2.5n-Step Sorting on $n \times n$ Meshes in the Presence of $o(\sqrt{n})$ Worst-Case Faults

Chi-Hsiang Yeh, Behrooz Parhami, Hua Lee, and Emmanouel A. Varvarigos
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106-9560, USA

Abstract

In this paper, we propose the robust algorithm-configured emulation (RACE) scheme for the design of simple and efficient robust algorithms that can run on faulty mesh-connected computers. We show that 1-1 *sorting (1 key per healthy processor) can be performed in* $2.5n + o(n)$ *communication steps and* $2n + o(n)$ *comparison steps on an* $n \times n$ *mesh with an arbitrary pattern of* $o(\sqrt{n})$ *faults. This running time has exactly the same leading constant as the best known algorithms for* 1-1 *sorting on an* $n \times n$ *fault-free mesh. We also formulate the mesh robustness theorem, which leads to a variety of efficient robust algorithms on faulty meshes.*

1. Introduction

A d-dimensional mesh consists of $n_1 n_2 \cdots n_d$ processors of degree $2d$ arranged in an $n_1 \times n_2 \times \cdots \times n_d$ 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 the most popular topologies for the interconnection of parallel processors.

Since a parallel computer consists of a complex assembly of many components, the probability that some fraction of the system fails is nonnegligible. A large volume of literature exist on the subject of fault tolerance in parallel systems [2, 3, 5, 8, 9, 11, 12, 13]. For computing on faulty meshes, Cole, Maggs, and Sitaraman [3] 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 [5], 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. In [8], an elegant but suboptimal robust sorting algorithm based on shearsort has been proposed for meshes with bypass capacity. In [11, 13], we showed 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 $n \times n$ mesh with $o(\sqrt{n})$ faults.

In this paper, we tackle the fault tolerance issue in meshes and tori using the *robust-algorithm approach* [9], which incorporates fault tolerance into the design of algorithms. We assume the *removal model* [9], where a faulty processor or link is removed from the network. No hardware redundancy or bypass capability is required and no assumption is made about the availability of a complete submesh. We propose the *robust algorithm-configured emulation (RACE) scheme* for the design of general robust algorithms. We show that sorting can be performed in $2.5n + o(n)$ communication steps and $2n + o(n)$ comparison steps (excluding precalculation time which is needed only once following each configuration change) on an $n \times n$ bidirectional mesh that has an arbitrary pattern of $o(\sqrt{n})$ faults, assuming that each healthy and connected processor has one of the keys to be sorted. Surprisingly, this running time has exactly the same leading constant as the best known algorithms for 1-1 sorting on an $n \times n$ fault-free mesh and is the best result reported thus far for sorting on faulty meshes for any ranking order.

We derive the *mesh robustness theorem*, which shows that the slowdown factor for performing a algorithm on a subset of healthy processors of a faulty mesh is $1 + o(1)$ relative to a fault-free $(n - o(S)) \times (n - o(S))$ mesh if the corresponding algorithm in a fault-free mesh performs S consecutive routing steps along the same dimension on the average, and there are at most $o(S)$ faulty processors in the mesh. Our results demonstrate that meshes and tori are robust in that they can solve many problems efficiently even when many processors and/or links fail.

2. The robust algorithm-configured emulation (RACE) scheme

In this section, we define the notion of virtual submeshes in faulty meshes and introduce a simple and efficient scheme for solving various problems on faulty meshes, without relying on hardware redundancy. We then develop several techniques for sorting on virtual submeshes with negligible overhead compared with fault-free meshes.

2.1. Definition of virtual Submeshes (VSMs)

A *virtual submesh (VSM)* of a d-D faulty mesh is obtained by embedding a smaller d-D mesh in it, where the embedded rows of the same dimension do not overlap and the embedded nodes and links are mapped onto healthy nodes and paths. More precisely, each node of this smaller mesh is mapped onto a different healthy node of the faulty mesh; each link of this smaller mesh is mapped onto a healthy path of the faulty mesh. The embedded rows (or columns) of a certain dimension i, $i = 1, 2, \ldots, d$, do not overlap with each other, and are called *dimension-i virtual rows* (or *virtual columns*) of the virtual submesh. Node $(x_1, x_2, \ldots, x_n)$ of the obtained virtual submesh (called a *VSM node*) is located at the intersection of virtual row x_1 of dimension 1, virtual row x_2 of dimension 2, ... , and virtual row x_n of dimension n. Note that these virtual rows of different dimensions are allowed to have more than one node in common, in which case we select one of the nodes at the intersection either arbitrarily or according to certain criteria (e.g., the dilation of the resultant embedding). Then, VSM nodes $(x_1, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots, x_n)$ for certain x_j, $j \neq i$, and all $y_i = 1, 2, \ldots, m_i$ form a *dimension-i row* of the virtual submesh and is also called a *VSM row*, where m_i is the length of a dimension-i VSM row. A *virtual subtorus* is defined analogously. Figure 1 shows a 3×4 virtual submesh in a mesh with 9 faults and the virtual rows and virtual columns of the virtual submesh. Figure 4b is an example of a 7×10 virtual submesh in a 9×12 mesh with 9 faulty processors.

There usually exist many virtual submeshes in a faulty mesh. In general, to achieve better performance, we prefer to maximize the number of VSM nodes and minimize,

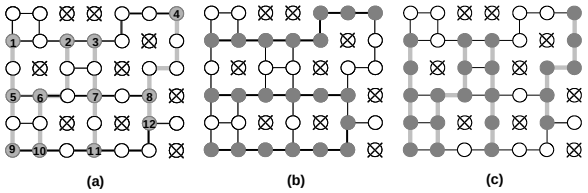

Figure 1. (a) A 3-by-4 VSM in a faulty 6-by-7 mesh with 9 faults. (b) Virtual rows of the VSM. (c) Virtual columns of the VSM.

for each dimension, the difference between the maximum length of virtual rows and the length m_i of a VSM row. The largest of these difference for all dimensions is called the *width overhead* of the virtual submesh. Clearly, when there are $o(n)$ faulty processors or links in a d-D $n \times n \times \cdots \times n$ mesh, it is guaranteed that a d-D $(n-o(n)) \times (n-o(n)) \times \cdots \times (n-o(n))$ virtual submesh with width overhead $o(n)$ exists. The simplest way to find such a virtual submesh is to select all the fault-free rows and columns as virtual rows and columns, though a larger virtual submesh may exist. When there are $o(n^2)$ random faults in a 2-D $n \times n$ mesh, we can show that a 2-D $(n-o(n)) \times (n-o(n))$ virtual submesh with width overhead $o(n)$ exists with high probability. The details will be reported in the near future.

2.2. The RACE scheme and the stepwise emulation technique (SET)

In this subsection we present the *robust algorithm-configured emulation (RACE) scheme*, which redistributes the data on a faulty network to a virtual subgraph (e.g., an $m_1 \times m_2$ virtual submesh), and then uses the virtual subgraph to emulate algorithms developed for a fault-free network. We also present the stepwise emulation technique (SET) for running mesh algorithms on virtual submeshes.

Let M be the total number of data items and a be the *load factor*, the maximum number of items per processor, in the virtual submesh. Then we have $a = \left\lceil \frac{M}{m_1 m_2} \right\rceil$ when the data are spread approximately evenly on the virtual submesh.

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

The RACE Scheme

- Stage 1: The data items to be processed are redistributed evenly to the processors on the virtual submesh such that a processor has at most a items. On the virtual submesh, a processor that has fewer than a items may pad its list with suitable "dummy element(s)" (e.g., ∞ for sorting).
- Stage 2: The virtual submesh emulates a corresponding algorithm on an $m_1 \times m_2$ mesh, each processor of which has at most a items.
- Stage 3: The results are redistributed back to healthy processors of the original $n_1 \times n_2$ faulty mesh.

Stage 2 of the RACE scheme can be implemented using the *stepwise emulation technique (SET)*, which directly emulates a transmission over the dimension-i link of a processor by sending the data item along the dimension-i virtual row to which the processor belongs.

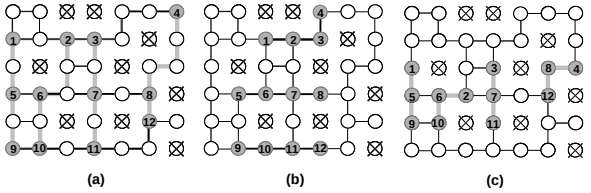

Figure 2. (a) A 3-by-4 VSM. (b) Compacted rows of the VSM. (c) Compacted columns of the VSM.

2.3. The compaction/expansion technique (CET)

In this subsection, we present the *compaction/expansion technique*, abbreviated *C/E technique* or *CET*, which can significantly reduces the time required for sorting and a variety of other algorithms on a virtual submesh.

Sorting on each row of the same dimension on a virtual submesh using CET (see Fig. 2 for an example) involves 4 phases:

CET Row Sort:

- Phase 0 (precalculation): Each dimension-i virtual row performs semigroup and prefix computation to determine the total number t of processors in the virtual row, and, for each VSM node, the number l of processors to its left that are not VSM nodes.
- Phase 1 (compaction): The items in each VSM node 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 2: A row sort is performed within each compacted row (the virtual subrow composed of the m_i neighboring nodes currently holding the data.
- Phase 3 (expansion): The sorted items in each of the m_i-node compacted rows are shifted back to VSM nodes; this is the inverse of Phase 1.

Phase 0 can be done in $O(m_i + t_{max})$ time using algorithms for semigroup and prefix computation on a virtual row [9]. This precalculation phase only needs to be executed once after each new processor or link failure. Sorting $2m_i$ items on an m_i-node bidirectional linear array requires m_i communication steps and $2m_i$ comparison steps by directly emulating odd-even transposition sort on a $2m_i$-node linear array [6, 7]. As a result, algorithm CET Row Sort can be performed using $m_i + o(m_i)$ communication steps and $2m_i$ comparison steps (excluding precalculation time) when $a = 2$ and $t_{max} = o(m_2)$. Clearly, when $f_B = o(m_i)$, the slowdown factor for row sort on the virtual submesh is $1 + o(1)$ for any fault pattern.

3. Robust sorting on faulty meshes

In this section, we derive fast algorithms to perform 1-1 sorting on an $n \times n$ 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 a faulty mesh onto a VSM

In this subsection, we describe how to select a proper virtual submesh and map the faulty mesh onto it.

We select nonoverlapping horizontal paths (or vertical paths) that extend from the leftmost column (or top row,

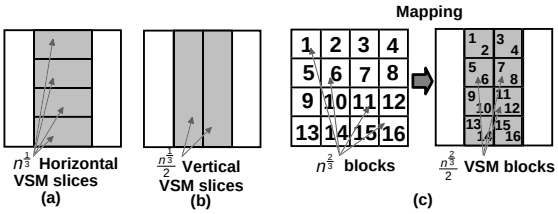

Figure 3. (a) The horizontal slices of a virtual submesh. (b) The vertical slices of a virtual submesh. (c) The intersections of these 4 horizontal slices and 2 vertical slices form 8 VSM blocks. The 16 blocks of the faulty mesh are mapped onto the 8 VSM blocks.

respectively) to the rightmost column (or bottom row, respectively) of the faulty mesh. Then the intersections of $m_1 = n - o(\sqrt{n})$ horizontal paths and the middle $m_2 = n/2 + o(\sqrt{n})$ vertical paths form a virtual submesh. For simplicity of algorithm description, we assume that $n^{2/3}$ is even. Then we partition the virtual submeshes into $n^{1/3}$ *horizontal VSM slices* and also into $n^{1/3}/2$ *vertical VSM slices*. There are $n^{2/3}/2$ *VSM blocks*, each located at the intersections of a horizontal VSM slice and a vertical VSM slice. Figure 3 illustrates a virtual submesh with 4 horizontal slices, 2 vertical slices, and 8 VSM blocks. We have to properly select m_1 and m_2 so that each of the VSM blocks contains at least $n^{4/3}$ nodes. Since there are no more than $o(\sqrt{n})$ faulty processors, the existence of such virtual submeshes is guaranteed. We call each of the $n^{2/3}$ $n^{2/3}$-by-$n^{2/3}$ submeshes of the faulty mesh a *block*. Then we can map the faulty mesh onto the virtual submesh by mapping two neighboring blocks of the faulty mesh onto a corresponding VSM block. Figure 3c shows such a mapping from the faulty mesh with 16 blocks to a virtual submesh with 8 VSM blocks.

The number of items per processor of the virtual submesh is at most $a = 2$ (for 1-1 sorting on such a faulty mesh). We call data items from blocks 1, 3, 5, ... the *first layer* of data, and data items from blocks 2, 4, 6, ... the *second layer* of data. A block is crossed by at least $n^{2/3} - O(f)$ horizontal paths and vertical paths. The segment of a horizontal (or vertical) path within a block is called a *virtual block-row* (or *virtual block-column*, respectively).

Note that we assume that n is the third power of an even number only for the simplicity of algorithm description. The algorithms presented in this section can be easily extended to more general $n_1 \times n_2$ meshes. Figure 4a illustrates an 8×8 block and its virtual block-columns; figure 4b illustrates a 7×10 VSM block which contains 3 faulty processors within its boundary. Two 8×8 blocks can be mapped onto a 7×10 VSM block with load factor $a = 2$. Note that in an $n \times n$ mesh with $o(\sqrt{n})$ faults, there is usually no more than one fault within the boundary of a VSM block, and fewer than one VSM block out of $\Omega(n^{1/6})$ VSM blocks on the average contains faults in it.

3.2. Data redistribution (DR)

In this subsection, we introduce an asymptotically optimal algorithm for performing *data redistribution*, which moves data from healthy and connected processors, each having one data item, to the corresponding processors in the virtual submesh.

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

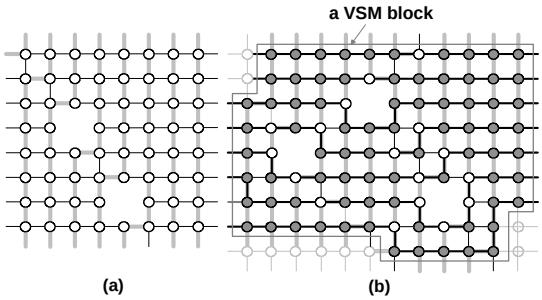

Figure 4. (a) An 8×8 block which contains 2 faulty processors within its boundary. The thick gray lines represent virtual block-columns. (b) A 7×10 VSM block which contains 3 faulty processors within its boundary. The thick dark lines represent virtual rows; the thick gray lines represent virtual columns; the shaded circles represent VSM nodes.

Data Redistribution (DR):

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

Phase 1 can be done by first routing each data item to one of the virtual block-rows/columns that surround the item, and then routing it along the virtual block-columns (if needed) to a nearby virtual 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 virtual block-row, which is a precalculation step and requires only $O(n^{2/3})$ time.

Theorem 3.1 *Data redistribution from an $n \times n$ mesh with $o(\sqrt{n})$ faulty processors onto an appropriate virtual submesh can be performed in $n/4 + o(n)$ steps.*

By reversing the process of algorithm DR, data redistribution from a virtual submesh to all healthy processors can be done in the same time $n/4 + o(n)$.

3.3. Robust sorting algorithms on faulty meshes

In this subsection, we show that 1-1 sorting can be performed in $2.5n + o(n)$ communication steps and $2n + o(n)$ comparison steps on an $n \times n$ mesh that has an arbitrary patterns of $o(\sqrt{n})$ faults.

The proposed robust sorting algorithm uses the RACE scheme, which redistributes the data items to the virtual submesh, then emulates a 2-2 mesh sorting algorithm in blockwise snakelike order on the virtual submesh, and finally redistributes the sorted items back to healthy processors. If we emulate the sorting algorithm proposed in [6], the data items will be sorted by our robust sorting algorithm according to

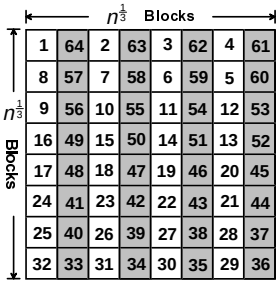

Figure 5. The folded odd/even snake (FOE-snake) order of the blocks of a faulty mesh. Blocks with ascending and descending orders are interleaved, leading to a folded snake with interleaved blocks.

a special ordering, called the *folded odd/even snake (FOE-snake) order* in this paper, where blocks are indexed along a folded blockwise snake that contains odd-numbered blocks in the forward direction interleaved with even-numbered blocks on the return portion of the snake (Fig. 5). More precisely, nodes in blocks $1,3,5,\ldots,n^{1/3}-1$, blocks $2n^{1/3}-1, 2n^{1/3}-3, 2n^{1/3}-5,\ldots,n^{1/3}+1$, blocks $2n^{1/3}+1, 2n^{1/3}+3, 2n^{1/3}+5,\ldots,2n^{1/3}-1$, ..., and blocks $n^{2/3}-1, n^{2/3}-3, n^{2/3}-5,\ldots,n^{2/3}-n^{1/3}+1$ are indexed in standard row-major snake order, followed by nodes in blocks $n^{2/3}-n^{1/3}+2, n^{2/3}-n^{1/3}+4, n^{2/3}-n^{1/3}+6,\ldots,n^{2/3}$, blocks $n^{2/3}-n^{1/3}, n^{2/3}-n^{1/3}-2, n^{2/3}-n^{1/3}-4,\ldots,n^{2/3}-2n^{1/3}+2$, ..., and blocks $n^{1/3}, n^{1/3}-2, n^{1/3}-4,\ldots,2$ indexed in backward row-major snake order. Note that we can use any ordering for nodes within the blocks, leading to different subclasses of the FOE-snake order. Figure 5 illustrates the FOE-snake order in a mesh with 64 blocks.

Stages 1 and 3 of the RACE scheme can be performed using algorithm DR and its inverse process, which collectively require $n/2+o(n)$ communication steps. Since 2-2 sorting can be performed using $n_1+2n_2+o(n_1+n_2)$ communication steps and $2n_1+o(n_1)$ comparison steps on an $n_1 \times n_2$ mesh [6] and this algorithm can be emulated on an $(n-o(\sqrt{n})) \times (n/2+o(n))$ virtual submesh with a factor of $1+o(1)$ slowdown, Stage 2 of the RACE scheme requires $2n+o(n)$ communication and comparison steps, leading to the following theorem.

Theorem 3.2 1-1 *sorting (*1 *key per healthy and connected processor) on an $n \times n$ bidirectional mesh that has an arbitrary pattern of $o(\sqrt{n})$ faulty processors can be performed in $2.5n+o(n)$ communication steps and $2n+o(n)$ comparison steps (excluding precalculation time).*

Proof: We can obtain such a robust sorting algorithm by emulating 2-2 sorting algorithms that require $2.5n+o(n)$ communication steps and $2n+o(n)$ comparison steps on $(n-o(n)) \times (n/2+o(n))$ fault-free meshes. By plugging the sorting algorithm proposed in [6] (as Phases 2 through 7) into the RACE scheme, we can obtain the following robust sorting algorithm.

FOE-Snake Sorting:

- Phase 1: Redistribute data items from each of the healthy and connected processors in the faulty mesh to the virtual submesh using algorithm DR. Pad each VSM node that has fewer than 2 data items with ∞ as its "dummy element(s)".
- Phase 2: Sort all the VSM blocks in parallel using CET.
- Phase 3: Partition each of the horizontal VSM slices into $n^{1/3}$ smaller horizontal slices, and perform *row-to-column mapping* [6] in each of them using SET.
- Phase 4: Sort each of the vertical VSM slices into *layer-last ordering* [6] using CET.
- Phase 5: Perform row-to-column mapping in each of the horizontal VSM slices using SET.
- Phase 6: Sort all neighboring VSM blocks 2 and 3, VSM blocks 4 and 5, VSM blocks 6 and 7, ..., and VSM blocks $n^{2/3}-2$ and $n^{2/3}-1$ using CET according to layer-last blockwise snake-like order [6].
- Phase 7: Sort all neighboring VSM blocks 1 and 2, VSM blocks 3 and 4, VSM blocks 5 and 6, ..., and VSM blocks $n^{2/3}-1$ and $n^{2/3}$ using CET according to *layer-last blockwise snake-like order*.
- Phase 8: Move the sorted items back to the VSM nodes that held data items, rather than dummy element(s), upon the completion of Phase 1.
- Phase 9: Redistribute each of the data items from the virtual submesh to the appropriate healthy processor in the faulty mesh using the inverse of algorithm DR.

Phases 2 through 7 sort the data items on the VSM according to layer-last blockwise snake-like order [6]. We refer the reader to [6] for the details of the algorithm and its proof of correctness. We can see that Phases 8 and 9 preserve the order of the sorted data according to the FOE-snake order. Therefore, following the proof for the sorting algorithm proposed in [6], we can show that FOE-snake sort is correct.

From Theorem 3.1, Phases 1 and 9 can be performed in $n/2+o(n)$ communication steps collectively. To perform 2-2 sorting on all VSM blocks for Phase 2 or VSM block-pairs for Phases 6 and 7 in parallel, each VSM block or block-pair can be sorted by emulating shearsort on it. These phases require $O(n^{2/3}\log n)$ time based on CET Row Sort. Phases 3 and 5 (row-to-column mapping) consist of some routing steps and can each be performed using $n/2+o(n)$ communication steps in a fault-free mesh. Since the width of the virtual submesh is $n/2+o(n)$ hops, Phases 3 and 5 can be emulated in the virtual submesh using SET with a slowdown factor of $1+o(1)$, for a total of $n+o(n)$ communication steps. Phase 4 can be performed by using the robust sorting algorithm proposed in [11, 13], which requires $n+o(n)$ communication steps and $2n+o(n)$ comparison steps.

Note that we can select m_1 and m_2 so that a VSM block has $n^{4/3}/2+O(n^{2/3})$ VSM nodes. Therefore, upon the completion of Phase 7, there are at most $O(n^{4/3})$ dummy elements "∞" located at the second layer of the first $O(1)$ VSM blocks. In order for Phase 9 (inverse of algorithm DR) to work, we have to move the data items in sorted order to the processors that originally had data items (upon the completion of Phase 1). We first compute the number of dummy elements at each VSM block after Phase 1 of the algorithm, and then compute the position for Phase 9 for each sorted item. This can be done in a precalculation step using semigroup and prefix computations on the VSM, which require $O(n)$ time. Since these data items are at most $O(n^{2/3})$ hops away from their positions for Phase 9, Phase 8 can be easily performed using $O(n^{2/3})$ communication steps.

The execution time for FOE-snake sort is dominated by Phases 1, 3, 4, 5, and 9, which collectively require $2.5n + o(n)$ communication steps and $2n + o(n)$ comparison steps □

Surprisingly, this running time has exactly the same leading constant as the fastest 1-1 sorting algorithm reported in the literature for $n \times n$ fault-free meshes [6, 7]. It is the best result reported thus far for sorting on faulty meshes for any ranking order.

We can generalize algorithm FOE-snake sort for meshes with f faults. The time required for sorting on an $n \times n$ 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 for worst-case fault patterns. We can also extend the sorting algorithm to robust h-h sorting with $h > 1$, by emulating a-a sorting on an appropriate VSM, where a is equal to $h+1$ when f is not very large.

4. The Mesh Robustness Theorem

A variant of the RACE scheme, where data items are input/output to/from VSM nodes, can be applied to various other problems to derive efficient robust algorithms, as indicated by the following theorem.

Theorem 4.1 (Mesh Robustness Theorem)
If an algorithm for a mesh performs S consecutive routing and computation steps along the same dimension on the average, and there exists an $m_1 \times m_2 \times \cdots \times m_d$ virtual submesh whose width overhead is $o(S)$, then the slowdown factor for performing the algorithm on a virtual submesh of the faulty mesh is $1 + o(1)$ relative to a fault-free $m_1 \times m_2 \times \cdots \times m_d$ mesh.

When there are $f = o(n)$ faults, it is guaranteed that a virtual submesh with comparable size and width overhead $o(n)$ exists, leading to the following corollary.

Corollary 4.2 *If an algorithm for a mesh performs S consecutive routing and computation steps along the same dimension on the average, and there are f faulty processors in an $n_1 \times n_2 \times \cdots n_d$ mesh, $f = o(S)$ and $f = o(n_{min})$, then the slowdown factor for performing the algorithm on a virtual submesh of the faulty mesh is $1 + o(1)$ relative to a fault-free $(n_1 - o(n_{min})) \times (n_2 - o(n_{min})) \times \cdots (n_d - o(n_{min}))$ mesh, where $n_{min} = \min(n_1, n_2, ..., n_d)$.*

A wide variety of algorithms have $S = O(n)$, and thus can run with a slowdown factor $1 + o(1)$ when the number of fault is $o(n)$. This mesh robustness theorem can be applied to N-node meshes with $o(N)$ random faults (or for p-faulty meshes [5] with $p = o(1)$) with high probability. A congestion-free virtual subarray (virtual submesh or virtual subtorus) is a virtual subarray embedded in a faulty array with congestion 1. It is guaranteed that an $(n_1 - o(n_{min})) \times (n_2 - o(n_{min})) \times \cdots (n_d - o(n_{min}))$ congestion-free virtual subarray with width overhead $o(n_{min})$ exists in an $n_1 \times n_2 \times \cdots \times n_d$ array with $o(n_{min})$ faults. Many important communication algorithms can be executed on congestion-free virtual subarrays with a factor of $1 + o(1)$ slowdown relative to a fault-free array. The details will be reported in the near future.

Conventional wisdom is that low-degree networks are less robust than high-degree networks. But our results indicate that low-dimensional meshes and tori are very robust in that an array with a large number of faulty processors and links has, for a large variety of problems, computation and communication powers similar to those of a fault-free array. For example, an N-node 2-D mesh with $N^{1/3}$ faults can execute many algorithms almost as fast as a slightly smaller fault-free mesh. Dally [4] and Agarwal [1] have shown that lower-dimensional networks achieve better performance than high-dimensional networks under various constraints, such as constant bisection bandwidth, fixed channel width, and fixed node size. Our robustness results for meshes and tori, combined with their previously established cost/performance benefits [1, 4], make the case for low-dimensional architectures even stronger.

5. Conclusion

In this paper, we have proposed the RACE scheme for designing efficient robust algorithms, and the fastest algorithms for sorting on faulty meshes. We showed that sorting on an $n \times n$ mesh that has $o(\sqrt{n})$ faulty processors can be performed in $2.5n + o(n)$ communication steps and $2n + o(n)$ comparison steps, which has the same leading constant as the best sorting algorithm for fault-free meshes. We also formulated the mesh robustness theorem, which extends the techniques and algorithms used in this paper to a variety of other important problems to obtain low-overhead robust algorithms for faulty meshes.

References

[1] Agarwal, A., "Limits on interconnection network performance," *IEEE Trans. Parallel Distrib. Sys.*, Vol. 2, no. 4, Oct. 1991, pp. 398-412.

[2] Bruck, J., R. Cypher, and C. Ho, "Fault-tolerant meshes and hypercubes with minimal numbers of spares," *IEEE Trans. Comput.*, vol. 42, no. 9, Sep. 1993, pp. 1089-1104.

[3] 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.

[4] Dally, W.J., "Performance analysis of k-ary n-cube interconnection networks," *IEEE Trans. Comput.*, Vol. 39, no. 6, Jun. 1990, pp. 775-785.

[5] 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.

[6] 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.

[7] 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.

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

[9] Parhami, B. and C.-H. Yeh, "The robust-algorithm approach to fault tolerance on processor arrays: fault models, fault diameter, and basic algorithms," *Proc. First Merged International Parallel Processing Symposium and Symp. Parallel and Distributed Processing*, Apr. 1998, pp. 742-746.

[10] Parhami, B., *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum Press, 1999.

[11] Yeh, C.-H. and B. Parhami, "Optimal sorting algorithms on incomplete meshes with arbitrary fault patterns," *Proc. Int'l Conf. Parallel Processing*, Aug. 1997, pp. 4-11.

[12] Yeh, C.-H., "Efficient low-degree interconnection networks for parallel processing: topologies, algorithms, VLSI layouts, and fault tolerance," Ph.D. dissertation, Dept. Electrical & Computer Engineering, Univ. of California, Santa Barbara, Mar. 1998.

[13] Yeh, C.-H. and B. Parhami, "Efficient sorting algorithms on incomplete meshes," *J. Parallel Distrib. Comput.*, to appear.