Fault-Tolerant Sorting in SIMD Hypercubes[1]

A. Mishra, *Y. Chang*, *L. Bhuyan* and *F. Lombardi*
Department of Computer Science
Texas A&M University
College Station, Texas 77843-3112
E-mail: {amishra,ychang,bhuyan,lombardi}@cs.tamu.edu

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. If all the key values to be sorted belong to the range 0 to $M-1$, where M is much smaller than N, sorting can be accomplished efficiently in $O(\log M * \log N)$ time.

Keywords: SIMD, Fault-tolerant, Parallel Sorting, Radix Sort.

Corresponding Author: Yeimkuan Chang, Research Associate.

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

1 Introduction

The hypercube architecture, by virtue of properties like embeddability, structural regularity and fault-tolerance, 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]. Designing parallel algorithms for multicomputers has been a major research issue. However, the development of *fault tolerant* parallel algorithms has received little attention. 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. It is reputed to consume roughly 20 percent of computing resources in large-scale installations [5]. Sorting is the the basis of many scientific and engineering applications such as those in computational geometry and particle-in-a-cell simulations [6], histogramming applications employed in image processing and computer vision [7], and commercial applications like data- or transaction-processing. In fact, sorting has also been proposed as an important benchmark for supercomputers [8].

To the best of our knowledge, there is no 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 [9]. In MIMD hypercubes, a processor can send messages along all dimensions at the same time. In a SIMD hypercube, processors operate in a lock-step fashion, so that at any time, messages are forwarded only along one dimension. In the proposed algorithm, we have focussed on SIMD hypercubes. Since SIMD hypercubes are a restricted version of their MIMD counterparts, our algorithm is also good for the MIMD environment. The proposed algorithm assumes a SIMD model with *activity control* [10]. 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, and is, in the worst case, of $O(\log^2 N)$ complexity. However, if all key values lie within the range 0 to $M-1$, where M is much less than N, the algorithm only takes $O(\log M{*}\log N)$ time.

In our fault-tolerant sorting algorithm, we have employed the most-significant-bit(MSB) radix sort, which has been implemented with high performance for data sets with unknown (non-uniform) distributions [11]. Other parallel sorting methods like bucket sort, though shown to be more efficient for known uniform data distributions [12], have not been shown to be as efficient as radix sort for non-uniform data distributions. In addition, implementations of parallel radix sort have been shown to be stable, deterministic, and much simpler to code [11].

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. As part of the flood operation, after each step of the normal fault-free algorithm, we transmit information from the fault-free nodes to neighbors containing incorrect data. As we will show later in section 4, we need two flood dimensions. However, not every dimension can be a candidate for flood dimension. 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.

The rest of this paper is organized as follows. In the next section, we define the hypercube model, and the notations employed in our algorithm. Then, we provide a sketch of the fault-tolerant algorithm in section 3. In section 4, the concept of flooding and flood dimensions is explained. In section 5, we present the analysis of our fault-tolerant algorithm. Finally, concluding remarks are given.

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 ith 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 simply stops. In addition, it is assumed that a diagnostic process has been successfully executed such that we know the locations of the faulty PE's before running the fault-tolerant algorithm.

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.

We assume that the values to be sorted may be quantized to have certain discrete values, say, 0,1,...,M-1. Each of these values may be specified by its binary representation $i_{m-1}i_{m-2}...i_0$, where $m = \lceil \log M \rceil$. Each value to be sorted is assumed to be stored in register V of each PE of the hypercube.

The following notations have been employed in this paper -

1. Capital letters denote registers in a PE. $C(i)$, for example, refers to the contents of register C of PE(i).
2. i_k represents bit k in the binary representation of integer i.
3. If integer i has the binary representation $i_{n-1}i_{n-2}...i_0$, then $i_{s:t}$ denotes the integer with the binary representation $i_s i_{s-1}...i_t$, where $n-1 \geq s \geq t \geq 0$.
4. f denotes the number of faulty PE's in the hypercube. The maximum number of faulty PE's which the proposed faul-tolerant algorithm tolerates, is $(n-1)$, so $f \leq (n-1)$.
5. PE's are activated by the condition specified in parenthesis after the statement. For example, consider the statement $A(i) = 1$ $(i = odd)$. The activation condition in this case is whether i is odd. If so, the register A of PE(i) is assigned the value 1.
6. A q-block is a set of 2^q PE's adjacent to one another, which have the same bit q in their indices. A q-block may be specified by an $(n-q)$-bit index $i_{n-1}i_{n-2}...i_q$. Two q-blocks with the same index except at bit q are said to be *complementary* q-blocks.

Each PE of the SIMD hypercube is assumed to have an ALU and a constant number of registers. During various data movement operations, three kinds of assignment notations have been used in this paper. They are as follows.

(a) '=' is used for intra-PE assignment: data movement between registers in a PE.

(b) '$\leftarrow$' is employed for inter-PE data assignment. Both the sender and the target registers contain the same data after this operation.

(c) '$\Leftarrow$' denotes inter-PE data movement: the contents of the sending register are set to null after the operation.

It is assumed that each of the above three kinds of assignments takes one unit of time.

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 *rank*ed, 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 complete 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 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. Each of the $RANK$, $BROADCAST$, $CONCENTRATE$ and $SET\text{-}FLAGS$ procedures is explained in detail in section 5.

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. In the worst case, some pseudo-faulty nodes may fail to receive the correct information even after the flood operation. 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 the 4-cube shown in Figure 3. Algorithm $First\text{-}example$ 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 4. Dimension 0 is the flood dimension of the example. The steps constituting the execution of algorithm $First\text{-}example$ are shown in Figure 4. 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 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. However, if we have a *pair* of FDs, whenever we come across a situation like the one outlined in our example above, we can employ the other FD for the flood operation. Below, we show that it is never true that there is an x-y condition along *both* the FDs at the same time. Hence there will always be at least one free FD for the flood operation to take place.

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: We prove the theorem by finding out which of the C_2^n possible pairs of dimensions of the n-cube can *not* be FD-pairs in the worst case. The central observation is the following: two faulty nodes, when placed at opposite ends of the diagonal of a 2-cube, prevent the two dimensions of the 2-cube from being an FD-pair. Let's consider Figure 5.

During the execution of algorithm *First-example*, suppose that node #3 becomes pseudo-faulty. Then, none of the two dimensions, 0 and 1, may be used to transmit data to node #3 because nodes 1 and 2 are faulty themselves. Hence, the two dimensions of the 2-cube may *not* be used as an FD-pair.

It follows that, 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 can not be used as an FD-pair. Since there are $(n-1)$ faults in the n-cube in the worst case, the number of pairs of dimensions that can not be used as FD-pairs, are C_2^{n-1}. Therefore, there are always at least $C_2^n - C_2^{n-1} = \frac{n(n-1)}{2}$ FD-pairs. $\square$

4.1 Finding an FD-pair

One pair of FDs is sufficient for our algorithm to work. In this subsection, we develop an efficient algorithm to find an FD-pair by eliminating all pairs of dimensions that can *not* possibly be FD-pairs. As we have already noted, 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 6. The basic idea is as follows. Assume that f faults have already been detected in the n-cube. Then, we store the addresses of the faulty PE's in the array $FAULT$. FAULT[0], FAULT[1],..., FAULT[$f-1$], therefore, 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. If the distance is one, it means that the two faulty PE's are neighbors along a dimension. Dimension1 is the variable representing that dimension, and we discard it from further consideration because it can not be a flood dimension. Similarly, if the distance is two, we discard the two dimensions of the 2-cube containing the two faulty PE's. The procedure *discard* simply maintains an array of all dimensions and deletes the rejected dimensions from the list. Lines 3 and 5 take $O(n)$ time because node addresses are of n bits. This algorithm, therefore, takes $O(n^3)$ time when executed sequentially. However, if it is performed in parallel on n nodes, the procedure takes $O(n^2)$ time to execute.

4.2 Determining the sequence of dimensions for flooding

In the worst case, a flood operation is necessary after each step of the fault-free sorting algorithm, to ensure that all pseudo-faulty PE's receive the correct information. We perform the flood operation after each step of the fault-free algorithm, except after the first step. The reason is as follows. The first step of the fault-free algorithm transmits data along dimension 0. If a faulty node fails to transmit along dimension 0, it will make the neighbor along that dimension pseudo-faulty. Logically, we should flood along FD 1 to rectify this situation. However, we don't find it necesary because anyway the fault-free step *does* transmit along dimension 1 in its very next step. 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). At the onset of every flood operation, therefore, a decision has to be made regarding which dimension to flood, depending on whether or not an x-y condition exists along dimension 0. This decision has to be made on-line, causing overhead. Instead, we could *predetermine* the dimension sequence for flooding to take place.

During 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, and equal either 0 or 1. Our objective now is to determine this dimension sequence. The procedure Dimension-seq performs this. 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 basic idea is as follows. Line 5 covers the 0-th dimension neighbors of f faulty PE's. For each such PE, we check whether any neighboring PE along any other dimension k is faulty. If it is so, after the regular operation along dimension k, we flood along dimension 1. Line 7 performs this.

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 8. 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 8. 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. We discuss them in order.

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. Whether a record is active or not is specified by two parameters, p and t. 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 9. The procedure *Find-FD-pair* in the RANK algorithm finds an FD pair, and the two dimensions belonging to the FD pair are now renumbered as dimensions 0 and 1. As noted in the previous section, we already have the predetermined dimension sequence in the register *seq*.

Register A contains information regarding whether a given PE is active or not (i.e., whether it has an active record 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. It might seem as if the data in register A is redundant, because we already have information on the active PE's in S. A is not redundant, however, because S gets updated after each step. Register R contains the rank of each PE. Lines 1-3 are the initialization steps. Line 13 specifies the flood step. Of course, flooding follows each step of the fault-free algorithm, except after the first step, when data transmission takes place along dimension 0. Flooding requires the adjustment of the rank of the pseudo-faulty PE, depending on which PE(s) is(are) active. This is done in lines 14-17.

An example of the RANK algorithm is given in Figure 10. 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 pth 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. This is done in parallel across all regions. Parameters a and b specify which characteristic of the region starting/ending PE must be broadcast to all PE's of 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 and en are two flags which indicate whether a PE is the starting or the ending PE of a region, respectively. $st(i) = 1$ 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 11. Registers C and D denote the transmitting and the result registers, respectively. After 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. Depending on the values of parameters a and b, we initialize the register $D(i)$. For example, if $a = 0, b = 0$, then the starting PE's of different regions will have their D values initialized to their respective indices, and the rest of the PE's will have their D values set to null.

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 basic principles of the fault-free part of the procedure are given as follows:

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.

In effect, the D register of a non-source PE will change its value from null to a non-null one *only when it receives a value from a PE in the same region to its left for the first time.* As it is a non-source PE, there is at least one more PE in the same region to its left which is a neighbor along some dimension, and can transmit a non-null value.

We have initialized the $C'(i)$ register to null. In the fault-free step, whenever $C'(i^{(q)})$ register fails to receive a value from $C(i)$, it remains at null, i.e., PE(i) is faulty, and PE($i^{(q)}$) is now pseudo-faulty, as indicated by its value of null. The value in $D(i)$ is not updated after it is changed for the first time from null to a non-null value. Subsequently, whenever an already-updated $D(i)$ receives a value, it is put into register $E(i)$, where it is used later for flooding purposes. During any step in the fault-free part, either $C(i)$ or $D(i)$ may receive null from $C'(i)$, indicating that they are pseudo-faulty.

In the flood operation, we flood along one of the FD's in order to rectify the incorrect data present (C and D registers) in the pseudo-faulty nodes. Similar to the RANK algorithm, BROADCAST, too, has a predetermined sequence of dimensions for carrying out both fault-free and flood steps. The main difference in this procedure, however, lies in the fact that unless PE(i) and PE($i^{(q)}$) reside in the same region, the D values in these two PE's should be different; thus, we may not simply equate them. Instead, we equate $D(i)$ to $E(i^{(q)})$. In the case of $C(i)$, however, we may simply equate it to $C(i^{(q)})$ independent of whether the two are in the same region. A formal proof of the BROADCAST procedure follows.

Claim. The BROADCAST procedure accomplishes the desired broadcast operation.
Proof. We deal with only the case for $a = 0$ and $b = 0$. The proof for other combinations of a and b values is similar. The proof of the correctness of the fault-free part of the procedure is given in [7]. We provide the additional proof that the flood operation transmits data correctly to the pseudo-faulty nodes. We consider the C and D registers separately.

Register C - We perform the flood operation after each step of the fault-free algorithm, except after the first step. The reason has been explained earlier.

After the flood-free step has been performed for $q = 0$ and $q = 1$, during any subsequent step, register C has the same value for all PE's in any 2-cube. During any flood step, we always flood along FD 0 or 1, along which the neighbor always lies in the same 2-cube; therefore, we may simply equate $C(i)$ to $C'(i^{(q)})$ when PE(i) is pseudo-faulty.

Register D - $D(i)$, too, receives the correct data from $D'(i^{(q)})$ ($q = 0$ or 1) if PE(i) is pseudo-faulty. However, in this case, consideration must be given to the fact that there is a possibility that PE(i) and PE($i^{(q)}$) may not lie in the same region. Let us consider the four possible scenarios for flooding the D register. It may be noted that 'v' denotes a PE having a non-null value in its D register, 'y' denotes a pseudo-faulty PE, and 'd', a *don't care* PE, that is, a PE whose D register value is irrelevant to the discussion. Cases (i) and

(ii) cover situations where flooding takes place along FD 0 (so the pseudo-faulty PE and the flooding PE are adjacent), and cases (iii) and (iv) represent cases where we use FD 1 for flooding: the pseudo-faulty PE and the flooding PE are neighbors along dimension 1.

(i) v y **case.**
This denotes the case where PE(k) (having $D(k) = v$) and PE($k+1$) (having $D(k+1) =$ *null*) are neighbors along FD *0*, and flooding must be performed along the same dimension. This is illustrated in Figure 12. Suppose that PE(k) and PE($k+1$) belong to different regions. Then, PE($k+1$) is the starting PE of a region, and it must have been initialized earlier, to $D(k+1) = k+1$. After that, the D value should never change again; so, $D(k+1) =$ *null* is an impossibility. We arrive at a contradiction. Hence, PE(k) and PE($k+1$) belong to the same region, and we simply equate the values in their D registers.

(ii) y v **case.**
In this case, PE(k) and PE($k+1$) are neighbors along FD 0, along which flooding must be performed. In case the two PE's belong to the same region, we simply equate $D(k)$ to $D(k+1)$. If, however, the two PE's are *not* in the same region, the situation becomes more complicated. This is illustrated in Figure 13. Obviously, we may not simply equate the values in their D registers. The solution is as follows.

PE($k+1$) is the first PE of a region. Let us consider the faulty node, denoted by PE(c), which transmitted along a dimension g during the execution of the fault-free algorithm, and caused PE(k) to become pseudo-faulty. As we have already noted, a non-source PE gets its D register updated only by another PE to its left *lying in the same region*. Therefore, PE(c) and PE(k) must lie in the same region. During transmission along the dimension g, when PE(c) caused PE(k) to become pseudo-faulty, PE($c+1$) must have transmitted to PE($k+1$). But since PE($k+1$) is the first PE of a region, the value of its D register does not get updated, and the incoming value is stored in its E register, instead. So, now it is possible to simply equate $D(k)$ to $E(k+1)$.

(iii) v d y **case.**
This is the case when flooding is performed along FD 1 to rectify the value of the D register of PE($k+2$). We can show that, in this case PE(k) and PE($k+2$) always lie in the same region, as follows. Let us assume that PE($k+2$) is pseudo-faulty due to transmission from a faulty node PE(c), $c < k+2$, along dimension g. Discounting any trivial cases, g must be greater than one, so PE(c) must lie to PE(k)'s left. But, as we discussed in the preceding paragraph, PE(c) and PE(k) lie in the same region, and therefore it follows that PE(k) and PE($k+2$) also lie in the same region. Subsequently, we equate the values in the D

registers of these two PE's.

(iv) $y\ d\ v$ **case.**
In this case, PE$(k+1)$, PE(k)'s neighbor along dimension 0, is faulty, so FD 0 may not be used for flooding. If PE(k) and PE$(k+2)$ lie in the same region, then we equate the D register values. Suppose they do not lie in the same region. Let us assume that the faulty node PE(c) caused PE(k) to become pseudo-faulty. Similar to the discussion for case (ii), we may establish that PE(c) and PE(k) lie in the same region. Along the dimension g, PE$(c+2)$ must have transmitted to PE$(k+2)$. But since PE$(k+2)$ is the first PE of a region, the value of its D register does not get updated, and the incoming value is stored in its E register. As in case (ii), we can equate $D(k)$ to $E(k+2)$. $\square$

An example of the BROADCAST algorithm is given in Figure 16. 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 after higher values of q. In the end, we observe that fault-free PE's have the proper D values.

5.3 CONCENTRATE

This procedure deals with what is known as the *adjusted* rank of a PE, which we define as follows [7]: 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. In fact, the adjusted rank of a record is also its final destination. The purpose of this procedure should be clear from the example shown in Figure 17. This algorithm routes data to its correct 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. In case of a fault-free hypercube, line 8 performs the data movement for routing, line 14 saves the data if it has reached its destination, and line 15 puts the data back to the transmitting registers in case the data has *not* reached its destination (we call such records *non-final* records).

In this procedure, flooding executes as follows. The registers F and F' are employed to detect the pseudo-faulty condition of a PE. If $F^{'}(i)$ does not receive any data from the F register of its neighbor, $F^{'}(i)$ remains at null. This implies that PE($i^{(q)}$) is faulty, and therefore PE(i) will become pseudo-faulty after the execution of line 8. Prior to this, we transfer the useful information in PE(i) to the W and S registers of its neighbor along the FD. r denotes the immediately next value of q, which is an FD. From the next step onwards, the W and S values are transmitted to their neighbors successively. These values are employed to flood pseudo-faulty PE's.

An example of the fault-tolerant CONCENTRATE procedure is provided in Figure 18. We have not indicated the V registers containing the key values, because they are manipulated in exactly the same fashion as the R registers. Non-final records are kept in the transmitting register, $\bar{R}$. If data reaches its final destination, it is stored in the "double-primed" registers. This is indicated by encircled values. Pseudo-faulty values, denoted by y, are rectified by flooding along the FD. We observe that all records reach their destinations correctly.

Claim. Procedure CONCENTRATE accomplishes the desired concentrate operation.
Proof. We know that during the execution of the fault-free algorithm, no *collision* occurs[7], where a collision implies a situation in which two non-final records appear on the same PE at the end of an iteration. Here, we provide the additional proof that flooding enables pseudo-faulty PE's to receive correct data. This scenario is illustrated in Figure 20. During the execution of the procedure, PE(k) observes a faulty node, PE(j), ahead along dimension q. Hence, it transmits data to its neighbor along the flood dimension, denoted by PE($k^{(FD)}$). This can not be a faulty node because it lies along the flood dimension. In the subsequent step, data moves along dimension q. Let us suppose that PE($k^{(FD)}$) transmits its data to PE($j+1$). Now, PE($j+1$) is not faulty, or else dimension 0 would not be an FD. Neither is this node pseudo-faulty, because it did not receive any data from a faulty node. So PE($j+1$) is a fault-free node. During the subsequent step along dimension $(q+1)$, PE(i) becomes pseudo-faulty. However, similar to the preceding argument, it may be shown that PE($i+1$) is fault-free. Hence, the correct data may now be flooded from PE($i+1$) to PE(i). In this fashion, all pseudo-faulty PE's will finally receive the correct data.

5.4 SET-FLAGS

This procedure sets the $st(en)$ flag of a PE if it is the starting(ending) PE of a region. Fig. 21 illustrates this algorithm. Just before the SET-FLAGS procedure is called, the CONCENTRATE algorithm packs all of the 1's, or the 0's, into the front 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 22. Let each PE(i) with $V(i)_p = 0$ send a signal, $r(i)$, to its neighbor. If the right neighbor PE(k) with $V(k)_p = 1$ receives the signal, then it will set its st flag. The setting of the en flag is similar; if the left neighbor PE(k) with $V(k)_p = 1$ receives the signal, it will set its en flag.

At each step, we let the data in register r of PE(i) be transmitted to the r' register of its neighbor. Next, for any PE(i), we check whether the contents of $r(i)$ and $r'(i)$ are equal. If they are different, either the st, or the en flag of PE(i) is set. During execution along dimension q, the condition $(i_q = i_{q+1})$ checks whether the data contained in the right-most PE of the left $(q-1)$-block, or the data contained in the left-most PE of the right $(q-1)$-block, must be used in the next step for setting the flags. The register S contains data needed for flooding.

Claim. The SET-FLAGS procedure is correct.
Proof. The proof that flooding succeeds in restoring correct data in the pseudo-faulty nodes is similar to the proof of the correctness of the fault-tolerant CONCENTRATE procedure given before.

5.5 The complete SORT procedure

The complete SORT procedure is given in Figure 23, and is similar to the fault-free SORT procedure provided in [7]. Each of the four fundamental operations employed in the SORT algorithm, namely, RANK, BROADCAST, CONCENTRATE and SET-FLAGS, is of $O(\log N)$ running time. 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. Therefore, sorting is performed in $O(\log M * \log N)$ 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.

The proposed algorithm can be extended to sort more than N keys. For example, suppose we wish to sort $2N$ keys on an n-cube. We store two key values on each node. One extra dimension spanning the two keys in each node may be introduced to form an $(n+1)$-cube. It is not difficult to verify that the proposed sorting algorithm can be applied directly. We skip the details because of space limitations.

6 CONCLUSION

In this paper, we developed a new fault-tolerant sorting algorithm based on the notion of flood steps for SIMD hypercube machines. Computational overheads (finding out an FD pair and determining the dimension sequence, both of which are one-time operations) are of the same complexity as the actual sorting algorithm in the worst case. Sorting can be performed more efficiently if all key values lie within a range of 0 to $M-1$. The time-complexity of this algorithm when $M = N$, $O(n^2)$, remains the same as that of previously published fault-free algorithms [7, 6]. Our algorithm can be extended to run on MIMD hypercubes. The proposed algorithm can also be extended to sort more than N keys.

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

[5] M. H. Nodine and J. S. Vitter, "Large-Scale Sorting in Parallel Memories," In *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 29–39, July 1991.

[6] L. V. Kale and S. Krishnan, "A Comparison Based Parallel Sorting Algorithm," In *Proc. International Conference on Parallel Processing*, pp. III–196–200, 1993.

[7] 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.

[8] K. Thearling and S. Smith, "An Improved Supercomputer Sorting Benchmark," In *Supercomputing '92*, pp. 14–19, November 1992.

[9] 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.

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

[11] 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.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.

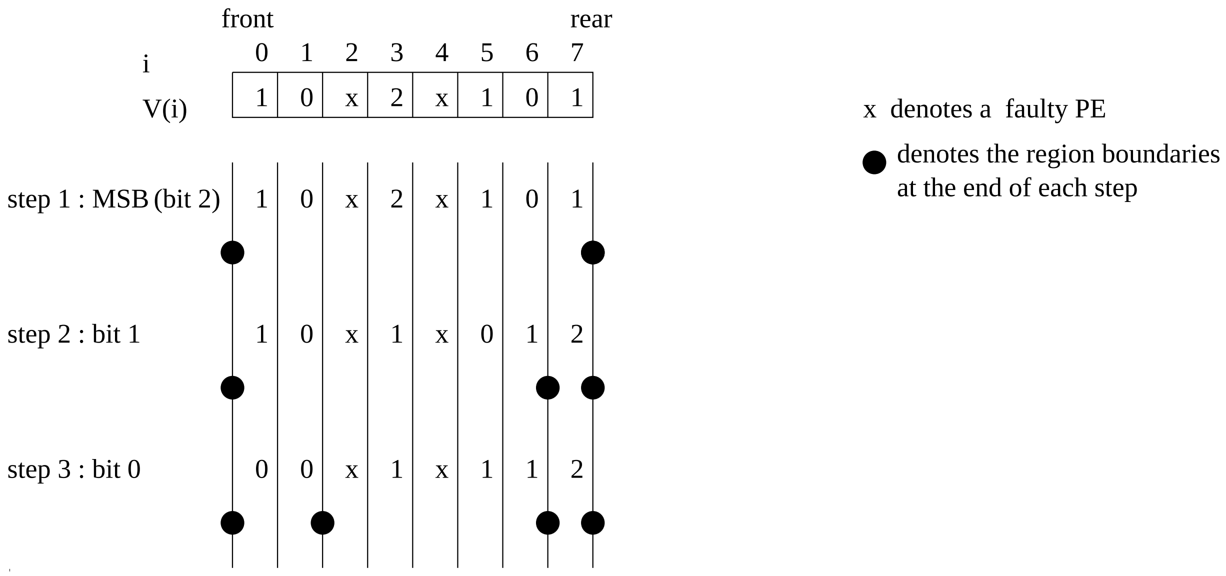

Figure 1: An example of the fault-tolerant algorithm for a 3-cube

Algorithm First-example
/* n is the dimension of the hypercube */
$T(i) = 1$
begin
  for $q = 0$ to $n - 1$ **do**
    $T(i^{(q)}) \leftarrow S(i)$
    $S(i) = S(i) + T(i)$
  endfor
end

Figure 2: Example to illustrate an x-y condition

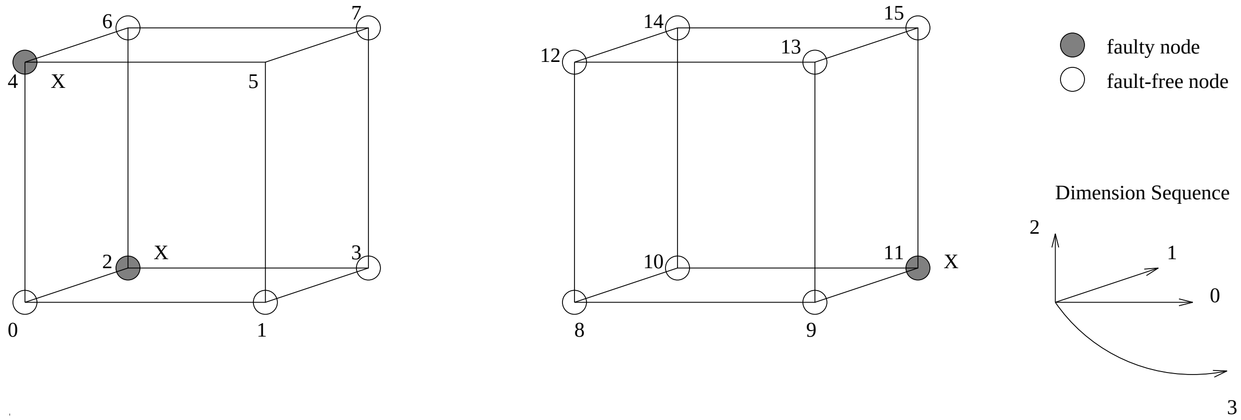

Figure 3: 4-cube for algorithm *First-example*

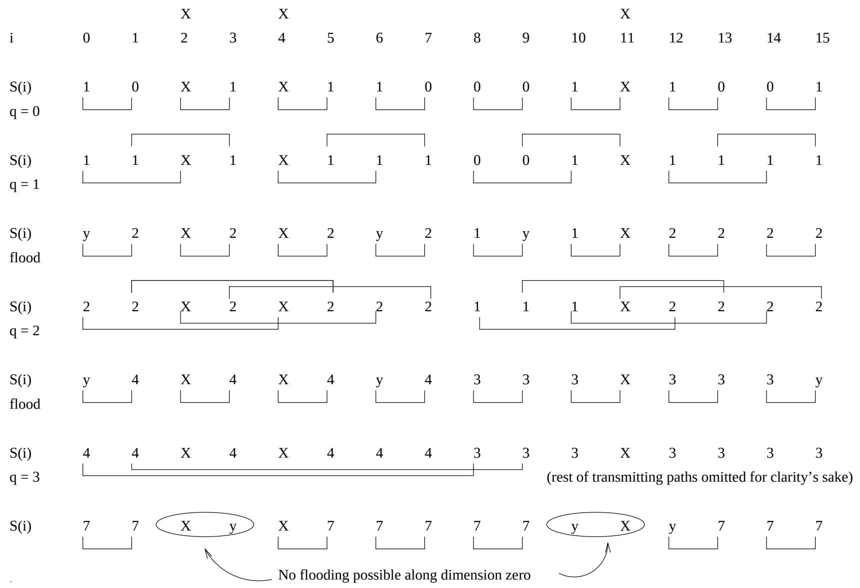

Figure 4: Flood Operation

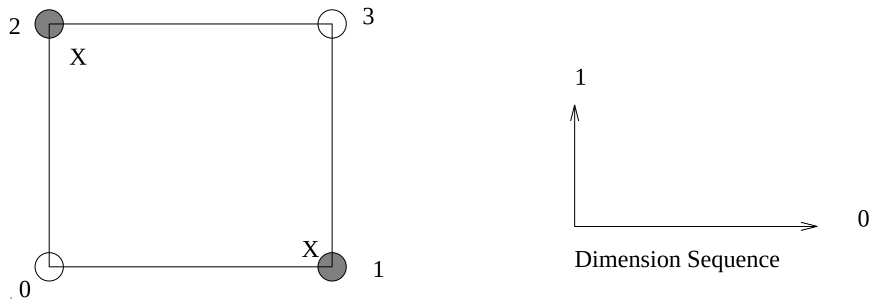

Figure 5: Worst-case scenario during the calculation of the FD-pair

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

Figure 6: Algorithm for finding an FD-pair

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

Figure 7: Algorithm for finding the dimension sequence

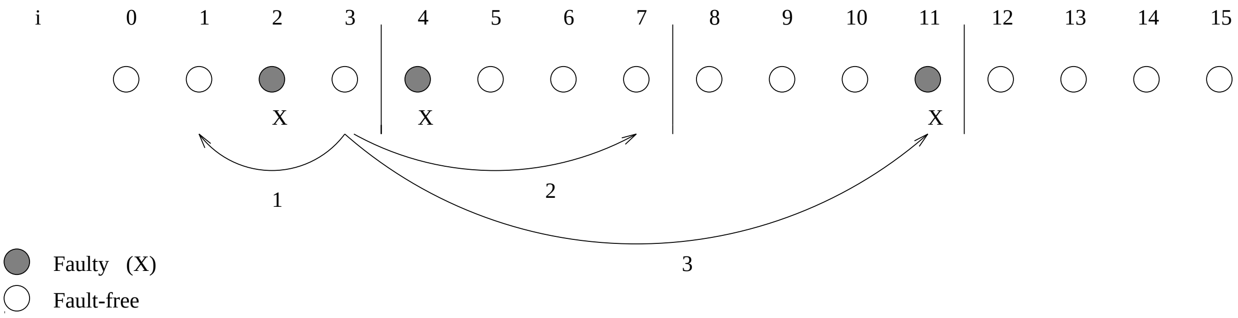

Figure 8: Example for determining a dimension sequence

procedure RANK$(p, t, varR)$

$R(i) = 0, F(i) = 0, T(i) = null$

$S(i) = 1(V(i)_p = t)$

$S(i) = 0(V(i)_p \neq t)$

begin

for $q = seq[0]$ to $seq[2n-2]$ **do**

$T(i^{(q)}) \leftarrow S(i)$

$F(i) = T(i)(j = 0)$

if 0-th or odd-numbered step/* *fault-free step* */

$R(i) = R(i) + T(i)(i_q = 1 \ \& \ T(i) \neq null)$

$S(i) = S(i) + T(i)(T(i) \neq null)$

$S(i) = null, R(i) = null(T(i) = null)$

endif

if even-numbered step & greater than 0 /* *flood step* */

$S(i) \leftarrow S(i^{(q)})(S(i) = null)$

$R(i) \leftarrow R(i^{(q)})$, $R(i) = R(i) - A(i)$ $(i_q = 0 \ \& \ q = 0 \ \& \ S(i) = null)$

$R(i) \leftarrow (R(i^{(q)}) + A(i^{(q)}))$ $(i_q = 1 \ \& \ q = 0 \ \& \ S(i) = null)$

$R(i) \leftarrow R(i^{(q)})$, $R(i) = R(i) - A(i) - A(i+1)$ $(i_q = 0 \ \& \ q = 1 \ \& \ S(i) = null)$

$R(i) \leftarrow R(i^{(q)})$, $R(i) = R(i) + A(i^{(q)}) + A(i^{(q)} + 1)(i_q = 1 \ \& \ q = 1 \ \& \ S(i) = null)$

endif

endfor

end

Figure 9: Fault-tolerant RANK algorithm

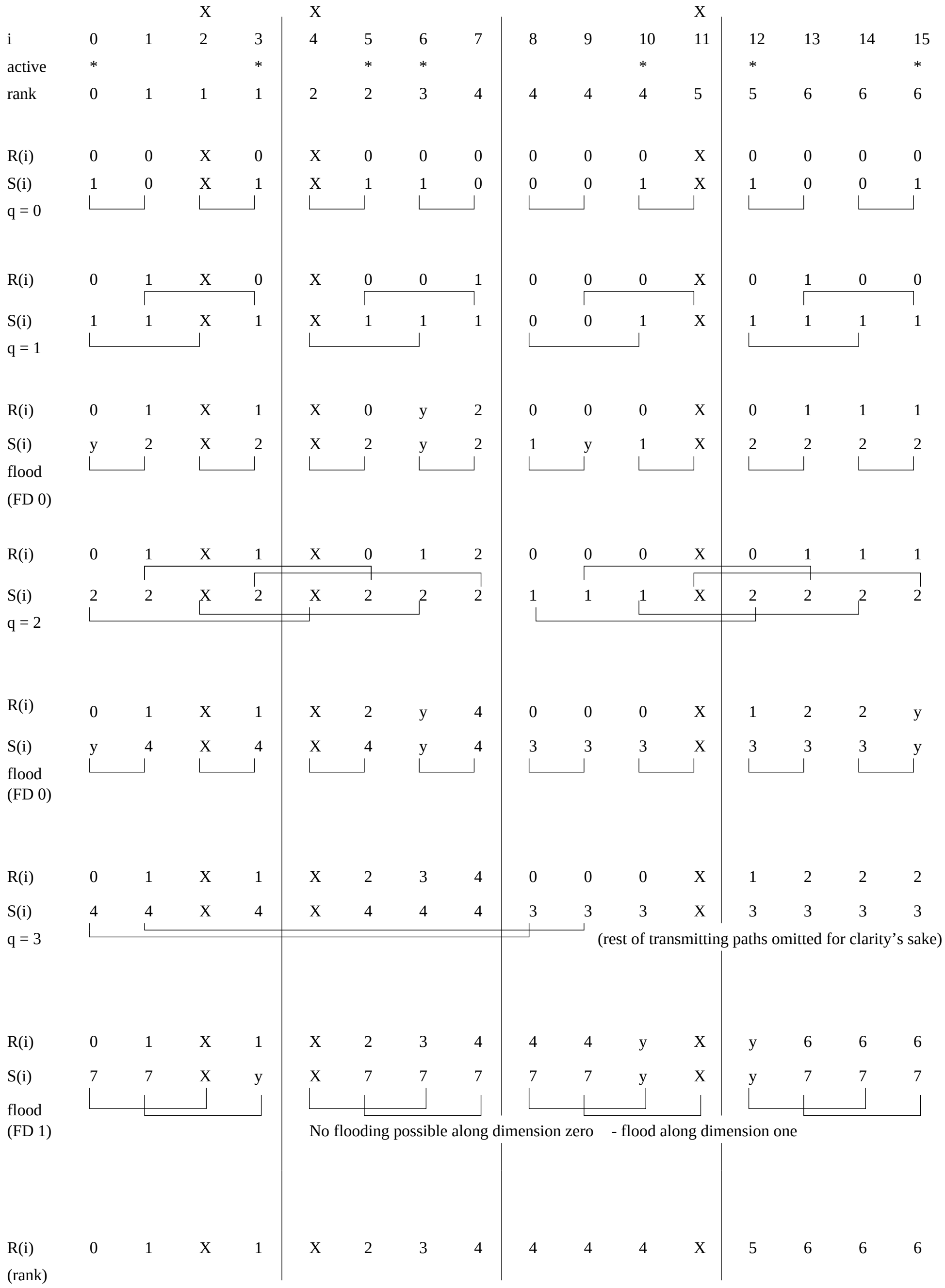

Figure 10: Example of RANK

```
procedure BROADCAST(a, b, var D)
D(i) = null, E(i) = null, C'(i) = null
begin
    case a, b :
        0,0:  D(i) = i(st(i) = 1)                          /* region-starting index */
        0,1:  D(i) = R_0(i)(st(i) = 1)                     /* region-starting rank */
        1,0:  D(i) = i(en(i) = 1)                          /* region-ending index */
        1,1:  D(i) = R_1(i)(en(i) = 1 & V(i) ≠null)        /* region-ending rank */
              D(i) = R_1(i)null(en(i) = 1 & V(i) =null)
    endcase
    C(i) = D(i)
    for q = seq[0] to seq[2n − 2] do
        if 0-th or odd-numbered step /* fault-free step */
            C'(i) = C(i)
            C'(i^(q)) ← C(i)(C(i) ≠ null)
            D(i) = C'(i)(i_q = 1 − a & D(i) = null )
            E(i) = C'(i)(i_q = 1 − a & D(i) ≠ null)
            C(i) = C'(i)(i_q = 1 − a & C(i) = null)
            C(i) = C'(i)(i_q = a)
        endif
        if even-numbered step and greater than 0 /* flood step */
            C(i) ← C(i^(q))(C(i) = null)
            D(i) ← D(i^(q))(D(i) = null, i_q = 1, q = 0) /* vy */
            D(i) ← D(i^(q))(D(i) = null, i_q = 0, q = 0) /* yv */
            D(i) ← E(i^(q))(D(i) = null, i_q = 0, q = 0) /* yv */
            D(i) ← D(i^(q))(D(i) = null, i_q = 1, q = 1) /* v d y */
            D(i) ← D(i^(q))(D(i) = null, i_q = 0, q = 1) /* y d v */
            D(i) ← E(i^(q))(D(i) = null, i_q = 0, q = 1) /* y d v */
        endif
    endfor
end
```

Figure 11: Fault-tolerant BROADCAST algorithm

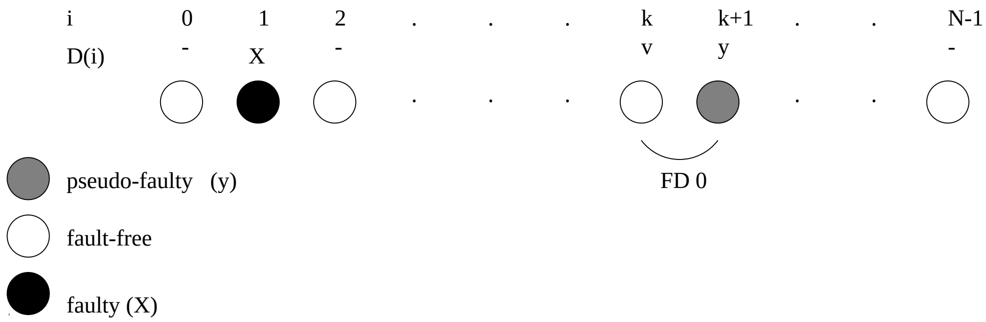

Figure 12: Example of the $v\ y$ case

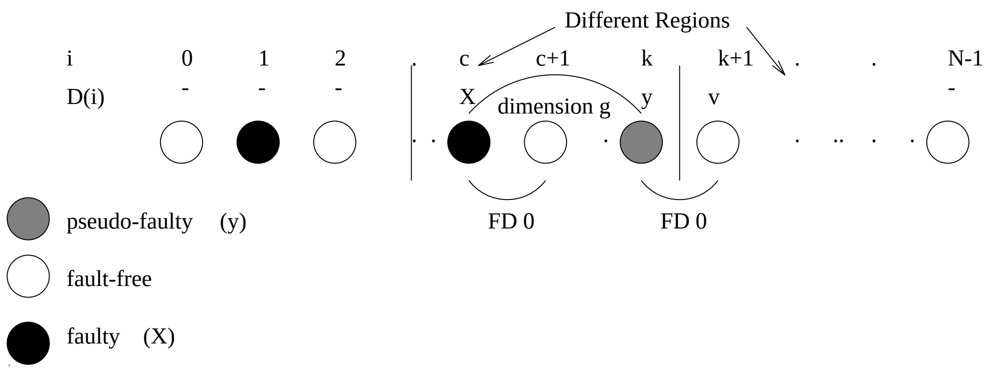

Figure 13: Example of the $y\ v$ case

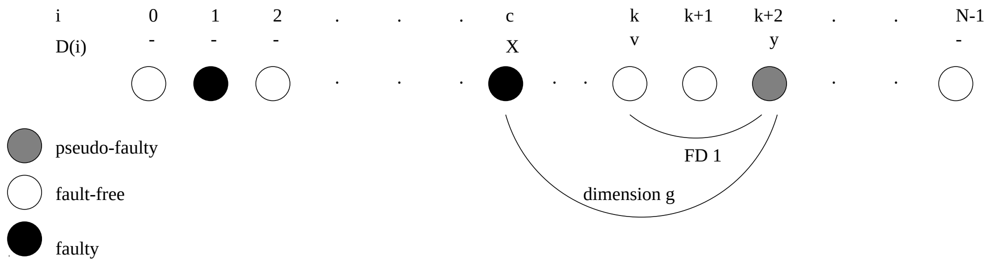

Figure 14: Example of the $v\ d\ y$ case

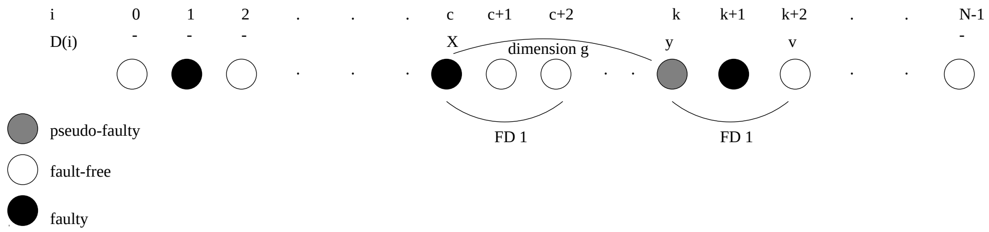

Figure 15: Example of the $y\ d\ v$ case

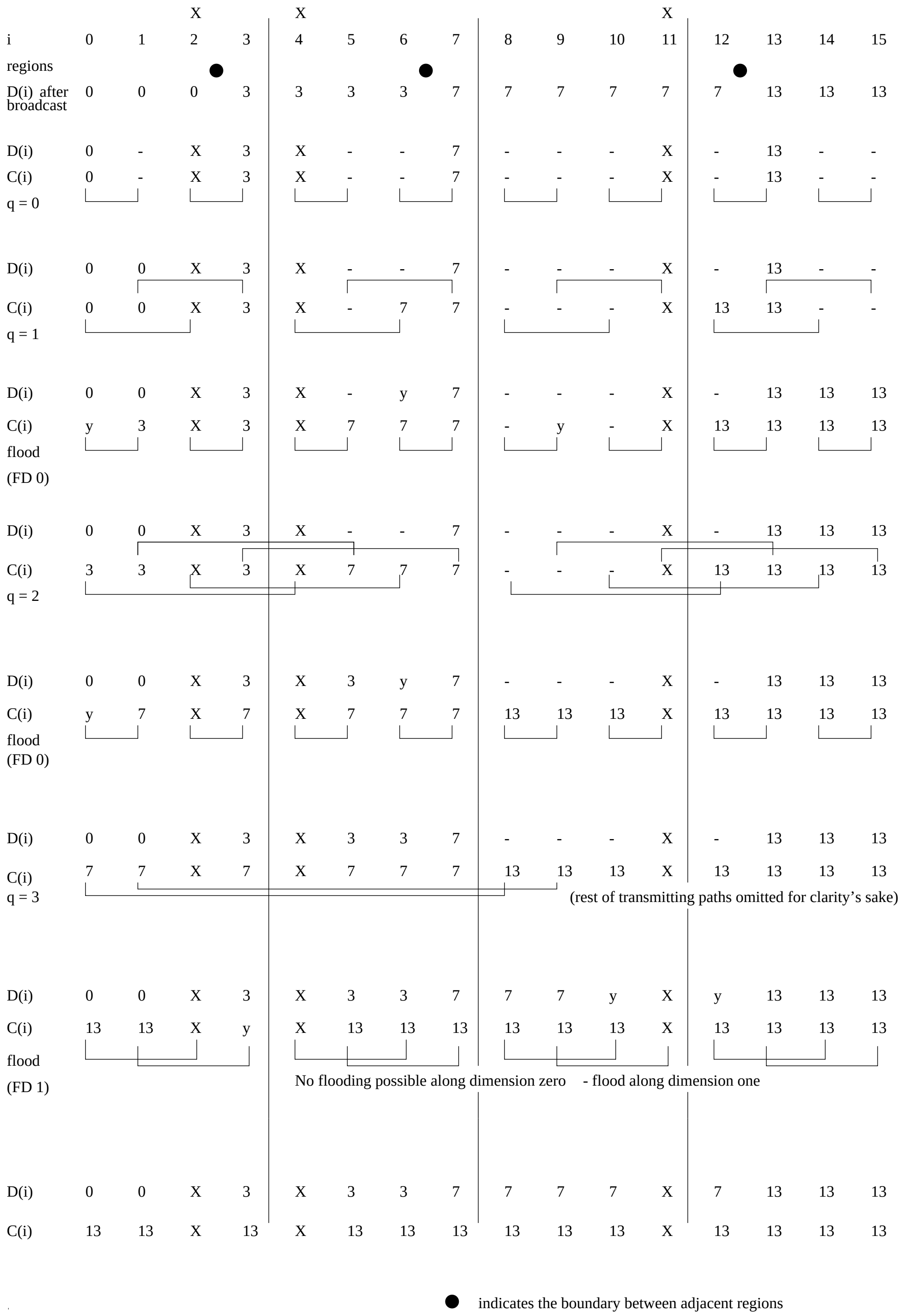

Figure 16: Example of BROADCAST

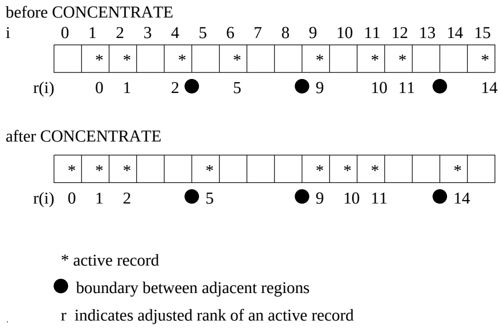

Figure 17: Example of CONCENTRATE for a 4-cube

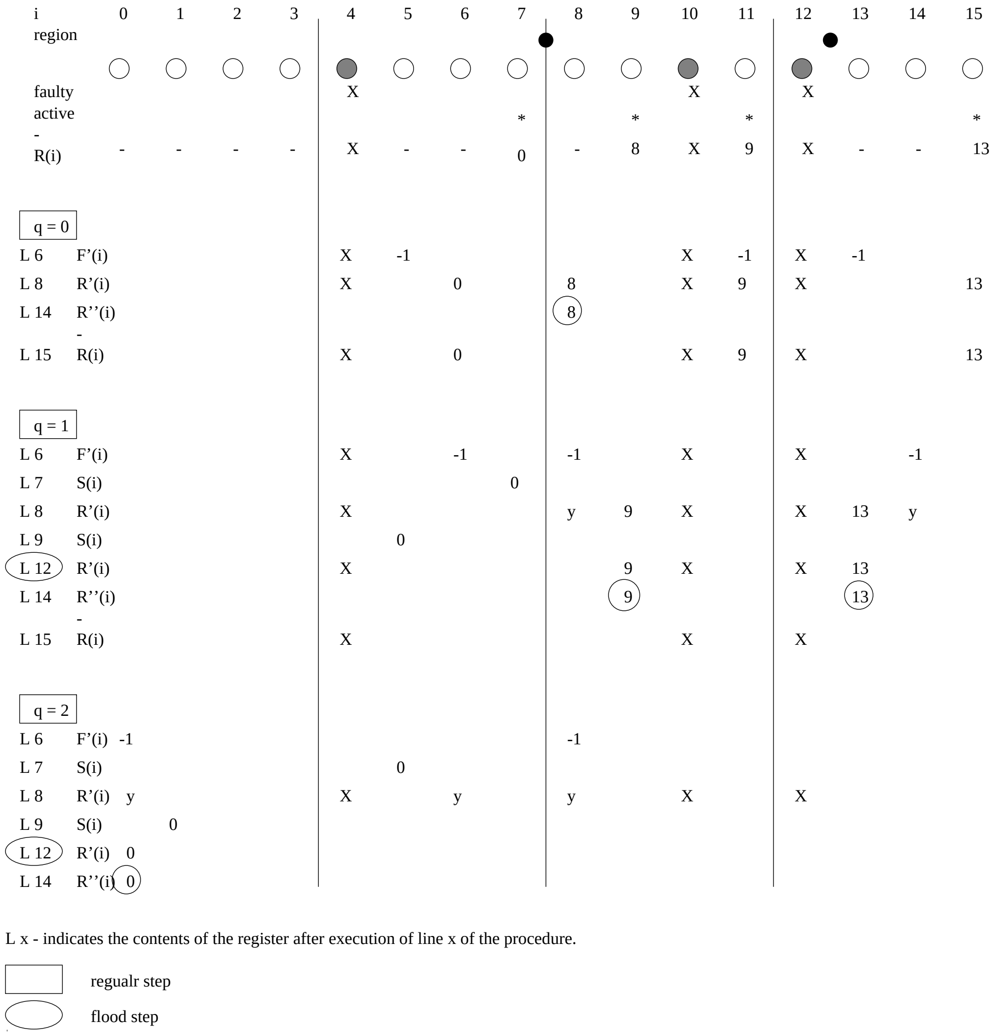

Figure 18: Example of the fault-tolerant CONCENTRATE algorithm

procedure CONCENTRATE(var $\bar{V}, \bar{R}$)

$V'(i) = V''(i) = null$

$V''(i) = \bar{V}(i)$ $(i = \bar{R}(i))$

begin

  for $q = seq[0]$ to $seq[2n-2]$ **do**

    $F(i) = 0$, $F'(i) = null$

    if even-numbered step /* *regular step* */

      $F'(i^{(q)}) \leftarrow F(i)$

      $(W(i^{(r)}), S(i^{(r)})) \leftarrow (\bar{V}(i), \bar{R}(i))$ $(F'(i) = null)$

      $(V'(i^{(q)}), R'(i^{(q)})) \Leftarrow (\bar{V}(i), \bar{R}(i))$ $(\bar{V}(i) \neq null$ & $\bar{R}(i)_q \neq i_q)$

      $(W(i^{(q)}), S(i^{(q)})) \Leftarrow (W(i), S(i))$ $(W(i) \neq null)$

    endif

    if odd-numbered step /* *flood step* */

      $(V'(i), R'(i)) \leftarrow (W(i^{(q)}), S(i^{(q)}))$ $(F'(i) = null)$ /* PE(i) is p.f. */

    endif

    $V''(i) = V'(i), R''(i) = R'(i)$ $(i = R'(i))$

    $\bar{V}(i) = V'(i), \bar{R}(i) = R'(i)$ $(i \neq R'(i))$

  endfor

  $\bar{V}(i) = V''(i), \bar{R}(i) = R''(i)$

end

Figure 19: Fault-tolerant CONCENTRATE algorithm

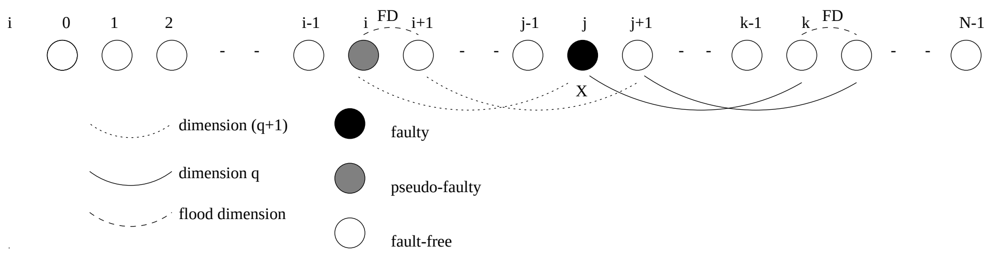

Figure 20: Proof of correctness of CONCENTRATE

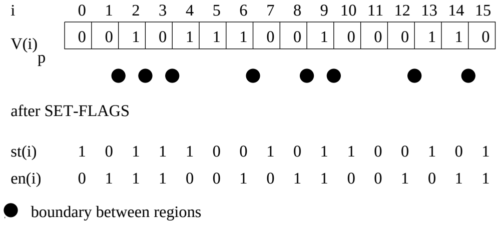

Figure 21: Example of SET-FLAGS for a 4-cube

procedure SET-FLAGS
$r'(i) = null$, $st(i) = en(i) = null$, $r(i) = V(i)_p$
begin
    for $q = seq[0]$ to $seq[2n-2]$ **do**
        if even-numbered step /* *regular step* */
            $r'(i^{(q)}) \Leftarrow r(i)$
            $st(i) = 1 \ (r'(i) \neq null \ \& \ r(i) \neq r'(i) \ \& \ i_q = 1 \ \& \ st(i) = null)$
            $st(i) = 0 \ (r'(i) \neq null \ \& \ r(i) = r'(i) \ \& \ st(i) = null)$
            $en(i) = 1 \ (r'(i) \neq null \ \& \ r(i) \neq r'(i) \ \& \ i_q = 0 \ \& \ en(i) = null)$
            $en(i) = 0 \ (r'(i) \neq null \ \& \ r(i) = r'(i) \ \& \ en(i) = null)$
            $r(i) = r'(i) \ (r'(i) \neq null \ \& \ i_{q+1} = i_q)$
        endif
        if odd-numbered step /* *flood step* */
            $S(i^{(q)}) \Leftarrow r'(i)$
            $st(i) = 1 \ (r'(i) = null \ \& \ S(i) \neq r(i) \ \& \ i_q = 1 \ \& \ st(i) = null)$
            $st(i) = 0 \ (r'(i) = null \ \& \ S(i) = r(i) \ \& \ st(i) = null)$
            $en(i) = 1 \ (r'(i) = null \ \& \ S(i) \neq r(i) \ \& \ i_q = 0 \ \& \ en(i) = null)$
            $en(i) = 0 \ (r'(i) = null \ \& \ S(i) = r(i) \ \& \ en(i) = null)$
            $r(i) = S(i) \ (r'(i) = null \ \& \ i_{q+1} = i_q)$
        endif
    endfor
end

Figure 22: Fault-tolerant SET-FLAGS algorithm

procedure SORT
$st(i) = en(i) = 0$
$st(0) = en(N-1) = 1$
begin
    for $p = \log M - 1$ to 0 **do**
        RANK(p,0,R_0)
        BROADCAST(0,0,S) /* broadcast region-starting indices*/
        BROADCAST(0,1,R') /* broadcast region-starting ranks*/
        $R_0(i) = R_0(i) + S(i) - R'(i)$ $(V(i)_p = 0)$ /* calculate adjusted rank */
        RANK(p,1,R_1)
        BROADCAST(1,0,E) /* broadcast region-ending indices*/
        BROADCAST(1,1,R') /* broadcast region-ending ranks */
        $R_1(i) = R_1(i) + E(i) - R'(i)$ $(V(i)_p = 1)$ /* calculate adjusted rank */
        $V_1(i) = V_0(i) = null$
        $V_t(i) = V(i)$ $(V(i)_p = t)$
        CONCENTRATE(V_0, R_0) /* routing data to destination */
        CONCENTRATE(V_1, R_1) /* routing data to destination */
        $V(i) = V_t(i)$ $(V(i)_p = t)$
        SET-FLAGS /* set flags for starting/ending PE's */
    endfor
end

Figure 23: The complete SORT procedure