consists of a faulty node and a pseudo-faulty node. It is

```
Algorithm First-example
/* n is the dimension of the hypercube */
1  T(i) = 1
   begin
2     for q = 0 to n − 1 do
3        T(i^(q)) ← S(i)
4        S(i) = S(i) + T(i)
      endfor
   end
```

Figure 2: Example to illustrate an x-y condition

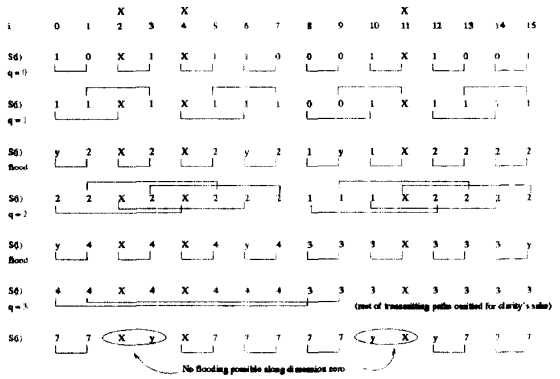

Figure 3: Flood Operation

clear that one more flood step along the flood dimension will not help the pseudo-faulty nodes receive the correct data. This is called an **x-y condition**.

Our example shows that having one flood dimension is insufficient in the worst case. We must, therefore, have at least a pair of FD's. The following theorem proves that having two FD's is sufficient in order to carry out flooding.

**Theorem 1.** *In an n-dimensional hypercube with at most (n-1) faults, it is always possible to find a pair of FDs such that there is never an x-y condition along both the dimensions simultaneously.*

**Proof:** Please refer to [6].

## 4.1 Finding an FD-pair

One pair of FDs is sufficient for our algorithm to work. In this subsection, we provide an efficient algorithm to find an FD-pair by eliminating all pairs of dimensions that can *not* possibly be FD-pairs. In the worst case, whenever two faulty PE's are located at a distance of two (links) from each other, the two dimensions of the 2-cube constituted by these faulty nodes may not be an FD-pair. In addition, if two faulty PE's are neighbors along a dimension, then that dimension can not obviously be used as a flood dimension. The algorithm to find an FD-pair is given in Figure 4. FAULT[0], FAULT[1],..., FAULT[$f-1$] represent the addresses of the faulty PE's. The procedure *distance* calculates the distance, in terms of number of links, between the two PE's in its argument. Lines 3 and 5 take $O(n)$ time because node addresses are of $n$ bits. If it is performed in parallel on $n$ nodes, the procedure takes $O(n^2)$, or $O(\log^2 N)$, time to execute.

```
Algorithm Find-FD-pair
/* n is the dimension of the hypercube */
begin
1     for j = 0 to f − 1 do
2        for k = 0 to f − 1 do
3           if distance(FAULT[j],FAULT[k]) ==1
4              discard(dimension1)
5           if distance(FAULT[j],FAULT[k]) ==2
6              discard(dimension1, dimension2)
7     endfor
end
```

Figure 4: Algorithm for finding an FD-pair

```
Algorithm Dimension-seq
begin
1  for i = 0 to 2n − 1
2     seq[i] = 0
3     if i%2 = 1 then seq[i] = ⌈i/2⌉
4     if i%2 = 0 and i ≥ 2 then
5        for i = F[0] to F[f − 1] do
6           for k = 1 to n do
7              if fault[i^(k)] = 1 then seq[2k − 1] = 1
8           endfor
9        endfor
10 endfor
end
```

Figure 5: Algorithm for finding the dimension sequence

## 4.2 Determining the sequence of dimensions for flooding

We perform the flood operation after each step of the fault-free algorithm. Flooding may take place along either of the two FDs belonging to the FD pair. Let us refer to the two FDs as dimensions 0 and 1. Usually, we flood along dimension 0 each time, unless we encounter an x-y condition along this dimension (in which case we choose dimension 1 to flood). We *predetermine* the dimension sequence for flooding to take place.

During the execution of the fault-free sorting algorithm, the dimension sequence for transferring information between neighbors is 0, 1, 2, 3, 4, ..., $n$-1. In the fault-tolerant algorithm, the dimension sequence, including flood dimensions, would be 0, 1, *, 2, *, 3, *, 4, *, 5, *, .... The *'s represent the flood dimensions,

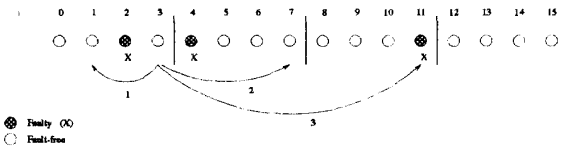


Figure 6: Example for determining a dimension sequence

and equal either 0 or 1. The procedure Dimension-seq (Figure 5) determines the dimension sequence. The array *fault* (maintained in the PE) records information about faulty PE's. $fault[i] = 1(0)$ means that $PE(i)$ is faulty (fault-free). The register $F$ contains a list of all fault-free PE's such that their neighbors along dimension 0 are faulty. The register *seq* contains the dimension sequence after the execution of the procedure. The way procedure Dimension-seq operates should be clear from the example shown in Figure 6. There are three faulty PE's in the 4-cube. Line 4 of the procedure Dimension-seq covers the 0-th dimension neighbors of faulty PE's, that is, nodes 3, 5 and 12. Consider node 3. The neighbors along dimensions 1, 2 and 3 are indicated in Figure 6. The neighbor of node 3 along dimension 3 is faulty, which will result in an x-y condition after execution along dimension 3 of the fault-tolerant algorithm (PE 11 is "x", PE 3 will be "y"). Hence, the flood dimension to be employed after dimension 3 of the fault-tolerant algorithm is dimension 1. After considering PE's 5 and 12 in a similar fashion, the dimension sequence is found to be 0, 1, 0, 2, 0, 3, 1.

## 5 Fault-tolerant Algorithm

The sorting algorithm is composed of four procedures: RANK, BROADCAST, CONCENTRATE and SET-FLAGS. For a detailed treatment of the proofs of correctness and examples of these algorithms, please see [6].

### 5.1 RANK

The rank of a PE is defined as the number of PE's having an *active* record preceding it. An active record is defined as a record that satisfies a certain condition such as, for example, having a particular bit of the value equal to 0 or 1. $p$ indicates bit $p$ of the value of the record, and $t$ is equal to 0 or 1. Hence, an active record is a record with $V(i)_p = t$. The ranking algorithm is given in Figure 7. Register $A$ contains information regarding whether a given PE is active

```
procedure RANK(p, t, varR)
1  R(i) = 0, F(i) = 0, T(i) = null
2  S(i) = 1(V(i)_p = t)
3  S(i) = 0(V(i)_p ≠ t)
 begin
4    for q = seq[0] to seq[2n − 2] do
5      T(i^(q)) ← S(i)
6      F(i) = T(i)(j = 0)
7      if 0-th or odd-numbered step/* fault-free step */
8        R(i) = R(i) + T(i)(i_q = 1 & T(i) ≠ null)
9        S(i) = S(i) + T(i)(T(i) ≠ null)
10       S(i) = null, R(i) = null(T(i) = null)
11     endif
12     if even-numbered step & greater than 0 /* flood step */
13       S(i) ← S(i^(q))(S(i) = null)
14       R(i) ← R(i^(q)), R(i) = R(i) − A(i) (i_q = 0 & q = 0 &
  S(i) = null)
15       R(i) ← (R(i^(q))+A(i^(q))) (i_q = 1 & q = 0 & S(i) = null)
16       R(i) ← R(i^(q)), R(i) = R(i) − A(i) − A(i + 1) (i_q = 0 &
  q = 1 & S(i) = null)
17       R(i) ← R(i^(q)), R(i) = R(i)+A(i^(q))+A(i^(q)+1)(i_q = 1
  & q = 1 & S(i) = null)
18     endif
19   endfor
 end
```

Figure 7: Fault-tolerant RANK algorithm

or not. $A[i] = 1(0)$ implies that the PE is active(not active). $F$ is the register that indicates the presence of a faulty neighbor along dimension 0. $F(i) = null$ means that $i$'s neighbor along dimension 0, is faulty. Register $S(i)$ has a value of 1 if $PE(i)$ is active, 0 if it is not active, and null if it is pseudo-faulty. Register $R$ contains the rank of each PE.

An example of the RANK algorithm is given in Figure 8. The register *seq*, in this case, contains the dimension sequence 0, 1, 0, 2, 0, 3, 1. In other words, we employ FD 0 to flood after executing the algorithm along dimensions 1 and 2, and FD 1 after dimension 3. The register $R$, as indicated at the bottom row of the figure, contains the correct rank in each fault-free PE.

### 5.2 BROADCAST

A *region* is defined as a maximal set of consecutive PE's having the same value in the $p$th bit of $V(i)$, where $0 \leq p \leq n$. Broadcasting is an operation that distributes the data in the region starting/ending PE to all the PE's belonging to that region. $a = 0(1)$ implies the selection of the region starting(ending) PE, while $b = 0(1)$ indicates that it is the index(rank) which must be broadcast. $st(i) = 1$ implies that $PE(i)$ is the starting PE of a region, while $en(i) = 1$ means that $PE(i)$ is the ending PE of a region.

The procedure to perform the broadcast operation is given in Figure 9. Registers $C$ and $D$ denote the transmitting and the result registers, respectively. Af-

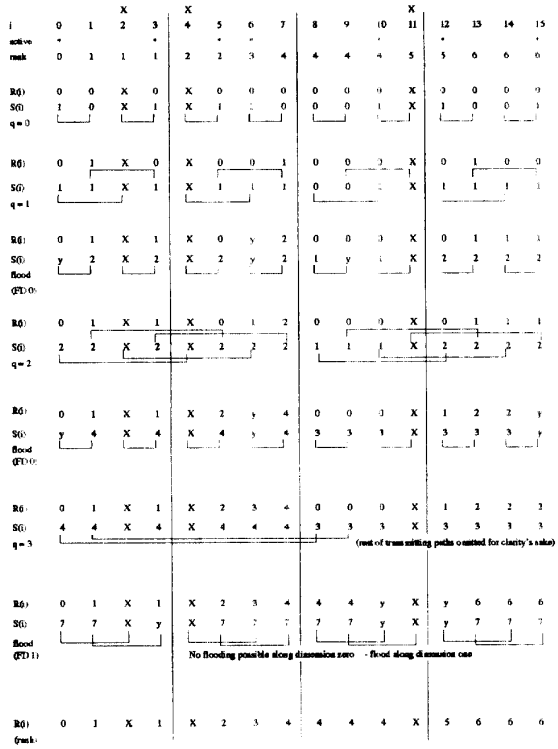

Figure 8: Example of RANK

```
procedure BROADCAST(a, b, var D)
1  D(i) = null, E(i) = null, C'(i) = null
   begin
2     case a, b :
3        0,0:   D(i) = i(st(i) = 1)  /* region-starting index */
4        0,1:   D(i) = R0(i)(st(i) = 1)
5        1,0:   D(i) = i(en(i) = 1)
6        1,1:   D(i) = R1(i)(en(i) = 1 & V(i) ≠null)
7               D(i) = R1(i) − 1(en(i) = 1 & V(i) =null)
8     endcase
9     C(i) = D(i)
10    for q = seq[0] to seq[2n − 2] do
11       if 0-th or odd-numbered step /* fault-free step */
12          C'(i) = C(i)
13          C'(i^(q)) ← C(i)(C(i) ≠ null)
14          D(i) = C'(i)(i_q = 1 − a & D(i) = null)
15          E(i) = C'(i)(i_q = 1 − a & D(i) ≠ null)
16          C(i) = C'(i)(i_q = 1 − a & C(i) = null)
17          C(i) = C'(i)(i_q = a)
18       endif
19       if even-numbered step and greater than 0 /* flood */
20          C(i) ← C(i^(q))(C(i) = null)
21          D(i) ← D(i^(q))(D(i) = null, i_q = 1, q = 0) /* vy */
22          D(i) ← D(i^(q))(D(i) = null, i_q = 0, q = 0) /* yv */
23          D(i) ← E(i^(q))(D(i) = null, i_q = 0, q = 0) /* yv */
24          D(i) ← D(i^(q))(D(i) = null, i_q = 1, q = 1)
25          D(i) ← D(i^(q))(D(i) = null, i_q = 0, q = 1)
26          D(i) ← E(i^(q))(D(i) = null, i_q = 0, q = 1)
27       endif
28    endfor
   end
```

Figure 9: Fault-tolerant BROADCAST algorithm

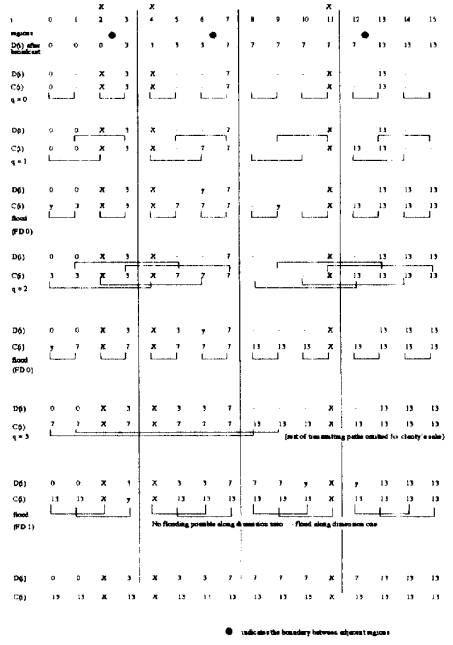

Figure 10: Example of BROADCAST

ter the broadcast operation, $D(i)$ contains the value of the index/rank of the starting/ending PE of the region in which $PE(i)$ lies. We visualize the hypercube as an array of PE's indexed from left to right, starting with index 0 and ending with $N$-1. The $D$ register of a non-source PE changes its value only when it receives data from its left side for the first time. The $C$ register of a non-source PE, on the other hand, changes its value under either of the following two conditions: $(a)$ when it has null data and receives data from its left side, or $(b)$ when it receives any data from its right side.

An example of the BROADCAST algorithm is given in Figure 10. As with the ranking procedure, the array *seq* holds the predetermined sequence of dimensions, for both regular operation and flooding. In the example, we consider a 4-cube with $a = 0, b = 0$. In the context of our fault-free algorithm, the 'starting PE' of a region means the first *fault-free* PE of that region. As usual, flooding is not performed after $q = 0$, but *is* performed after higher values of $q$. In the end, we observe that fault-free PE's have the proper $D$ values.

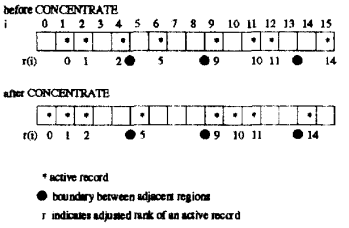


Figure 11: Example of CONCENTRATE for a 4-cube

**procedure** CONCENTRATE(var $\bar{V}, \bar{R}$)
1 $V^{'}(i) = V^{''}(i) = null$
2 $V^{''}(i) = \bar{V}(i)\ (i = \bar{R}(i))$
**begin**
3 **for** $q = seq[0]$ to $seq[2n-2]$ **do**
4 $F(i) = 0, F^{'}(i) = null$
5 **if** even-numbered step /* *regular step* */
6 $F^{'}(i^{(q)}) \leftarrow F(i)$
7 $(W(i^{(r)}), S(i^{(r)})) \leftarrow (\bar{V}(i), \bar{R}(i))\ (F^{'}(i) = null)$
8 $(V^{'}(i^{(q)}), R^{'}(i^{(q)})) \Leftarrow (\bar{V}(i), \bar{R}(i))\ (\bar{V}(i) \neq null\ \&\ \bar{R}(i)_q \neq i_q)$
9 $(W(i^{(q)}), S(i^{(q)})) \Leftarrow (W(i), S(i))\ (W(i) \neq null)$
10 **endif**
11 **if** odd-numbered step /* *flood step* */
12 $(V^{'}(i), R^{'}(i)) \leftarrow (W(i^{(q)}), S(i^{(q)}))\ (F^{'}(i) = null)$
13 **endif**
14 $V^{''}(i) = V^{'}(i), R^{''}(i) = R^{'}(i)\ (i = R^{'}(i))$
15 $\bar{V}(i) = V^{'}(i), \bar{R}(i) = R^{'}(i)\ (i \neq R^{'}(i))$
16 **endfor**
17 $\bar{V}(i) = V^{''}(i), \bar{R}(i) = R^{''}(i)$
**end**

Figure 12: Fault-tolerant CONCENTRATE algorithm

## 5.3 CONCENTRATE

This procedure (see Figure 12) deals with what is known as the *adjusted* rank of a PE, which we define as follows [11]: adjusted rank = *absolute* rank + index of the region-starting/ending PE − *absolute* rank of the region-starting/ending PE. The CONCENTRATE procedure concentrates all active records (records with bit $p$ of the data being equal to 0 (1)) in the front (rear) of their respective regions. The purpose of this procedure should be clear from the example shown in Figure 11. In fact, the adjusted rank of a record is also its final destination. The input to the algorithm consists of the key values of active records, and their adjusted ranks. $\bar{V}(i)$ and $\bar{R}(i)$ represent the key value and the adjusted rank of an active record, respectively. They are the transmitting registers of PE($i$). The "primed" registers are employed as temporary storage in order to avoid conflicts.

**procedure** SET-FLAGS
1 $r^{'}(i) = null,\ st(i) = en(i) = null,\ r(i) = V(i)_p$
**begin**
2 **for** $q = seq[0]$ to $seq[2n-2]$ **do**
3 **if** even-numbered step /* *regular step* */
4 $r^{'}(i^{(q)}) \Leftarrow r(i)$
5 $st(i) = 1\ (r^{'}(i) \neq null\ \&\ r(i) \neq r^{'}(i)\ \&\ i_q = 1\ \&\ st(i) = null)$
6 $st(i) = 0\ (r^{'}(i) \neq null\ \&\ r(i) = r^{'}(i)\ \&\ st(i) = null)$
7 $en(i) = 1\ (r^{'}(i) \neq null\ \&\ r(i) \neq r^{'}(i)\ \&\ i_q = 0\ \&\ en(i) = null)$
8 $en(i) = 0\ (r^{'}(i) \neq null\ \&\ r(i) = r^{'}(i)\ \&\ en(i) = null)$
9 $r(i) = r^{'}(i)\ (r^{'}(i) \neq null\ \&\ i_{q+1} = i_q)$
10 **endif**
11 **if** odd-numbered step /* *flood step* */
12 $S(i^{(q)}) \Leftarrow r^{'}(i)$
13 $st(i) = 1\ (r^{'}(i) = null\ \&\ S(i) \neq r(i)\ \&\ i_q = 1\ \&\ st(i) = null)$
14 $st(i) = 0\ (r^{'}(i) = null\ \&\ S(i) = r(i)\ \&\ st(i) = null)$
15 $en(i) = 1\ (r^{'}(i) = null\ \&\ S(i) \neq r(i)\ \&\ i_q = 0\ \&\ en(i) = null)$
16 $en(i) = 0\ (r^{'}(i) = null\ \&\ S(i) = r(i)\ \&\ en(i) = null)$
17 $r(i) = S(i)\ (r^{'}(i) = null\ \&\ i_{q+1} = i_q)$
18 **endif**
19 **endfor**
**end**

Figure 13: Fault-tolerant SET-FLAGS algorithm

## 5.4 SET-FLAGS

This procedure sets the $st(en)$ flag of a PE if it is the starting(ending) PE of a region. SET-FLAGS simply creates new region boundaries: every region is divided into two sub-regions now, one containing the 0's and the other, 1's. The algorithm is given in Figure 13.

## 5.5 The complete SORT procedure

The complete SORT procedure is given in Figure 14, and is similar to the fault-free SORT procedure in [11]. Each iteration of the complete algorithm (lines 4 to 17, included) accomplishes radix sort along one bit of the key values on the PE's, and takes $O(\log N)$ time. The main sorting algorithm, thus, is of complexity $O(\log M * \log N)$. Finding an FD pair, a one-time procedure, takes $O(\log^2 N)$ time to complete. Therefore, the complete algorithm may be accomplished in $O(\log M * \log N) + O(\log^2 N)$ time.

Line 2 sets the $st$ and the $en$ flags for the initial region which encompasses all the PE's. Lines 4-7 (8-11) compute the adjusted rank for each key value with $V(i)_p = 0$ ($V(i)_p = 1$). After the CONCENTRATE operation, the flag-setting operation is performed for the newly-created regions in line 17.

```
procedure SORT
1  st(i) = en(i) = 0
2  st(0) = en(N − 1) = 1
   begin
3    for p = log M − 1 to 0 do
4      RANK(p,0,R0)
5      BROADCAST(0,0,S) /* region-starting indices*/
6      BROADCAST(0,1,R') /* region-starting ranks*/
7      R0(i) = R0(i) + S(i) − R'(i) (V(i)p = 0) /* adj rank */
8      RANK(p,1,R1)
9      BROADCAST(1,0,E) /* region-ending indices*/
10     BROADCAST(1,1,R') /* region-ending ranks */
11     R1(i) = R1(i) + E(i) − R'(i) (V(i)p = 1) /* adj rank */
12     V1(i) = V0(i) = null
13     Vt(i) = V(i) (V(i)p = t)
14     CONCENTRATE(V0, R0) /* to destination */
15     CONCENTRATE(V1, R1) /* to destination */
16     V(i) = Vt(i) (V(i)p = t)
17     SET-FLAGS /* for starting/ending PE's */
18   endfor
   end
```

Figure 14: The complete SORT procedure

## 6 CONCLUSION

In this paper, we developed a new fault-tolerant sorting algorithm that sorts upto $2^n = N$ keys in a faulty SIMD hypercube. We introduced the concept of *flood* dimensions in order to route data around faulty processors. Our algorithm can be extended to run on MIMD hypercubes. Future research directions include extending the algorithm to sort more than $N$ keys, and reducing the number of flood steps, if not the time complexity, needed to ensure the correct execution of the sorting algorithm.

## References


[1] B. Becker and H. Simon, "How Robust is the n-Cube?," *Information and Computation*, pp. 162–178, 1988.

[2] A. Wang, R. Cypher, and Mayr, "Embedding Complete Binary Trees in Faulty Hypercubes," In *Proc. International Symposium on Parallel and Distributed Processing*, pp. 112–119, December 1991.

[3] M. Y. Chan and S. J. Lee, "Fault-Tolerant Embedding of Complete Binary Trees in Hypercubes," *IEEE Transactions on Parallel and Distributed Systems*, pp. 277–288, March 1993.

[4] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA, 1989.

[5] Y. Chang, "Fault Tolerant Broadcasting Algorithms in SIMD Hypercubes," In *Proc. International Symposium on Parallel and Distributed Processing*, pp. 348–351, Dec. 1993.

[6] A. Mishra, Y. Chang, L. N. Bhuyan, and F. Lombardi, "Fault-Tolerant Sorting in SIMD Hypercubes," Technical Report 95-004, Department of Computer Science, Texas A&M University, January 1995.

[7] B. M. McMillin and L. M. Ni, "Reliable Distributed Sorting Through the Application-Oriented Fault Tolerance Paradigm," *IEEE Transactions on Parallel and Distributed Systems*, pp. 411–420, July 1992.

[8] T. Leighton, B. Maggs, and R. Sitaraman, "On the Fault Tolerance of Some Popular Bounded-Degree Networks," In *33rd IEEE Symp on Foundations of Computer Science*, pp. 542–552, 1992.

[9] P. J. Narayanan, "Processor Autonomy on SIMD Architectures," In *Proc. the ACM International Conference on Supercomputing*, pp. 127–136, July 1993.

[10] G. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A Comparison of Sorting Algorithms for the Connection Machine CM-2," In *Symposium on Parallel Algorithms and Architectures*, pp. 3–16, July 1991.

[11] W.-M. Lin and V. K. P. Kumar, "Efficient Histogramming on Hypercube SIMD Machines," *Computer Vision, Graphics, and Image Processing*, vol. 41, pp. 104–120, 1990.