ART: Robustness of Meshes and Tori for Parallel and Distributed Computation

Chi-Hsiang Yeh
Dept. of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, K7L 3N6, Canada
yeh@ee.queensu.ca

Behrooz Parhami
Dept. of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106, USA
parhami@ece.ucsb.edu

Abstract

In this paper, we formulate the array robustness theorems (ARTs) for efficient computation and communication on faulty arrays. No hardware redundancy is required and no assumption is made about the availability of a complete submesh or subtorus. Based on ARTs, a very wide variety of problems, including sorting, FFT, total exchange, permutation, and some matrix operations, can be solved with a slowdown factor of $1 + o(1)$. The number of faults tolerated by ARTs ranges from $o(\min(n^{1-\frac{1}{d}}, \frac{n}{d}, \frac{n}{h}))$ for n-ary d-cubes with worst-case faults to as large as $o(N)$ for most N-node 2-D meshes or tori with random faults, where h is the number of data items per processor. The resultant running times are the best results reported thus far for solving many problems on faulty arrays. Based on ARTs and several other components such as robust libraries, the priority emulation discipline, and $X'Y'$ routing, we introduce the robust adaptation interface layer (RAIL) as a middleware between ordinary algorithms/programs (that are originally developed for fault-free arrays) and the faulty network/hardware. In effect, RAIL provides a virtual fault-free network to higher layers, while ordinary algorithms/programs are transformed through RAIL into corresponding robust algorithms/programs that can run on faulty networks.

1. Introduction

A d-dimensional mesh consists of $n_1 n_2 \cdots n_d$ nodes 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. The scalability, compact layout, constant/small node-degree, desirable algorithmic properties, and many other advantages have made meshes, tori, and n-ary d-cubes the most popular topologies for the interconnection of parallel processors. A very large variety of algorithms have been proposed for these networks [3, 10, 11, 13, 16, 15, 17]. These algorithms usually assume that a fault-free mesh or torus is available, and most of them cannot be applied to faulty meshes or tori directly, even in the presence of only a small number of faulty elements.

To utilize in faulty arrays the very large body of ordinary algorithms originally developed for fault-free arrays without redesigning or modifying them one by one, we add the *robust adaptation interface layer (RAIL)* as a middleware between ordinary algorithms/programs and the faulty hardware/network (i.e., faulty meshes, tori, or n-ary d-cubes). In RAIL, no hardware redundancy (e.g., spare processors or links) is necessary and the availability of a complete (fault-free) submesh or subtorus is not required. RAIL can be envisioned as an MPI-like middleware, where the details for tolerating faults are transparent to applications and hidden from the programmers. As a result, from the algorithm/program point of view, the adaptation layer located on top of the hardware layer hides the faulty processors and/or links from the algorithms, so that a virtual fault-free network is provided to higher layers and ordinary algorithms/programs can be executed on such a platform without modifications; from the hardware/network point of view, the adaptation layer in effect incorporates fault tolerance into the design of algorithms so that ordinary algorithms/programs are transformed into robust algorithms/programs that can run on faulty networks. In addition to using RAIL as a middleware, the proposed techniques can also lead to robust algorithms/functions that may be implemented in a library, which can be invoked by programmers and/or the operating system so that the applications have the flexibility to efficiently utilize the fault tolerance features.

Based on RAIL, we formulate the *array robustness theorems (ARTs)*, which show that a wide variety of important problems, such as sorting, permutation routing, unicast, broadcast, total exchange, reduction (e.g., semigroup computation), prefix computation, selection, fast Fourier transform (FFT), matrix multiplication, and ascend/descend algorithms can be executed on faulty arrays (i.e., meshes, tori, or n-ary d-cubes) with a factor of $1 + o(1)$ slowdown relative to a fault-free array of the same type. ARTs classify computing and communication problems into categorizes that can be efficiently supported by RAIL, associate them with the required RAIL techniques and procedures systematically, and quantify the resultant performance. 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 proces-

sors 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, such as sorting, permutation routing, reduction (e.g., semigroup computation), prefix computation, FFT, matrix multiplication, unicast, broadcast, total-exchange, multinode broadcast, random routing, and dynamic broadcast, almost as fast as a slightly smaller fault-free mesh. Dally [8] 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, tori, and n-ary d-cubes, combined with their previously established cost/performance benefits [1, 8], make the case for low-dimensional architectures even stronger.

2. The robust adaptation interface layer

In this section we present several components for the robust adaptation interface layer (RAIL).

2.1. Basic components of RAIL

Various algorithms have been developed for mesh-connected computers and their variants, such as tori and n-ary d-cubes, based on the assumption that a fault-free mesh (or torus, n-ary d-cube) is available [3, 10, 11, 13, 16, 15, 17]. Since fault tolerance is very important to parallel processing, a variety of techniques for adaptive fault-tolerant routing or reconfiguring faulty arrays have also been proposed [5, 6, 12, 18, 19, 23]. In particular, we propose in [23] $X'Y'$ routing for deadlock-free wormhole routing faulty arrays. Combination of ordinary algorithms and appropriate adaptive or fault-tolerant routing schemes constitutes the first component for RAIL. However, such combined algorithms in general do not guarantee optimal performance, and the resultant performance is highly dependent on the fault patterns, which may lead to very poor performance in the worst case (e.g., degraded by a large nonconstant factor); while the reconfiguration techniques either use redundant links and nodes to restore a complete mesh or require a complete submesh/subtorus for performing parallel algorithms, which may be expensive to build or unavailable.

Since it is impractical to redesign all algorithms for faulty meshes and tori one by one and guaranteed optimal or satisfactory performance is important for some applications or computing environments, several systematic methods for transforming ordinary algorithms to obtain robust algorithms that can run on faulty arrays have been proposed [7, 14, 23]. One general fault tolerance scheme along this line, called *the multi-scale self-simulation scheme*, was devised by Cole, Maggs, and Sitaraman [7], who showed 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 $\varepsilon > 0$. Due to the robustness of the scheme, we incorporate it into RAIL as another component.

However, the multi-scale self-simulation scheme is relatively complicated, leading to difficult implementation issues, and comes with huge performance penalties in practice, given the significant increase in the leading constants of the running times. In [23], we proposed the *robust algorithm-configured emulation (RACE) scheme* as a software-based fault tolerance scheme for the systematic transformation of ordinary algorithms to obtain corresponding robust algorithms that are fast and easy to implement. In [23], RACE was used to incorporate fault tolerance into the design of algorithms, and is applicable to many computation and communication problems to obtain robust algorithms that have slowdown factors $1 + o(1)$ relative to the best algorithms for fault-free arrays. As a comparison, the multi-scale self-simulation scheme [7] is applicable to almost any problems, while RACE is only applicable to a (large) subset of them. However, RACE is also a general emulation scheme and is applicable to most important computation and communication problems in parallel computation. Moreover, when RACE is applicable to a certain application, it is usually preferable to use RACE instead of the multi-scale self-simulation scheme, since the former usually achieves considerably better performance (by a moderate to large constant factor) and is easier to implement. In this paper, we propose to use RACE to generate important robust algorithms that can be used as library routines. Such robust lobraries constitute another component of RAIL. Note that algorithms in robust lobraries can be further optimized. They can also be derived/transformed using techniques other than RACE or redesigned from scratch.

In addition to generating robust algorithms, we propose in this paper to use RACE in a way different from [23], by implementing it as a robust middleware between ordinary algorithms/programs and the faulty network/hardware. A difference between RAIL and RACE alone is that RAIL is more versatile and is applicable to any computation or communication problems. In Subsection 2.2 we define in the following subsections the notion of virtual subarrays (VSA) in faulty arrays (with or without wraparound) that can utilize the RAIL middleware for efficient computing and communication on faulty arrays without relying on hardware redundancy. Based on VSA, RACE, and several related techniques, we formulate, the array robustness theorems (ARTs) for solving a wide variety of problems on faulty arrays with negligible overhead. RACE and the associated ARTs become an important component of RAIL. More details can be found in Subsection 2.3 and [23] for RACE and Sections 3, 4, and 5 for ARTs.

The *priority emulation discipline* is a heuristic strategy that can be used in priority-based schemes for emulating ordinary algorithms with low cost and improved performance (or performance that can be guaranteed probabilisticly). It is another component of RAIL and will be introduced in Subsection 2.4. Other techniques can also be incorporated into RAIL at a later time and will be reported in the future.

Figure 1 illustrates the relationship between RAIL, robust algorithms, hardware-based fault tolerance layer, and their interfacing with algorithm/program and network/hardware layers. VSA denotes virtual subarrays to be defined in Subsection 2.2. SET and CET are the *stepwise emulation technique* and the *compaction/expansion technique* to be introduced in Subsections 2.3 and 4, respectively, for performing RACE. Algorithms/programs that conform to ar-
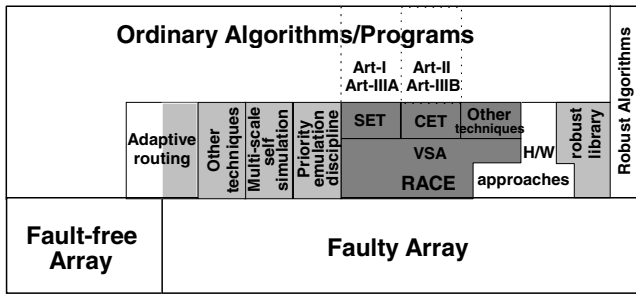
Figure 1. The robust adaptation interface layer (RAIL), associated components, and its interface with related layers. RAIL is represented as shaded areas (including dark and light ones). The dark area is representing the RACE scheme, a component of RAIL.

Figure 2. (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.

ray robustness theorem I (ART-I) or condition A of ART-III can use SET; while algorithms/programs that conform to ART-II or condition B of ART-III can use CET. More details concerning how the various components of RAIL should be used will be presented in Subsection 2.5.

Note that the execution of ordinary algorithms on RAIL usually has negligible degradation compared to fault-free systems (e.g., a factor of $1 + o(1)$ slowdown), and are relatively easy to implement after reconfiguration is performed, which only needs to be done once after a fault occurs or is recovered from. Note also that when the number of faults is small, reconfiguration for RAIL can be performed in a short time, and possible configurations may be precomputed and stored, if so desired, in a distributed manner and broadcast when needed. Moreover, as indicated in Fig. 1, RAIL can also work in combination with previous hardware-based fault tolerance schemes. This can be done by executing ordinary algorithms when the number of faults has not exceeded the limit of the hardware fault tolerance scheme, while activating an appropriate component of RAIL otherwise. In effect, RAIL transform any ordinary algorithms to robust ones, including a very wide variety of important problems at high speed, and converting the faulty array into a virtual fault-free array, even for faulty meshes and tori whose rows and columns are (almost) all incomplete and those without any complete submesh or subtorus.

2.2. Virtual subarrays (VSAs) as a basis for RAIL

A *virtual subarray (VSA)* of a d-dimensional faulty array (with or without wraparound) is obtained by embedding a smaller d-dimensional array 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 (See Fig. 2a). More precisely, each node of this smaller array is mapped onto a different healthy node of the faulty array; each link of this smaller array is mapped onto a healthy path of the faulty array. The embedded rows (or columns) of a certain dimension i, $i = 1, 2, ..., d$, do not overlap with each other, and are called *dimension-i virtual rows* (or *vir-*
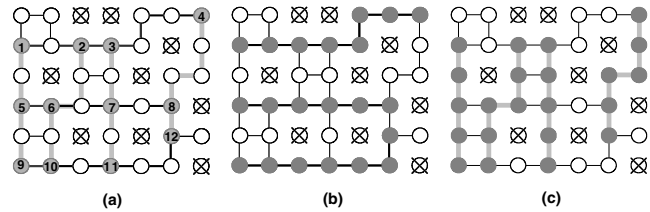
tual columns) of the virtual subarray. Figures 2b and 2c show the virtual rows and virtual columns of a 3×4 virtual subarray.

Node $(x_1, x_2, ..., x_d)$ of the virtual subarray (called a *VSA 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_d of dimension d. Note that the 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., minimizing the dilation of the resultant embedding). Then, VSA nodes $(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_d)$ for certain x_j, $j \neq i$, and all $x_i = 1, 2, ..., m_i$ form a *dimension-i row* of the virtual subarray that has length equal to m_i and is called a *VSA row*. The nodes of a VSA row form a subset of the corresponding dimension-i virtual row. Figure 2a shows a 3×4 virtual subarray in an array with 9 faults and the associated VSA nodes.

A *virtual submesh (VSM)* is a virtual subarray without wraparound; a *virtual subtorus (VST)* is a virtual subarray with wraparound. A *congestion-free virtual subarray* is a virtual subarray embedded in a faulty array with load and congestion both equal to 1 [11]. All the embedded links of a congestion-free virtual subarray correspond to a set of nonoverlapping paths in the faulty array. In other words, a dimension-i virtual row intersects with a dimension-j virtual row at exactly one node if $i \neq j$, while a virtual row does not intersect with another virtual row of the same dimension i. Many important problems can be solved efficiently on congestion-free virtual subarrays. More details concerning VSAs can be found in [23].

2.3. The RACE scheme as a component of RAIL

In RACE, we assume that a preprocessing stage has identified a virtual subarray to be used (perhaps at reconfiguration time using the simple method presented in Subsection 2.2 or a more complicated method). We can then redistribute the data on a faulty network to a virtual subgraph and then uses the virtual subgraph to emulate algorithms developed for a fault-free network. The proposed RACE scheme have three basic stages:

Basic Stages for Robust Emulation

- Stage 1: The data items to be processed are redistributed evenly to the processors on the virtual subarray such that a VSA processor has at most a items. On the virtual subarray, a processor that has fewer than a items may pad its list with suitable "dummy item(s)" (e.g., ∞ for sorting).

- Stage 2: The virtual subarray emulates a corresponding algorithm on an $m_1 \times m_2 \times \cdots \times m_d$ array, each processor of which has at most a items.

- Stage 3: The results are redistributed back to healthy processors of the original $n_1 \times n_2 \times \cdots \times n_d$ faulty array.

More details concerning RACE can be found in [23].

Stage 2 of the RACE scheme can be implemented using various techniques such as SET or CET. In the *stepwise emulation technique (SET)* we directly emulate 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. If the virtual subarray is a complete array, (that is, no faulty processor exists within it and the embedded smaller array has dilation 1), no degradation is caused by Stage 2 using this naive method. On the other hand, under wormhole or cut-through routing, or packet-switching with a large load, the overhead caused by Stage 2, when implemented with SET, is negligible even when the array is faulty. Also, many algorithms, such as semigroup and prefix computations [14], can be emulated using SET with small overhead even if packet-switching is employed and the load is 1. Moreover, SET will be used is used to formulate the array robustness theorem I (ART-I) for fault-tolerant computing on congestion-free virtual subarrays in Section 3.

We can also use other techniques to implement Stage 2 of the RACE scheme. For example, the compaction/expansion technique leads to the array robustness theorem II (ART-II), presented in Section 4. Moreover, efficient implementation of Stages 1 and 3 extends ARTs I and II to the array robustness theorem III (ART-III), as presented in Section 5. A wide variety of important problems can then be solved efficiently in faulty arrays based on these ARTs.

2.4. The priority emulation discipline for RAIL

In this subsection we propose a heuristic strategy, called the *priority emulation discipline*, for effective implementation of robust algorithms based on RAIL in case where computation-free routing (i.e., routing without any computation steps) is performed or when an average of S or more routing steps separate consecutive computation steps. Our goal is to perform these routing tasks or algorithms on a virtual subarray of the faulty array with small slowdown relative to a fault-free $m_1 \times m_2 \times \cdots \times m_d$ array of the same type, when there exists a congestion-free $m_1 \times m_2 \times \cdots \times m_d$ virtual subarray with small width overhead (e.g., $o(S/b)$) and the routing path for any packet makes at most a small number b of turns (i.e., a routing path changes its dimensions at most b times). The discipline is also applicable to configurations different from virtual subarrays in RACE, but no guaranteed results are described here for such applications.

We first look at several problems that may occur when using naive methods to implement such algorithms on a virtual subarray and then propose strategies to improve the performance. The first problem is that when a packet X is delayed by $o(S)$ steps when it arrives at node V (for example, due to dilation along its routing path), there can be up to $o(dS)$ other packets at node V which are routed after packet X in the original algorithm for fault-free array but arrive at node V before packet X in the faulty array. If the network nodes of the faulty array use a first-come first-serve (FCFS) discipline, then packet X will be delayed by $o(dS)$ steps at node V and the resultant algorithm is suboptimal when d is not a constant. What makes the situation worse is that packet X may then be delayed by $o(d^2S)$ steps at the next node, $o(d^3S)$ steps at the following node, and so on. Since some packets may be severely delayed, the performance of the algorithms may degrade considerably when using such a naive method. One may argue that the preceding problem will not occur if the packets of the emulated algorithm are scheduled to arrive at a node right before they can be transmitted. However, this will give rise to accumulated delays of different types: when a packet X is delayed by $o(S)$ steps when it arrives at node V, there can be up to $o(S)$ packets at node V which are routed after packet X in the original algorithm for fault-free array but arrive at node V before packet X in the faulty array. Therefore, packet X will be delayed by $o(S)$ steps at node V, and may be delayed by another $o(S)$ steps at the next node, and so on.

In what follows we present a strategy, called the *priority emulation discipline*, which is easy to implement and can solve the above problems. The central idea of this strategy is that if a packet X_1 should be transmitted before another packet X_2 at a node (and over a link) according to the schedule of the algorithm on the fault-free array being emulated, then packet X_1 has higher priority for transmission over that link in the faulty array. Moreover, when there is at least one packet requiring transmission over a link, that link will not be idle so that the network resources are effectively utilized. An intuitive reason that the priority emulation discipline can improve the performance is that the more a packet is delayed, the higher its priority becomes in its future transmissions so that it is usually not further delayed at (temporarily) congested nodes/links. Therefore, the delay of a packet does not accumulate along its routing path and its slowdown is primarily due to the dilated paths between VSA nodes along its routing path.

Implementation of the priority emulation discipline is simple: all packets are still routed along the same paths as those in the emulated algorithm, and each packet begins its routing at the same time as in the emulated algorithm (though it may be queued at the starting node). We first consider the case where in the emulated algorithm packets arrive at a node right before their transmissions (without queueing). Each packet X begins with a tag that holds the starting timestamp T_X. Whenever packet X is transmitted through a VSA node, the timestamp T_X is increased by one. When there are multiple packets contending a node (or a link under the multi-port communication model), a packet Y that has the smallest timestamp T_Y is transmitted, since T_Y is the time that packet Y should have been transmitted over that node or link if the array were fault-free. Note that

there cannot be multiple packets with the same tag at a node (and over a certain link under the multi-port communication model), since T_Y is the time for packet Y to be transmitted over that link in the emulated algorithm and there can be only one packet transmitted over that link at the same time in the emulated fault-free array. In a more general case where packets of the emulated algorithm may arrive at a node before their transmissions and are queued there, we simply set T_X to be the time for packet X to be transmitted at the corresponding node in the emulated fault-free array. Note that the virtual subarray is congestion free, so once we resolve the order for transmissions at a VSA node, packets never compete for links again along the path between two VSA nodes (i.e., along an embedded link).

2.5. Typical operations of RAIL

RAIL is a layer inserted between ordinary algorithms/ programs and the faulty hardware/network (i.e., the faulty array). We can envision RAIL as an MPI-like middleware, where the details for tolerating faults are transparent to the program developers, similar to the way the details for implementing collective communications are hidden from parallel programmers using MPI. In addition to ease of programming, such characteristics also improve the portability of parallel programs across platforms.

RAIL is located on top of the layer corresponding to a faulty array, and below the layer corresponding to ordinary algorithms and programs designed for fault-free arrays. RAIL hides the faulty elements from the algorithms and programs, so that a virtual fault-free network is provided to the algorithm/program layer; RAIL also transforms ordinary algorithms and programs into robust algorithms/programs that can run on the faulty array. Figure 1 illustrates the relationship between RAIL and the algorithm/program and network/hardware layers.

As presented in Subsection 2.1, RAIL consists of various components, including the RACE scheme presented in Subsection 2.3, the priority emulation discipline proposed in Subsection 2.4, as well as other fault tolerance schemes such as adaptive fault-tolerant routing [18, 23] and the multi-scale self-simulation scheme proposed by Cole, Maggs, and Sitaraman [7]. When a new fault occurs, RAIL first checks whether there are spare processors and/or links that can tolerate the fault and resume a complete array. If not, RAIL may estimate and/or measure the performance by using adaptive fault-tolerant routing and/or the priority emulation discipline to get around the fault, before the reconfiguration phase of RACE is completed. If the expected or measured performance is poor, RAIL will switch to RACE, which guarantees an optimal slowdown (within a factor of $1 + o(1)$) for most important computation and communication problems when the total number of faults are not very large. RAIL will select the appropriate techniques, such as SET or CET, to perform the required emulation, according to the switching technique used, the number of items per node, and the categories for the problems being executed. In the unusual case where RACE is not applicable to the problems or perform poorly, RAIL will consider switching to other schemes such as the multi-scale self-simulation

scheme that is applicable to most problems. RAIL can, of course, test these options in a different order according to the specific characteristics of the applications and the parallel system, as well as the information available. We expect that more fault tolerance schemes will be developed and added to RAIL due to the importance of fault tolerance in future parallel and distributed systems. We will report further developments in the future.

3. Fault-tolerant communication and computation on congestion-free virtual subarrays

Recall that a congestion-free virtual subarray is a virtual subarray embedded in a faulty array with congestion 1. In this section, we show that many important problems can be solved efficiently on congestion-free virtual subarrays based on RAIL.

3.1 ART-I for congestion-free virtual subarrays

In this subsection, we focus on a specific class of algorithms, which perform an average of S consecutive routing steps along each of the dimensions without any computation step in between. We propose the *phase synchronization discipline* and show that these algorithms can be emulated on congestion-free virtual subarrays with negligible slowdown under this discipline using the RACE scheme.

When using the phase synchronization discipline, all nodes simply synchronize at the end of a phase (which may perform certain computation steps). If there exists a congestion-free $m_1 \times m_2 \times \cdots \times m_d$ virtual subarray whose width overhead is $o(S)$, then the average slowdown factor for each phase is $1 + o(1)$ relative to a fault-free $m_1 \times m_2 \times \cdots \times m_d$ array of the same type. This is because routing along a dimension of a congestion-free virtual subarray is only delayed by $o(S)$ steps additively. Moreover, if we synchronize each phase, the interaction between delayed packets will never cause excessive accumulation of delay on certain packets, since the delay of one phase has no effect on the next phase. The RACE scheme under the phase synchronization discipline is very easy to implement and is powerful in that many important robust communication and computation algorithms can be performed with a factor of $1 + o(1)$ slowdown, as indicated in the following theorem and corollaries.

Theorem 3.1 (Array Robustness Theorem I (ART-I)) *If an algorithm for an array (i.e., mesh or torus) performs an average of S consecutive routing steps along each of the dimensions (at the same time) without any computation step in between, and there exists a congestion-free $m_1 \times m_2 \times \cdots \times m_d$ virtual subarray whose width overhead is $o(S)$, then the slowdown factor for performing the algorithm on a virtual subarray of the faulty array is $1 + o(1)$ relative to a fault-free $m_1 \times m_2 \times \cdots \times m_d$ array of the same type.*

When the number of faulty processors and/or links in an $n_1 \times n_2 \times \cdots \times n_d$ is $o(n_{min})$, where $n_{min} = \min(n_1, n_2, ..., n_d)$, it is guaranteed

that at least 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 and is easy to find, leading to the following corollary.

Corollary 3.2 *If an algorithm for an array (i.e., mesh or torus) performs an average of S consecutive routing steps along each of the dimensions (at the same time) without any computation step in between, and there are $o(\min(S, n_{min}))$ faulty processors in an $n_1 \times n_2 \times \cdots n_d$ array, then the slow-down factor for performing the algorithm on a virtual subarray of the faulty array is $1 + o(1)$ relative to a fault-free $(n_1 - o(n_{min})) \times (n_2 - o(n_{min})) \times \cdots (n_d - o(n_{min}))$ array of the same type.*

Note that the number of faulty processors and links whose tolerance is indicated by ART-I and Corollary 3.2 is not small for low-dimensional arrays (i.e., small d). For example, these results apply to an N-node 2-D mesh/torus with $\Theta(\sqrt{N}/\log N)$ faults for which the size of the virtual subarray is $1 - o(1)$ that of the entire array. Communication algorithms, such as unicast (node-to-node routing), broadcast, total-exchange, and multinode broadcast, are at the heart of many applications [3]. Based on ART-I and Corollary 3.2, we can show that these algorithms as well as a variety of other important algorithms can be executed on congestion-free virtual subarrays with a factor of $1 + o(1)$ slowdown relative to a fault-free array. Since optimal algorithms for total exchange are among the most complicated communication algorithms, we describe, in the following subsection, an asymptotically optimal algorithm for performing total-exchange in faulty n-ary d-cubes as an example of applications of ART-I.

Various other computation or communication problems can also be executed efficiently on top of RAIL over a (possibly) faulty array. In particular, many applications, such as FFT, bitonic sort, matrix multiplication, convolution, and permutation can be formulated as ascend/descend algorithms [11] and performed efficiently based on ART-I [23]. Execution of these and other algorithms based on ART-I will be reported in the near future.

3.2. Total exchange on faulty *n*-ary *d*-cubes

In what follows we propose a generally applicable algorithm for executing g total exchange tasks in any fault-free vertex- and edge-symmetric network, where g is the degree of the network. We show that $(N-1)D_{ave}$ communication time is sufficient in an N-node network and is strictly optimal, where D_{ave} is the average internode distance of the network. We label the nodes of the network as $0, 1, 2, ..., N-1$ and let dimensions $r_{i,1}, r_{i,2}, ..., r_{i,q_i}$ be the dimensions of links in the order encountered along a shortest path between node 0 and node i, $i = 1, 2, 3, ..., N-1$. We first present an algorithm $\mathcal{A}_1$ for executing a TE task under the *single-dimension communication (SDC) model* [21], where only links of the same dimension can be used at the same time for the entire network. At stage i of the algorithm $\mathcal{A}_1$, $i = 1, 2, 3, ..., N-1$, each node successively sends a packet through links of dimensions $r_{i,1}, r_{i,2}, ..., r_{i,q_i}$. Since the network is vertex-symmetric, all the N packets sent by differ-

ent nodes during stage i have different destinations (which is equivalent to a permutation task), and a node sends packets to different destinations during different stages (which is equivalent to a single-node scatter task [3]). Collectively, the network performs a TE task during the $N-1$ stages, under the SDC model. Since all the packets are sent via shortest paths, the total time required is equal to $(N-1)D_{ave}$.

To optimally perform g TE tasks under the all-port communication model, we will execute all of them at the same time, each under the SDC model. However, contention over links will occur if we directly apply algorithm $\mathcal{A}_1$ for all the g tasks. In what follows, we show how contention can be completely avoided. We define algorithm $\mathcal{A}_i$, $i = 2, 3, 4, ..., g$, as the algorithm where packets are sent over links of dimension $((j + i - 2) \bmod d) + 1$ during a step if, at the same step of algorithm $\mathcal{A}_1$, packets are sent through links of dimension j, $j \in \{1, 2, 3, ..., g\}$ (along the same direction in networks like n-ary d-cubes). Clearly, algorithms $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, ... \mathcal{A}_g$ are guaranteed to use links of different dimensions at any given step, and all of them can be simultaneously completed in $(N-1)D_{ave}$ time.

Since the network is vertex and edge symmetric, the N packets sent during stage i of a certain algorithm $\mathcal{A}_j$ have different destinations, and a node sends packets to different destinations during different stages. As a result, each algorithm $\mathcal{A}_j$, $j = 1, 2, 3, ... g$, performs a TE during the $N-1$ stages. Therefore, we can perform g TE tasks in $(N-1)D_{ave}$ time under the all-port communication model.

If we allow packets to be split into mini-packets that can be routed independently without any overhead, then the time required to perform a single TE task becomes $(N-1)D_{ave}/g$. Many papers have proposed algorithms for executing TE tasks in several popular topologies under several communication models. The result given above applies to general symmetric topologies, and gives optimal execution times for several important topologies, including hypercubes, n-ary d-cubes, generalized hypercubes [4], and star graphs [2]. The preceding algorithm for the all-port communication model is essentially based on finding an algorithm for executing total exchange under the single-dimension communication model and then rotating the dimensions of the single-dimension algorithm by i dimensions for all $i = 1, 2, ..., 2d-1$ in order to find an algorithm that fully utilizes all dimensions concurrently. This strategy is useful in many routing problems and will be referred to as the *Single-To-All Rotation (STAR)* technique in this paper. The resultant algorithm for performing TE is called a *STAR TE algorithm*.

To apply the STAR TE algorithm to a faulty n-ary d-cube, we first identify a congestion-free virtual m-ary d-subcube in it, which is a virtual subtorus (VST) with the same length m along each VST row. From the notion provided by ART-I, we know that the number of turns in a routing path should be minimized in order to improve the performance and increase the number of faults tolerated without affecting the leading constant of the running time. Therefore, we use a special case of the STAR TE algorithm where links of the same dimension in a shortest path must be a connected subgraph of that path (i.e., they form a unique VSA subrow for that dimension). The resultant path between each of the source-destination pairs has at most $d-1$ turns and, thus, the

length of a path can be increased by at most $dW_{overhead}$ hops, where $W_{overhead}$ is the width overhead. From ART-I, when $W_{overhead} = o(D_{ave}/d) = o(m)$, the slowdown factor for executing $g = 2d$ TE tasks on the virtual m-ary d-subcube is only $1 + o(1)$ compared to a fault-free m-ary d-cube, leading to the following corollary.

Corollary 3.3 *We can execute $2d$ instances of a TE task in an n-ary d-cube that has an arbitrary pattern of $o(n/d)$ faulty processors and/or links in $(N-1)D_{ave} = dnN/4 + o(dnN)$ communication time, which is optimal within a factor of $1 + o(1)$, where D_{ave} is the average distance of a fault-free $(n - o(n/d))$-ary d-cube, $N = n^d - o(n^d)$ is the size of the virtual $(n - o(n/d))$-ary d-subcube, and data items are input/output to/from the virtual $(n - o(n/d))$-ary d-subcube.*

4. Fault-tolerant computation on general virtual subarrays

In this section, we introduce the *compaction/expansion technique (CET)*, derive the array robustness theorem II (ART-II) based on CET, and then present its applications to robust sorting and permutation routing.

4.1. ART-II for general virtual subarrays

Fault-tolerant computing in a virtual submesh using CET is based on the following 4 phases:

The Compaction/Expansion Technique (CET):

- 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 that are not VSM nodes 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: The consecutive operations (e.g., routing and computation) along rows of the dimension are performed within each compacted row (i.e., the virtual subrow composed of the m_i neighboring nodes currently holding the data).

- Phase 3 (expansion): The data 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 completed in $O(m_i + t_{max})$ time using algorithms for semigroup and prefix computation on a virtual row [14]. This precalculation phase only needs to be executed once after each new processor or link failure. Phases 1 and 3 can each be performed in $a\lceil t/2 \rceil$ time. The integer t is usually small, and is guaranteed to be smaller than f_B and the length of the virtual row. Clearly, Phase 2 can be performed
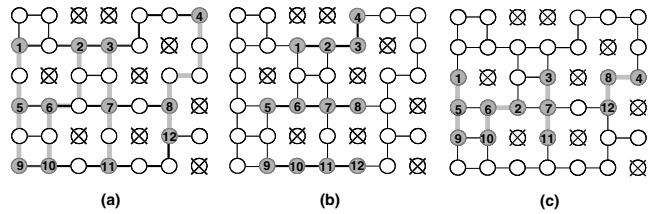

Figure 3. (a) A 3-by-4 VSM. (b) Compacted rows of the VSM. (c) Compacted columns of the VSM.

without any slowdown compared to operations along fault-free m_i-node rows.

Many important problems such as sorting require computation between every few routing steps and thus cannot be solved efficiently in faulty meshes using ART-I. Based on CET, we can solve this class of problems efficiently on a mesh which may contain significantly more faults. Figure 3 provides an example for sorting on compacted rows and columns of a virtual submesh based on CET. The shaded circles in Fig. 3a represent a 3×4 virtual submesh within a 6×7 mesh with 9 faulty processors. The shaded circles in Fig. 3b represent the positions of data items for performing CET row sort upon completion of Phase 1. The processors that hold the data items from a VSM row upon completion of Phase 1 are collectively called a *compacted row*. The number X in a circle represents the position for the data item that was initially held by processor X of the virtual submesh. The shaded circles in Fig. 3c represent the positions of data items for performing CET sort along columns. The processors that hold the data items from a VSM column form a *compacted column*.

The following theorem provides the conditions and characterizes the class of problems that can be solved efficiently based on CET.

Theorem 4.1 (Array Robustness Theorem II (ART-II))
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.

In addition to the different requirements for the underlying communication patterns, another important difference of ART-II compared to ART-I is that ART-II can potentially tolerate a significantly larger number of faults with negligible slowdown. For example, it can be shown that, using techniques similar to those used in [9], an N-node 2-D array with $o(N)$ random faults or a p-faulty array with faulty rate $p = o(1)$ contains $(N - o(N))$-node virtual subarrays with probability $1 - o(1)$; such an faulty array, however, is not likely to contain a congestion-free virtual subarray with comparable size, making ART-I inapplicable. (We conjecture that the preceding results can be extended to arrays of

higher dimensions $d \geq 3$.) When a congestion-free virtual subarray is available, we can execute d copies of such an algorithm concurrently, 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 $o(\min(S, n_{min}))$ faulty processors in an $n_1 \times n_2 \times \cdots n_d$ array, then the slowdown factor for performing (up to) d copies of the algorithm on a virtual subarray of the faulty array is $1 + o(1)$ relative to performing a copy of the algorithm on a fault-free $(n_1 - o(n_{min})) \times (n_2 - o(n_{min})) \times \cdots (n_d - o(n_{min}))$ array of the same type.*

A wide variety of algorithms, such as reduction (e.g., semigroup computation), prefix computation [15], FFT [11], sorting (e.g. the algorithms proposed in [10]), and matrix multiplication, have $S = O(n)$, and thus can run with a slowdown factor of $1 + o(1)$ in a mesh with an arbitrary pattern of $o(n)$ faults or in $100 - o(1)$ out of 100 N-node 2-D meshes with $o(N)$ random faults. In what follows, we present robust sorting in faulty meshes and n-ary d-cubes as examples of how ART-II can be applied.

4.2. Sorting in faulty meshes and *n*-ary *d*-cubes

In [22], we have shown that 1-1 sorting in an $n \times n$ mesh with $o(n)$ faults can be performed in $2.5n + o(n)$ communication time using the FOE-snake order [22], a variant of the blockwise snakelike order (see Fig. 4). The robust sorting algorithm proposed in [22] is actually a special case of ART-II since we emulate the sorting algorithm proposed in [10], which performs $\Theta(n)$ operations along a dimension on the average. In [14], we have shown that 1-1 sorting in an $n \times n$ bypass mesh with $o(n^{1/4})$ faults can be performed in $2.5n + o(n)$ communication time using row order or snakelike order by emulating the Schnorr/Shamir sorting algorithm [11, 13, 17]). The average number of steps along a dimension in the Schnorr/Shamir sorting algorithm is only $\Theta(n^{1/4})$, so the number of faults that can be tolerated without increasing the leading constant of the running time is only $o(n^{1/4})$ when direct emulation is used. In [20, 24], we have also shown that 1-1 sorting in an $n \times n$ mesh with $o(n)$ faults can be performed in $3n + o(n)$ communication time using row order or snakelike order by emulating a variant of the Schnorr/Shamir sorting algorithm [11, 13, 17]). We increase the number of faults tolerated to $o(n)$ by modifying the Schnorr/Shamir sorting algorithm so that the average number of steps along a dimension is increased to $o(n)$. All the aforementioned sorting algorithms can be viewed as the applications of ART-II to 1-1 sorting in 2-D faulty meshes.

Resorting to ART-II, we can easily extend the results given in these papers from 1-1 sorting to h-h sorting and from 2-D meshes to d-D faulty meshes and tori. These generalizations can be achieved by emulating the sorting algorithm for fault-free meshes and n-ary d-cubes given in [10], which performs $\Theta(n)$ steps along a dimension on the average. In summary, if an $m_1 \times m_2 \times \cdots \times m_d$ virtual submesh with width overhead $o(m_{min})$ exists, h-h sorting on the virtual submesh can be performed in $hm_1/2 + 2\lceil \frac{h}{4} \rceil (m_2 + m_3 + \cdots + m_d) + o(hdm_{min})$ communication steps (excluding precalculation time), where $d = O(1)$, $m_i = \Theta(m_j)$, and $m_{min} = \min(m_1, m_2, ..., m_d)$.

When $h = 1$, we can fold the data items to the middle of the virtual submesh and use 2-2 sorting of preceding method as the algorithms given in [10, 13, 20, 22, 24], leading to the following result: If an $m_1 \times m_2 \times \cdots \times m_d$ virtual submesh with width overhead $o(m_{min})$ exists, 1-1 sorting on the virtual submesh can be performed in $m_1/2 + 2(m_2 + m_3 + \cdots + m_d) + o(dm_{min})$ communication steps (excluding precalculation time), where $d = O(1)$ and $m_i = \Theta(m_j)$ for all i and j. The algorithm we proposed in [22] represents a special case of the preceding results with $d = 2$. The h-h sorting algorithm for fault-free n-ary d-cubes given in [10] requires $\lceil \frac{h}{4} \rceil (d - 1)n + o(hdn)$ communication steps that can be emulated using SET with a slowdown factor of $1 + o(1)$, in addition to $\lceil \frac{h}{4} \rceil n + o(hn)$ communication steps and $hn + o(hn)$ computation steps, which can be emulated using CET with a slowdown factor of $1 + o(1)$. These results also lead to efficient permutation routing and the details are omitted in this paper.

5. Fault-tolerant computing in the entire faulty array

Up to now, we have assumed that data items are input/output to/from a virtual subarray. In this section, we formulate the *array robustness theorem III (ART-III)* for fault-tolerant computing and communication on the entire faulty array (in contrast to computing on a virtual subarray, as required by ART-I and ART-II).

An important feature of RAIL, based on ART-III, is that all healthy and connected nodes, rather than only nodes belonging to a virtual subarray, can be used for computation. Thus, computing resources are not wasted unnecessarily, which is particularly important to computation-bound problems. As for communication-bound problems, utilizing all healthy nodes to execute a task does not necessarily increase the performance, since the bisection bandwidth is not increased and a certain amount of communication overhead is required to distribute data items to nodes that do not belong to the virtual subarray. Therefore, ART-I and ART-II, which only use nodes on a virtual subarray for computation in order to facilitate simpler and more efficient communications, may be more suitable for solving communication-bound problems. Note that even when ART-I and ART-II are applied to the execution of communication-intensive subtasks within an application, no healthy node is disabled; rather, all such nodes are kept alive and running at all times. A load balancing policy may improve the system performance by appropriately distributing the subtasks among all healthy nodes.

5.1. ART-III for the entire faulty array

If we map healthy and connected processors in a faulty array to a virtual subarray in row-major order, then the time required for data redistribution is $\Omega(n)$ in usually case, since there usually exists a worst-case scenario where a healthy

1	2	3	4	5	6
12	11	10	9	8	7
13	14	15	16	17	18
24	23	22	21	20	19
25	26	27	28	29	30
36	35	34	33	32	31

Figure 4. An example of blockwise snakelike order in an array with 36 blocks. A processor in block i is ranked before a processor in block j if $i < j$.

processor near the beginning of a row is mapped to a VSA node near the end of a VSA row. If a different mapping from the faulty array to its virtual subarray can be used, such as mapping $\Theta(n^{1-\frac{1}{d}}) \times \cdots \times \Theta(n^{1-\frac{1}{d}})$ blocks of healthy processors in a faulty array to $\Theta(n^{1-\frac{1}{d}}) \times \cdots \times \Theta(n^{1-\frac{1}{d}})$ blocks in a virtual subarray in blockwise snakelike order (see Fig. 4), we can show that the time required for data redistribution can be reduced to $o(dn)$. In [22], we have introduced algorithm *data redistribution (DR)* on 2-D arrays and analyze its performance. The algorithm can be extended to d-D arrays with $d \geq 3$ and the details are omitted.

Lemma 5.1 *Data redistribution from a d-D $n \times \cdots n$ array with $o(n^{1-\frac{1}{d}})$ faulty processors onto an appropriate virtual subarray, or its reverse process, can be performed in $o(hdn)$ steps, where h is the number of data items per healthy processor.*

Algorithm DR and its extensions for higher-dimensional arrays are generally-applicable methods for data redistribution to/from virtual subarrays, and is guaranteed to run in negligible time (e.g., $o(n_{min})$ time) when the number f of faults is small (e.g., $f = o(n_{min}^{1-\frac{1}{d}})$). In practice, comparable performance may also be achieved for much larger f, since worst-case scenarios are unlikely. Moreover, since algorithm DR will be invoked frequently if most problems are computed using all healthy and connected processors of an array, it is usually worth the effort to improve and optimize this process, even if only lower order terms of the total execution time can be improved. When data redistribution can be performed in negligible time, we have the following theorem for fault-tolerant computing in the entire faulty array.

Theorem 5.2 (Array Robustness Theorem III (ART-III))
Let T be the total time required for performing the algorithm on the fault-free array. If there exists an $m_1 \times m_2 \times \cdots \times m_d$ virtual subarray whose width overhead is $o(S)$, data redistribution and its inverse process can be performed in $o(T)$ time, and

(A) an algorithm for an array (i.e., mesh or torus) performs an average of S consecutive routing steps along each of the dimensions (at the same time) without any computation step in between and the virtual subarray is congestion-free, or

(B) an algorithm for a mesh performs S consecutive routing and computation steps along the same dimension on the average,

then the slowdown factor for performing the algorithm on the entire faulty array, each healthy and connected processor of which has at most h data items, is $1 + o(1)$ relative to a fault-free $m_1 \times m_2 \times \cdots \times m_d$ array of the same type with load at most $\lceil h(\prod_{i=1}^{d} n_i - f) / \prod_{i=1}^{d} m_i \rceil$.

When the number of faults is small and data items are mapped to blocks of appropriate size nearby, it is guaranteed that data redistribution can be performed in negligible time and that a congestion-free virtual subarray exists, leading to the following corollary.

Corollary 5.3 *Let T be the total time required for performing the algorithm on the fault-free array. If there are f faulty processors in an $n_1 \times n_2 \times \cdots n_d$ array with $f = o(\min(S, T_{min}^{1-\frac{1}{d}}, \frac{n_{min}}{hd}))$ (or $f = o(\min(S, T_{min}^{1-\frac{1}{d}}, \frac{n_{min}}{d}))$), blockwise snakelike mapping from all healthy processors to the virtual subarray is allowed in the emulation, and*

(A) an algorithm for an array (i.e., mesh or torus) performs an average of S consecutive routing steps along each of the dimensions (at the same time) without any computation step in between and the virtual subarray is congestion-free, or

(B) an algorithm for a mesh performs S consecutive routing and computation steps along the same dimension on the average,

then the slowdown factor for performing the algorithm on the entire faulty array, each healthy and connected processor of which has at most h data items, is $1 + o(1)$ relative to a fault-free $(n_1 - o(\frac{n_{min}}{hd})) \times (n_2 - o(\frac{n_{min}}{hd})) \times \cdots (n_d - o(\frac{n_{min}}{hd}))$ array of the same type with load at most $h + 1$ (or $h + o(h)$, respectively).

In what follows, we present results for performing total exchange in faulty meshes, tori, and n-ary d-cubes, as examples of how ART-III can be applied.

5.2. Solving problems based on ART-III

Since mappings similar to blockwise snakelike order [10] are usually allowed for the emulation of communication algorithms, we have the following corollary based on ART-III and Corollary 3.3.

Corollary 5.4 *A total of $2d - 1$ (or $2d - o(d)$) TE tasks can be executed in an n-ary d-cube that has an arbitrary pattern of f faulty processors and/or links in $dnN/4 + o(dnN)$ communication time, which is optimal within a factor of $1 + o(1)$ when d is not a constant, where $f = o(\min(n^{1-\frac{1}{d}}, n/d^2))$ (or $f = o(\min(n^{1-\frac{1}{d}}, n/d))$, respectively), $N = n^d - f$ is the number of healthy processors, and data items are input/output to/from each of the healthy and connected processors in the entire faulty n-ary d-cube.*

If we allow a packet to be split into $2d - o(d)$ mini-packets, the time required to perform a single TE task becomes $nN/8 + o(nN)$ communication time when d is not a constant and a mini-packet requires $1/(2d - o(d))$ time for transmission. This time complexity is also optimal within a factor of $1 + o(1)$ for both fault-free and faulty n-ary d-cubes.

Similar results can also be obtained for sorting and permutation routing in the entire faulty array. More details will be reported in the near future.

6. Conclusion

In this paper, we have proposed to add a RAIL middleware for interfacing the large body of ordinary algorithms/programs designed for fault-free arrays with a possibly faulty network/hardware. Based on RAIL, we formulated three array robustness theorems (ARTs), which lead to the fastest known solutions for a variety of important problems such as sorting, permutation routing, total-exchange, and ascend/descend algorithms on faulty arrays. ARTs classify computation and communication problems into several categories, each of which indicates the required RAIL techniques and quantifies the resultant performance according the available parameters such as the total number of faults.

An important implication of our results is that low-dimensional meshes, tori, and k-ary n-cubes are robust in that they can solve many problems with negligible slow-down in the presence of a (relatively) large number of faults. Thus, if low-dimensional meshes, tori, and k-ary n-cubes have comparable or even better performance than hypercubes and high-dimensional k-ary n-cubes (as indicated in [8] under certain assumptions), then the superiority of low-dimensional networks will be even more pronounced when the network/hardware may be faulty. The approach and techniques proposed in this paper can also be applied to a variety of other important problems as well as other network topologies. The details will be reported in the near future.

References

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

[2] Akers, S.B., D. Harel, and B. Krishnamurthy, "The star graph: an attractive alternative to the n-cube," *Proc. Int'l Conf. Parallel Processing,* 1987, pp. 393-400.

[3] Bertsekas, D.P. and J. Tsitsiklis, *Parallel and Distributed Computation – Numerical Methods,* Athena Scientific, 1997.

[4] Bhuyan, L.N. and D.P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Trans. Comput.,* vol. 33, no. 4, Apr. 1984, pp. 323-333.

[5] 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.

[6] Bruck, J. and R. Cypher, and C.-H. Ho, "Wildcard dimensions, coding theory and fault-tolerant meshes and hypercubes," *IEEE Trans. Comput.,* vol. 44, no. 1, Jan. 1995, pp. 150-155.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] Mazzaferri, R. and T.M. Murray, "The connection network class for fault tolerant meshes," *IEEE Trans. Comput.,* vol. 44, no. 1, Jan. 1995, pp. 131-138.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] Su, C.C. and K.G. Shin, "Adaptive fault-tolerant deadlock-free routing in meshes and hypercubes," *IEEE Trans. Comput.,* vol. 45, no. 6, Jun. 1996, pp. 666-683.

[19] Tzeng, N.-F. and G. Lin, "Maximum reconfiguration of 2-D mesh systems with faults," *Proc. Int'l Conf. Parallel Processing,* vol. 1, 1996. pp. 77-84.

[20] 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.

[21] 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.

[22] Yeh, C.-H., B. Parhami, H. Lee, and E.A. Varvarigos, "2.5n-step sorting on $n \times n$ meshes in the presence of $o(\sqrt{n})$ worst-case faults," *Proc. Merged Int'l Parallel Processing Symp. & Symp. Parallel and Distributed Processing,* Apr. 1999, pp. 436-440.

[23] Yeh, C.-H. and B. Parhami, E.A. Varvarigos, and Theodora A. Varvarigou, "RACE: a software-based fault tolerance scheme for systematically transforming ordinary algorithms to robust algorithms, *Proc. Int'l Parallel and Distributed Processing Symp.,* 2001.

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