Optimal Sorting Algorithms on Incomplete Meshes with Arbitrary Fault Patterns

Chi-Hsiang Yeh and Behrooz Parhami
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106-9560, USA

Abstract

In this paper, we propose simple and efficient algorithms for sorting on incomplete meshes. No hardware redundancy is required and no assumption is made about the availability of a complete submesh. The proposed robust sorting algorithms are very efficient when only a few processors are faulty and degrade gracefully as the number of faults increases. In particular, we show that 1-1 *sorting (*1 *key per healthy processor) in row-major or snakelike row-major order can be performed in* $3n + o(n)$ *communication and comparison steps on an* $n \times n$ *incomplete mesh that has an arbitrary pattern of* $o(\sqrt{n})$ *faulty processors. This is the fastest algorithm reported thus far for sorting in row-major and snakelike row-major orders on faulty meshes and the time complexity is quite close to its lower bound.*

1 Introduction

A d-dimensional mesh consists of $n_1 n_2 \cdots n_d$ processors of degree $2d$ arranged in an $n_1 \times n_2 \times n_3 \times \cdots \times n_d$ grid. When wraparound links are used for all dimensions, a d-dimensional torus results. Because of their scalability, compact layout, constant node-degree, desirable algorithmic properties, and many other advantages, meshes and tori have become popular topologies for the interconnection of parallel processors.

Sorting is one of the most important and useful building blocks in the development of parallel applications. Various algorithms have been developed for sorting on mesh-connected computers [4, 5, 6, 8, 9]. These algorithms usually assume that a fault-free mesh is available. For computing on incomplete meshes, Cole, Maggs, and Sitaraman [1] have shown that an $n \times n$ mesh can be emulated with constant slowdown on an $n \times n$ mesh that has $n^{1-\varepsilon}$ faulty processors for any fixed $\varepsilon > 0$. In [2], Kaklamanis et. al. showed that almost every $n \times n$ p-faulty mesh and any mesh with at most $n/3$ faults can sort n^2 packets in $O(n)$ time. These results are of great theoretical importance but the algorithms are quite complicated and the leading constants for the running times are large. In [7], an elegant but suboptimal robust sorting algorithm based on shearsort has been proposed. However, the robust shearsort can only be executed on meshes with bypass capacity over faulty processors [7].

In this paper, we propose efficient algorithms for sorting on incomplete meshes. No hardware redundancy or bypass capability is required and no assumption is made about the availability of a complete submesh. The proposed algorithms can be executed at high speed in the presence of a small number of faults and degrade gracefully as the number of faults increases. They may even work on meshes whose rows and columns are all incomplete and meshes without any complete submesh. In particular, we show that sorting in row-major or snakelike row-major order can be performed in $3n + o(n)$ communication and comparison steps (excluding precalculation time) on an $n \times n$ bidirectional mesh that has an arbitrary pattern of $o(\sqrt{n})$ faults, assuming that each healthy processor has one of the keys to be sorted. These are the best results reported thus far for sorting on incomplete meshes under the assumed fault conditions and ranking orders. The techniques and results given in this paper can be easily extended to higher dimensional meshes and tori as well as to a variety of other fault-tolerant algorithms, such as semigroup and prefix computation, selection, permutation, fast Fourier transform, and matrix multiplication.

The remainder of the paper is organized as follows. In Section 2, we introduce the basic scheme and several techniques for efficient sorting on incomplete meshes. We also develop simple and efficient algorithms for sorting on a subset of healthy processors which we call a virtual submesh. In Section 3, we derive efficient subroutines for redistributing data from/to all healthy processors to/from virtual submeshes. We then develop a robust sorting algorithm and analyze its complexity for several fault assumptions. In Section 4 we conclude the paper.

0190-3918/97 $10.00

2 Sorting on a subset of healthy processors

In this section, we introduce a simple and efficient scheme based on virtual submeshes for solving various problems on incomplete meshes without hardware redundancy. We then develop several techniques for performing sorting on virtual submeshes with negligible overhead compared with fault-free meshes.

2.1 Virtual submeshes (VSMs)

The basic scheme for the proposed robust sorting algorithms is to redistribute the data on the original mesh to a subset of the healthy processors, which we call a virtual submesh, and then use the virtual submesh to emulate algorithms on the corresponding mesh. In this subsection, we describe the simplest version of virtual submeshes.

We first select a $p_1 \times p_2$ submesh within the original incomplete mesh. The selected submesh is called a *boundary mesh (BM)*. A row (or column) within the boundary mesh that has no faulty processor is called a *complete BM-row* (or a *complete BM-column*, respectively). The $m_1 m_2$ processors at the intersections of the m_1 complete BM-rows and the m_2 complete BM-columns within the boundary mesh form the virtual submesh (VSM). A row (or column) within the boundary mesh with at least one faulty processor is called an *incomplete BM-row* (or an *incomplete BM-column*, respectively). Note that a complete BM-row (or BM-column) may be part of an incomplete row (or column) of the entire mesh. Two examples for simple VSMs are illustrated in Figs. 1 and 2a. Fig. 1 shows a virtual submesh within an $n_1 \times n_2 = 6 \times 7$ incomplete mesh with 6 faulty processors. The shaded circles represent the $m_1 \times m_2 = 3 \times 4$ VSM within the $p_1 \times p_2 = 5 \times 6$ boundary mesh. The numbers in circles represent the logical processor addresses in the VSM. In Fig. 2a, the entire 6×7 mesh is selected as the boundary mesh and the 4×5 VSM is comprised of the 20 shaded nodes.

Let M be the total number of items to be sorted and a be the *load factor*, the maximum number of items per processor, in the VSM. Then we have load factor $a = \left\lceil \frac{M}{m_1 m_2} \right\rceil$ when the data are spread approximately evenly on the VSM. When there is only one item to be sorted per healthy processor and the number of faults is not large, we can have $a = 2$. The load factor for the VSM in Fig. 1 is 3 and the one for the VSM in Fig. 2a is 2, assuming one key per healthy processor.

The proposed basic scheme for performing robust sorting involves 3 stages, as described below. We assume that a preprocessing stage has identified a virtual submesh to be used (perhaps at reconfiguration time).

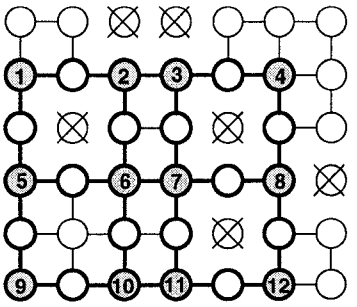

Figure 1. A 3-by-4 VSM represented by shaded circles. The intersections of rows 2–6 and columns 1–6 form the 5-by-6 boundary mesh.

Virtual-Submesh Emulation:

- Stage 1: The data items to be sorted are redistributed evenly to the processors on the VSM such that a processor has at most a items. On the VSM, a processor that has fewer than a items pads its list with ∞ as its "dummy element(s)".
- Stage 2: The VSM emulates a-a sorting on an $m_1 \times m_2$ mesh.
- Stage 3: The sorted data items are redistributed back to healthy processors of the original $n_1 \times n_2$ incomplete mesh with proper ordering.

Since processors of the VSM belong to complete BM-rows and complete BM-columns, a naive method to implement Stage 2 is to directly emulate a transmission over the N (or E, W, S) link of a processor by sending the data item over a path consisting of all the N links between the processor and its *virtual N (or E, W, S) neighbor* in the VSM. When the boundary mesh is complete (that is, no faulty processor exists within it), no degradation is caused by Stage 2 using the naive method.

Let f_{BM} be the total number of faulty processors in the boundary mesh. By using the previous naive method, sorting on the $m_1 \times m_2$ VSM requires $O((a + f_{BM})(m_1 + m_2))$ time in the worst case by emulating an optimal sorting algorithm. In the following subsections, we will develop several techniques to significantly reduce the slowdown factor.

2.2 Compaction/expansion (C/E) techniques

In this subsection, we present a useful technique, which we call the *compaction/expansion (C/E) technique*, that can significantly reduce the time required for sorting on a VSM.

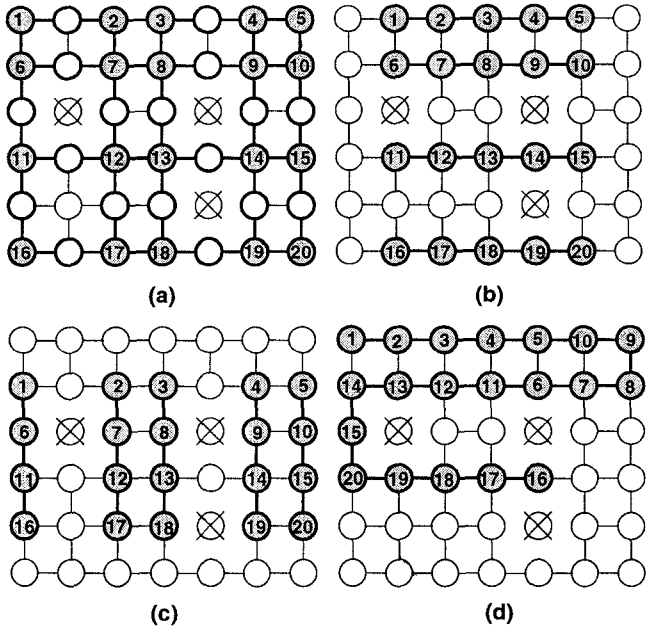

Figure 2. A 4-by-5 VSM (a) and its compacted rows (b), columns (c), and snake (d).

Sorting on each row of the VSM based on the C/E technique involves 4 phases:

C/E-Row Sort (CERS):

- Phase 1 (precalculation): Each complete BM-row performs semigroup and prefix computation to determine the total number t of incomplete BM-columns within the boundary mesh and, for each processor the number l of incomplete BM-columns to its left.
- Phase 2 (compaction): The items in each processor of the VSM are shifted to the left by $l - \lceil t/2 \rceil$ positions if $l - \lceil t/2 \rceil > 0$ and the items are shifted to the right by $\lceil t/2 \rceil - l$ positions if $l - \lceil t/2 \rceil < 0$.
- Phase 3: A row sort is peformed within each of the m_2-node linear arrays (compacted rows).
- Phase 4 (expansion): The sorted items in each of the m_2-node compacted rows are shifted back to processors of the VSM; this is the inverse of Phase 2.

Phase 1 can be done in $O(p_2)$ time using algorithms for semigroup and prefix computation on a fault-free p_2-node linear array. This precalculation phase only needs to be executed once after a new processor or link failure. Phases 2 and 4 can each be done in $a\lceil t/2 \rceil$ time. The integer t is usually small and we have $t < p_2$ and $t \leq f_{BM}$, where f_{BM} is the total number of faults within the boundary mesh. Clearly, Phase 3 can be done in $O(am_2)$ time using odd-even transposition sort, neighborhood sort, or their modified versions.

Compared with sorting on an m_2-node linear array, Phases 1,2 and 4 are the overhead for performing row sort on the VSM. Since Phase 1 is a precalculation phase and only needs to be executed once, Phases 2 and 4 constitute the *effective overhead*. Compared with the naive method which in the worst case has $O(f_{BM}p_2)$ effective overhead, that of CERS is significantly reduced to $O(af_{BM})$. As a case in point, the naive method that does not use the C/E technique has overhead $\Theta(m_2)$ even when there is only one faulty processor within the boundary mesh; while algorithm CERS has overhead $2a$ when there are one or two faulty processors within the boundary submesh. Column sort can be executed in a manner similar to the row sort algorithm CERS. Figure 2 provides an example for sorting on a VSM based on the C/E technique. The shaded circles in Fig. 2a represent a 4×5 VSM within a 6×7 mesh with 3 faulty processors. The shaded circles in Fig. 2b represent the positions of data items for performing row sort based on the C/E technique upon completion of Phase 2 of algorithm CERS. The processors that hold the data elements from a row of the VSM are collectively called a *compacted row*. The number i in a circle represents the position for the data item that was initially held by processor i of the VSM. The shaded circles in Fig. 2c represent the positions of data items for performing column sort based on the C/E technique. The processors that hold the data elements from a column of the VSM form a *compacted column*.

Sorting $2m_2$ elements on an m_2-node bidirectional linear array requires m_2 communication steps and $2m_2$ comparison steps by directly emulating odd-even transposition sort on a $2m_2$-node linear array [4, 6]. As a result, algorithm CERS can be performed using $m_2 + o(m_2)$ communication steps and $2m_2$ comparison steps (excluding precalculation time) when $a = 2$ and $f_{BM} = o(m_2)$. Clearly, when $f_{BM} = o(m_2)$, the slowdown factor for row sort on the VSM is $1 + o(1)$ for any fault pattern.

2.3 A simple sorting algorithm on VSMs

By using the C/E technique, sorting on a VSM can be easily done by emulating shearsort on meshes. More precisely, we can sort on the VSM by performing row sort in Phases $1, 3, 5, ..., 2\log_2 m_1 - 1$ and column sort in Phases $2, 4, 6, ..., 2\log_2 m_1$, using algorithm CERS and the column-sort version of algorithm CERS. Since each pair of steps can be done in $O(a(m_1 + m_2 + t))$ time, sorting on an $m_1 \times m_2$ VSM can be done in $O(a(m_1 + m_2 + t)\log m_1)$ time.

Similarly, if we emulate Revsort [9] using the C/E technique, sorting on VSMs can be performed in $O(a(m_1 + m_2 + t)\log\log m_1)$ time. The resultant overhead is negligible when $f_{BM} = o(\min(m_1, m_2))$ (that is, the overhead is only $o(a(m_1 + m_2)\log m_1)$ compared with shearsort and only $o(a(m_1 + m_2)\log\log m_1)$ compared with Revsort on an $m_1 \times m_2$ complete mesh).

2.4 Odd-even transposition on the snakelike path

Although sorting based on emulating shearsort is simple, the required time is suboptimal. To obtain optimal sorting algorithms on VSMs, we have to emulate optimal sorting algorithms on meshes. These algorithms need to perform odd-even transposition sort on the overall snakelike path and may result in significant degradation using the naive method described in Subsection 2.1. In this subsection, we present a more complicated compaction/expansion process for odd-even transposition on the overall snakelike path of a VSM.

Similar to the CERS algorithm, the items to be sorted will be "compacted" onto part of the snakelike path consisting of all the m_1 complete subrows and the processors bridging two complete subrows if they are not physically contiguous. For simplicity, we consider sending the items to the first $m_1 m_2$ processors of the snake. Performing k steps of odd-even transposition along the overall snake of the VSM based on the compaction/expansion process involves 4 phases:

C/E-Snake Odd-Even Transposition (CEST):

- Phase 1: (precalculation) Prefix computation is performed along the snakelike path of the VSM to determine, for each processor, the number u of processors before it that do not have any item to be sorted.
- Phase 2: (compaction) The items in each processor are sent to the processor u positions before it.
- Phase 3: k odd-even transposition steps are performed along the $m_1 m_2$-node subsnake (compacted snake).
- Phase 4: (expansion) The sorted items in the $m_1 m_2$-node compacted snake are sent back to processors belonging to the VSM; this is the inverse of Phase 2.

Figure 2d shows a *compacted snake* of a VSM, which is comprised of the processors that hold the data elements from the entire snake of the VSM. The numbers in circles represent the original positions of the data items in the VSM.

Phase 1 can be performed using a variant of parallel prefix computation on a mesh by ignoring the processors on incomplete BM-rows except for those on the rightmost (or a middle) complete BM-column. Note that the prefix values of processors in a BM-row is computed either from left to right or from right to left according the direction of the snake. This precalculation phase requires $O(p_1 + p_2)$ time. If we have the results of the precalculation phases for both algorithm CERS and its column-sort version available, the prefix values for Phase 1 can be determined directly by the numbers of incomplete BM-rows and BM-columns and the position of a processor in $O(1)$ time.

If we implement Phase 2 by shifting the items along the snake, the time required can be as large as $\Theta(f_{BM} m_1)$ in the worst case. One way to implement Phase 2 in considerably shorter time is to first send each item to the complete BM-row to which it belongs or the immediately following complete BM-row if the new position of the item is not on a complete BM-row. Then we route data items on each complete BM-row and the vertical segment before it, until the addresses of processors originally holding the items are in ascending order and each processor (except for the last u processors along the snakelike path) has a items. Upon completion, all the items to be sorted along the snake of the VSM are redistributed to a subsnake of length $m_1 m_2$ within the snakelike path of the boundary mesh. Phase 2 can thus be performed in $O(f_{BM})$ (for routing on BM-columns) + $\max(p_2, ap_2/2) + O(f_{BM})$ (for routing on BM-rows and vertical segments) = $\max(p_2, ap_2/2) + O(f_{BM})$ steps.

Thus, Phase 2 and its inverse phase, Phase 4, can each be done in $p_2 + o(p_2)$ time when $a = 2$ and $f_{BM} = o(p_2)$. Phase 3 clearly requires $O(k)$ time.

Based on these C/E techniques, various optimal algorithms for sorting on VSMs can be obtained by simply emulating optimal sorting algorithms (e.g., several recursive sorting algorithms and the (modified) Schnorr/Schamir sorting algorithm [5, 6, 9]).

2.5 General VSMs

In some cases, VSMs of the type used thus far may become quite small with a relatively small number of faulty processors, leading to a large load factor a, and thus a significant slowdown, in emulating mesh algorithms. In such cases, we can use *pairwise complete rows* and *pairwise complete columns* to simulate complete rows and columns, respectively. We briefly introduce the techniques as follows.

Pairwise complete columns are defined as two adjacent columns that contain at least one path from the top row to the bottom row. Pairwise complete rows are defined analogously. We can then define a more general version of VSMs by selecting a processor from each of the intersections of these (pairwise) complete rows and columns. An example is shown in Fig. 3. Detection of a pairwise complete column can be easily done as follows. By sending two signals originating from the left and right columns in the column pair, up to two paths are obtained if the signals do not switch column

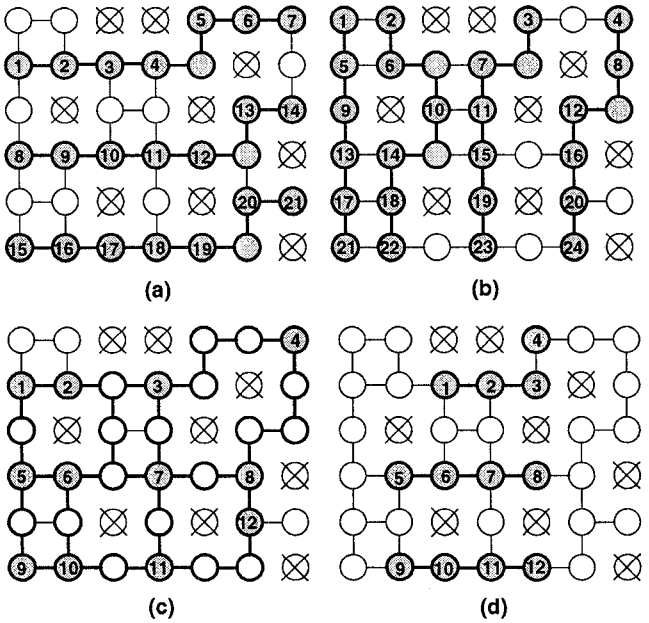

Figure 3. Pairwise complete rows/columns. (a) Pairwise complete rows. (b) Pairwise complete columns. (c) A resultant 3-by-4 VSM. (d) Compacted rows of the VSM.

unless a faulty processor or link is encountered. At least one of the two constructed paths is guaranteed to have the shortest length among all possible paths within the two columns. To emulate a complete column on a pairwise complete column, we can use the C/E technique on the constructed path in a manner similar to algorithm CERS. Figure 3d provides an example where each row of a VSM is compacted to speed up row sort.

This technique can be easily generalized to utilize all (reasonably short) nonoverlapping paths from the top BM-row to the bottom BM-row and nonoverlapping paths from the leftmost BM-column to the rightmost BM-column to obtain a larger VSM at the intersection of these paths. The top/bottom row and the leftmost/rightmost column of a boundary mesh can also be paths that are not straight. The analysis for algorithms on the simplest version of VSMs can be extended to the general version of VSMs by substituting p_1, p_2, and t with $p_{1,max}$, $p_{2,max}$, and t_G, respectively, where $p_{1,max}$ and $p_{2,max}$ are the maximum lengths of column paths and row paths, respectively, within the generalized boundary mesh, and $t_G = \max(p_{1,max} - m_1, p_{2,max} - m_2)$. When $p_{1,max} = \Theta(m_1)$ and $p_{2,max} = \Theta(m_2)$, optimal sorting algorithms for the VSM can be obtained by simply emulating optimal sorting algorithms on meshes using C/E techniques.

3 Robust sorting on incomplete meshes

In this section, we derive fast algorithms to perform 1-1 sorting on an $n \times n$ incomplete mesh that has $f = o(\sqrt{n})$ faulty processors, where each healthy and connected processor holds one of the keys to be sorted.

3.1 Mapping an incomplete mesh onto a VSM

In this subsection, we describe how to select a proper VSM and map the incomplete mesh onto it.

We use the entire incomplete mesh as the boundary mesh. We select $m_1 = n - o(n)$ complete rows and the middle $m_2 = n/2 + o(n)$ complete columns, such that $m_1 \times m_2 \geq (n^2 - f)/2$. We also require that the selected complete rows be separated by no more than $f + 1$ hops if some complete rows are not selected. Then the intersections of the selected m_1 rows and m_2 columns form a desired VSM. Since there are no more than $o(\sqrt{n})$ faulty processors, the existence of such VSMs is guaranteed. Obviously, the number of items per processor is $a = 2$ for 1-1 sorting on such incomplete meshes. For simplicity of algorithm description, we assume that $\sqrt{n}$ is an integer. We call each of the n $\sqrt{n}$-by-$\sqrt{n}$ submeshes of the incomplete mesh a *block*, and a $\sqrt{n}$-node complete row (or column) within a block a *complete block-row* (or *complete block-column*). A block is crossed by at least $\sqrt{n} - f$ complete block-columns and complete block-rows.

To sort the items in row-major order, we first perform a prefix computation in row-major order to determine the number of healthy processors that precede each of the healthy processors. Then the i^{th} healthy processor is mapped onto the $\lceil i/2 \rceil^{th}$ processor in the VSM in snakelike row-major order. Figure 4 illustrates such a mapping for a 6×7 mesh with 3 faulty processors (as shown in Fig. 4a) onto a 4×5 VSM (Fig. 4d). To sort the items in snakelike row-major order, blockwise order, or other orderings, we perform a prefix computation in the respective order, and map each healthy processor onto the VSM in snakelike row-major order.

3.2 Data redistribution

In this subsection, we introduce an efficient algorithm for performing *data redistribution*, which moves data from healthy processors, each having one data item, to the corresponding processors in the VSM. We then analyze its performance and show that it is optimal for row-major and snakelike row-major mapping orders.

The algorithm DR for data redistribution is comprised of 4 phases:

Data Redistribution (DR):

- Phase 1: In each block, all data items are routed to a nearby complete block-row.
- Phase 2: In each block, all data items are spread approximately evenly along the block-row onto processors at the intersections of complete columns and complete block-rows.
- Phase 3: Each data item is sent along the complete column to which it currently belongs to the complete row to which the data item will belong in the VSM.
- Phase 4: Each data item is sent along the complete row to which it currently belongs to the desired position in the VSM.

Phase 1 can be done by first routing any data item to one of the complete block columns/rows that surround the item, and then routing it along the complete block-columns to a nearby complete block-row within its block. The desired location for each data item at the end of Phase 2 can be determined by performing prefix computation in each complete block-row, which is a precalculation step and requires only $O(\sqrt{n})$ time. If a processor in a complete column has a constant number of data items at the end of the initial step for Phase 1, it can skip the latter step and Phase 2 without increasing the leading constant of the running time. Figure 4 provides an example for moving data from a 6×7 incomplete mesh with 3 faulty processors (Fig. 4a) to a 4×5 VSM (Fig. 4d) using algorithm DR. In Fig. 4b, the original incomplete mesh is partitioned into 6 blocks. The number i in a circle represents the current position for the data item that was held by processor i of the original incomplete mesh upon completion of Phase 2. Note that processors belong to complete columns have skipped the second step of Phase 1 and Phase 2. Figure 4c shows the intermediate positions for data items upon completion of Phase 3.

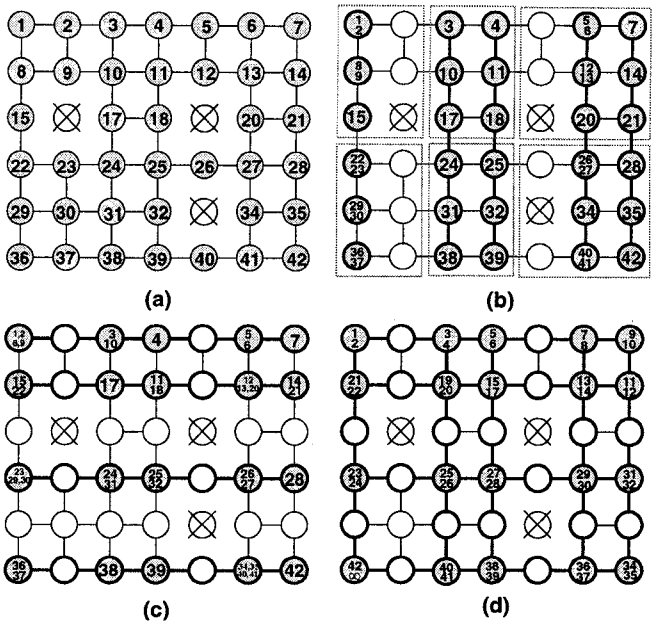

Figure 4. Data redistribution from the incomplete mesh to a VSM using algorithm DR. (a) The 39 data elements. (b) Positions after Phase 2. (c) Positions after Phase 3. (d) The final positions.

Lemma 3.1 *Data redistribution from an $n \times n$ incomplete mesh with $o(\sqrt{n})$ faulty processors in row-major or snakelike row-major orders onto an appropriate VSM in snakelike row-major order (as described in Subsection 3.1) can be performed in $3n/4 + o(n)$ steps.*

Proof: Since complete block rows/columns are separated by no more than $o(\sqrt{n})$ hops, at most $o(n)$ data items are surrounded by nearby complete block-columns and complete block-rows. Therefore, the first step of Phase 1 of algorithm DR can be executed in $o(n)$ time, and at the end of this step, no more than $o(n)$ data items will be located between 2 nearest intersection nodes of complete block rows and columns along a column. Since the distance between two complete block-rows is $o(\sqrt{n})$, the second step of Phase 1 requires $o(\sqrt{n}) + o(n) = o(n)$ time. Since no more than $o(n)$ data items will be routed along a complete block-row during Phase 2 and the length of a block-row is $\sqrt{n}$, Phase 2 requires no more than $o(n)$ time. Upon completion of Phase 2, there are $\sqrt{n} + o(\sqrt{n})$ data items in every complete block-column. Since an item only needs to be routed for no more than $o(n)$ hops during Phase 3, this phase requires $o(n)$ time. The maximum distance for a data item to be routed during Phase 4 is $3n/4 + o(n)$. For row-major and snakelike row-major orders, a $\sqrt{n}$-node complete block-row will holds no more than $2\sqrt{n}$ items at the end of Phase 3 since it is impossible for data items from three different rows to be mapped onto the same row of the VSM for either of the orders. This property is sufficient to show that the maximum number of items that can cross a processor in one direction during Phase 4 is no more than $3n/4 + o(n)$. The worst possible case occurs at a processor X in the $\frac{5n}{8}$-th column of the incomplete mesh (i.e., the $(\frac{3n}{8} \pm o(n))$-th column in the VSM), when the healthy processors at the intersections of two nearby rows and columns $5n/8 + 1, 5n/8 + 2, \ldots, n$ are mapped to processors that are to the left of processor X in the VSM. As a result, Phase 4 can be performed in $3n/4 + o(n)$ time for row-major and snakelike row-major orders. □

Note that for some ranking orders (e.g., blockwise order), the maximum number of items that will cross a processor

in one direction during Phase 4 is $n + o(n)$. Therefore, algorithm DR requires $n + o(n)$ time for such ranking orders. We can, however, modify the data-redistribution algorithm to reduce its running time to $3n/4 + o(n)$ for blockwise ordering.

By reversing the process of algorithm DR, data redistribution from a VSM to all healthy processors can be done in the same time for respective ranking order.

If we use algorithm DR and its inverse process based on the previous mapping orders as subroutines to perform 1-1 robust sorting, Stages 1 and 3 of virtual-submesh emulation require $3n/2 + o(n)$ communication steps. The required time can be reduced to $n + o(n)$ by using different mapping strategies for Stage 1 of virtual-submesh emulation. In what follows, we describe an algorithm for obtaining such mapping and performing data redistribution.

Mapping and Data Redistribution (MDR):

- Phase 1: In each block, all data items are routed to a nearby complete block-column.
- Phase 2: In each block, all data items are spread approximately evenly along the block-column onto processors at the intersections of complete rows and complete block-columns.
- Phase 3: Each data item is sent along the complete row to which it currently belongs until each processor of the middle $n/2 + o(n)$ complete columns has 2 items (except for the leftmost or rightmost ones).
- Phase 4: An appropriate VSM has to include all the processors that currently have at least one item. If a processor in the VSM has fewer than 2 items, it is padded with dummy values ∞.

Lemma 3.2 *Data redistribution from an $n \times n$ incomplete mesh with $o(\sqrt{n})$ faulty processors onto an $(n - o(n)) \times (n/2 + o(n))$ VSM can be optimally executed in $n/4 + o(n)$ steps.*

Proof: From algorithm MDR, it can be seen that the row number of the new position in the VSM for a data item is at most $o(\sqrt{n})$ from the row number of its original position, and the column number of the new position in the VSM for a data item is at most $n/4 + o(n)$ from that of its original position. Moreover, data elements can be concentrated at the middle $n/2 + o(n)$ complete columns in a pipelined manner so that no more than $n/4 + o(n)$ data elements need to be sent across a link. The required time is clearly $n/4 + o(n)$ for algorithm MDR. □

3.3 A $3n$-step robust sorting algorithm

In this subsection, we show that 1-1 sorting can be performed in $3n + o(n)$ communication and comparison steps on an $n \times n$ incomplete mesh that has an arbitrary patterns of $o(\sqrt{n})$ faults.

For simplicity of algorithm description, we choose m_1 and m_2 to be the sixth powers of integers. (There exist $o(n^{1/6})$ possible integers for both m_1 and m_2.) The boundary mesh for the VSM is partitioned into $(m_1 m_2)^{1/6}$ blocks, which we call *VSM blocks*. More precisely, each VSM block contains exactly $(m_1 m_2)^{5/6}$ processors of the original VSM arranged as an $m_1^{5/6}$-by-$m_2^{5/6}$ small virtual submesh, but may have more than $m_1^{5/6} m_2^{5/6}$ processors in it. A "column of VSM blocks" is called a *vertical VSM slice*.

The proposed robust sorting algorithm first redistributes the data items to the VSM, then emulates the Schnorr/Schamir sorting algorithm [5, 9] modified for 2-2 sorting using larger blocks on the VSM, and finally redistributes sorted data back to healthy processors.

The robust sorting algorithm is composed of 10 phases:

Robust Sorting (RS):

- Phase 0: Redistribute data items from each of the healthy processors in the incomplete mesh to an appropriate VSM using algorithm MDR.
- Phase 1: Sort each VSM block.
- Phase 2: Perform an $m_2^{1/6}$-way unshuffle of the VSM-columns. (That is, permute the columns such that the $m_2^{5/6}$ columns in each VSM block are distributed evenly among the $m_2^{1/6}$ vertical VSM slices.)
- Phase 3: Sort each VSM block into snakelike row-major order.
- Phase 4: Sort each column of the VSM in linear order using the C/E technique. (That is, sort the $2m_1$ data items in each VSM-column as an $m_1 \times 2$ mesh in row-major order.)
- Phase 5: Collectively sort VSM blocks 1 and 2, VSM blocks 3 and 4, VSM blocks 5 and 6,..., of each vertical VSM slice into snakelike row-major order.
- Phase 6: Collectively sort VSM blocks 2 and 3, VSM blocks 4 and 5, VSM blocks 6 and 7,..., of each vertical VSM slice into snakelike row-major order
- Phase 7: Sort each row of the VSM in linear order according to the direction of the overall snake using algorithm CERS.

- Phase 8: Perform $2\sqrt{m_1}$ odd-even transposition steps on the overall snake of the VSM (without using the C/E technique).

- Phase 9: Redistribute each of the data items from the VSM in snakelike row-major order to the appropriate healthy processor in the incomplete mesh in row-major order for row-major sorting (or in snakelike row-major order for snakelike row-major sorting) using the inverse process of algorithm DR.

Theorem 3.3 1-1 *sorting (1 key per healthy and connected processor) in row-major or snakelike row-major order on an $n \times n$ bidirectional mesh that has an artibrary pattern of $o(\sqrt{n})$ faulty processors can be performed in $3n + o(n)$ communication and comparison steps (excluding precalculation time).*

Proof: The correctness of algorithm RS can be proved using 0-1 principle [3] and the proof is similar to those given in [5, pp. 148–151] and [9]. From Lemmas 3.1 and 3.2, Phases 0 and 9 can be performed in $n/4 + o(n)$ and $3n/4 + o(n)$ communication steps, respectively. To perform 2-2 sorting on all VSM blocks for Phases 1 and 3 of algorithm RS (or VSM block pairs for Phases 5 and 6) in parallel, each VSM block (or VSM block pair, respectively) is sorted by emulating shearsort on it. These phases require $O(n^{5/6} \log n)$ time as shown in Subsection 2.3. Phase 2 requires $n/2 + o(n)$ communication steps since the width of the VSM is $n/2 + o(n)$ hops and no more than $n/2 + o(n)$ items will cross a link in the same direction during this phase. From the analysis given in Subsection 2.2, we know that Phase 4 requires $m_1 + o(n) = n \pm o(n)$ communication steps and $2m_1 = 2n - o(n)$ comparison steps while Phase 7 requires $m_2 + o(n) = n/2 + o(n)$ communication steps and $n + o(n)$ comparison steps (without precalculation time). Since two neighboring processors of the VSM are separated by no more than $f + 1 = o(\sqrt{n})$ hops, Phase 8 requires at most $o(n)$ communication steps and $O(\sqrt{n})$ comparison steps by directly performing the odd-even transposition steps (that is, by using the naive method mentioned at the end of Subsection 2.1). □

When data are input/output to/from VSMs directly, sorting can be performed in $2.5n + o(n)$ communication steps and $3n + o(n)$ comparison steps on an $n \times n$ VSM within an incomplete mesh that has an arbitrary pattern of $o(\sqrt{n})$ faults. This result can be easily generalized to incomplete meshes with $o(n)$ faults by using larger VSM blocks. The running time has the same leading constant as the best known algorithms for 1-1 sorting on an $n \times n$ fault-free mesh [4, 6].

We can generalize algorithm RS for incomplete meshes with larger f by using larger blocks for Phases 0 and 9 (i.e., algorithms MDR and DR). The time required for sorting on an $n \times n$ incomplete mesh with f faults is $O(n + f^2)$, where $f < (1 - \varepsilon)n$ for any fixed $\varepsilon > 0$. The extra $O(f^2)$ communication steps are required by algorithms DR and MDR for worst-case fault patterns.

4 Conclusion

In this paper, we have proposed efficient robust algorithms for sorting on incomplete meshes. The proposed algorithms are efficient when the number of faults is not large and degrade gracefully as the number of faults increases. In particular, we showed that sorting on an $n \times n$ incomplete mesh that has $o(\sqrt{n})$ faulty processors can be performed in $3n + o(n)$ communication and comparison steps. This is the fastest algorithm reported thus far for sorting on an incomplete mesh in row-major and snakelike row-major orders. These techniques can also be applied to a variety of other important problems to obtain robust algorithms with negligible overheads. Details of these results will be reported in the future.

References

[1] Cole, R., B. Maggs, and R. Sitaraman, "Multi-scale self-simulation: a technique for reconfiguring arrays with faults," *ACM Symp. Theory of Computing*, 1993, pp. 561-572.

[2] Kaklamanis, C., A.R., Karlin, F.T. Leighton, V. Milenkovic, P. Eaghavan, S. Rao, C. Thomborson, and A. Tsantilas, "Asymptotically tight bounds for computing with faulty arrays of processors," *Proc. Symp. Foundations of Computer Science*, vol. 1, 1990, pp. 285-296.

[3] Knuth, D.E., *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, Reading, Mass., Addison-Wesley, 1973.

[4] Kunde, M. "Concentrated regular data streams on grids: sorting and routing near to the bisection bound," *Proc. Symp. on Foundations of Computer Science*, 1991, pp. 141-150.

[5] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan-Kaufman, San Mateo, CA, 1992.

[6] Nigam, M. and S. Sahni, "Sorting n^2 numbers on $n \times n$ meshes," *IEEE Trans. Parallel Distrib. Sys.*, vol. 6, no. 12, Dec. 1995, pp. 1221-1225.

[7] Parhami B. and C.-Y. Hung, "Robust shearsort on incomplete bypass meshes," *Proc. Int'l Parallel Processing Symp.*, 1995, pp 304-311.

[8] Park, A. and K. Balasubramanian, "Reducing communication costs for sorting on mesh-connected and linearly connected parallel computers," *J. Parallel Distrib. Comput.*, vol. 9, no. 3, Jul. 1990 pp. 318-322.

[9] Schnorr, C.P. and Shamir, A., "An optimal sorting algorithm for mesh connected computers," *Proc. Symp. Theory of Computing*, 1986, pp. 255-263.