RACE: A Software-Based Fault Tolerance Scheme for Systematically Transforming Ordinary Algorithms to Robust Algorithms

Chi-Hsiang Yeh, Behrooz Parhami, Emmanouel A. Varvarigos, and Theodora A. Varvarigou

Abstract

We propose the robust algorithm-configured emulation (RACE) scheme for efficient parallel computation and communication in the presence of faults. A wide variety of algorithms originally designed for fault-free meshes, tori, and k-ary n-cubes can be transformed to corresponding robust algorithms through RACE. In particular, optimal robust algorithms can be derived for total exchange (TE) and ascend/descend operations with a factor of $1+o(1)$ *slowdown. Also, RACE can tolerate a large number of faulty elements, without relying on hardware redundancy or any assumption about the availability of a complete subarray.*

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, small node-degree, and desirable algorithmic properties 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 [1, 9, 12]. 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.

In order to utilize the large body of algorithms already developed for fault-free meshes, tori, and n-ary d-cubes, many hardware-based schemes have been proposed to reconfigure a faulty array and ensure the availability of an intact array with desired dimensions in a very short time, despite the presence of faults [2, 4, 15]. Reconfiguration switching and standby sparing are examples of methods in this category. Such fault-tolerant systems are, however, expensive to implement and the number of faults that can be tolerated is limited by the redundancy of the hardware, which is in turn limited by the cost overhead that can be afforded.

A software approach based on robust algorithms [11] aims at designing programs that are easy to implement and can run on faulty meshes and tori efficiently, without having to rely on hardware redundancy. In [8], 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, where the fault rate p is a constant that is sufficiently small. In [10] we proposed a robust sorting algorithm based on shearsort, which can be executed on meshes with bypass capacity over faulty processors. We then showed in [16, 19] that 1-1 sorting (1 key per healthy and connected 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 (without any bypass capacity) with an arbitrary pattern of $o(\sqrt{n})$ faults. In [18], we further improved the sorting time to $2.5n + o(n)$ communication steps and $2n + o(n)$ comparison steps using a different ordering (i.e., a variant of blockwise snakelike order), which is asymptotically as fast as the fastest sorting algorithm for fault-free meshes (within a factor of $1 + o(1)$). These results demonstrate that some fairly complex problems such as sorting can be solved by faulty meshes in times comparable to those required in fault-free meshes. Several other robust algorithms or fault tolerance schemes can be found in [3, 6, 14].

In spite of the aforementioned positive results, it is impractical to redesign all algorithms one by one for faulty meshes and tori; hence, the motivation to devise a systematic method for transforming ordinary algorithms for fault-free networks to obtain fast (with a slowdown factor no more than $1+o(1)$) and easy-to-implement robust algorithms. No such transformation method has been reported in the literature thus far. One general fault tolerance scheme along this line was devised by Cole, Maggs, and Sitaraman [5], 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 $\varepsilon > 0$. This result is of great theoretical importance. However, their proposed emulation scheme is quite

Chi-Hsiang Yeh is with the Dept. of Electrical and Computer Engineering, Queen's University, Kingston, Ontario, K7L 3N6, Canada.

Behrooz Parhami is with the Dept. of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA.

Emmanouel A. Varvarigos is with the Dept. of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA. Work performed while with TU Delft, Netherlands.

Theodora A. Varvarigou is with the Division of Computer Science, Dept. of Electrical and Computer Engineering, National Technical University of Athens, GR-157 73, Athens, Greece.

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. We aim to develop another general emulation scheme that is far more efficient.

We propose the *robust algorithm-configured emulation (RACE) scheme* for fault-tolerant parallel computation and communication without relying on hardware redundancy. The RACE scheme essentially adds an "adaptation layer" between ordinary algorithms and the faulty network. From the algorithm 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 can be executed on such a platform without modifications; from the network point of view, the adaptation layer incorporates fault tolerance into the design of algorithms based on virtual subgraph embeddings, so that the RACE scheme in effect transforms ordinary algorithms to corresponding robust algorithms that can run on faulty networks. The resultant robust algorithms usually have 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. Also, when the number of faults is small, reconfiguration for RACE can be performed in a short time, and possible configurations for different patterns of faults may be precomputed and stored, if so desired, in a distributed manner and broadcast when needed. Moreover, the RACE scheme can work in combination with previous hardware-based fault tolerance schemes, by executing ordinary algorithms when the number of faults has not exceeded the limit of the hardware fault tolerance scheme, and executing robust algorithms otherwise.

2. Faulty arrays, emulation, and routing

In this section, we define the notion of virtual subarrays (VSA) in faulty arrays (with or without wraparound) and introduce the RACE scheme, an efficient fault tolerance scheme for computing and communication on faulty arrays without relying on hardware redundancy.

2.1. Virtual subarrays and reconfiguration

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. 1a). 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 *virtual columns*) of the virtual subarray. Figures 1b and 1c show the virtual rows and virtual columns of a 3×4 virtual subarray.

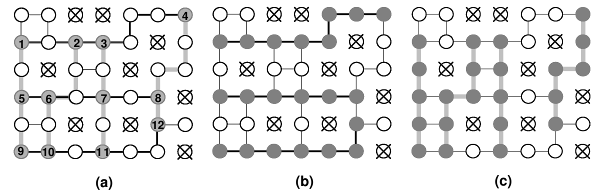

Figure 1. (a) A 3-by-4 VSM in a faulty 6-by-7 mesh with 9 faults. (b) Virtual rows of the VSM. (c) Virtual columns of the VSM.

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 1a 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 [9]. 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.

There usually exist many virtual subarrays in a faulty array. In general, to achieve better performance, we prefer to maximize the number of VSA nodes and minimize, for each dimension, the difference between the maximum length of virtual rows and the length m_i of a VSA row. The largest of these differences for all dimensions is called the *width overhead* of the virtual subarray. When there are $o(n)$ faulty processors or links in a d-D $n \times n \times \cdots \times n$ array, it is guaranteed that a d-D $(n - o(n)) \times (n - o(n)) \times \cdots \times (n - o(n))$ virtual subarray with width overhead $o(n)$ exists. In what follows we present a simple reconfiguration method for finding such a congestion-free virtual subarray in any N-node faulty array with $o(n)$ faults.

To reconfigure such a faulty array, we start with all the fault-free rows and columns as the candidate virtual rows and columns for a VSA. Then we check each of the candidate virtual rows and columns and determine whether it is located at the same hyperplane (i.e., $(d-1)$-D subarray) as at least one faulty processor or link. Finally, the VSA nodes are selected at the intersections of the candidate virtual rows and columns that pass the preceding examination (i.e., the rows and columns that are not at the same hyperplane as any fault). Since the embedded links are disjoint, the resultant VSA is a congestion-free virtual subarray. Also, the resultant width overhead is $o(n)$ since a virtual row or column can be "stretched" by only $o(n)$ hops when we have $o(n)$ faults.

Since the neighbors of a faulty processor or link can broadcast the corresponding error messages along all the hyperplanes to which the fault belongs in $O(dn)$ time, broadcasting f faults requires $O(fdn)$ time in the worst case, leading to $o(dn^2)$ time for reconfiguration. Note that, in usual cases, reconfiguration using this simple method requires $\Theta(dn)$ time since these broadcasting tasks can mostly be executed in parallel. More complicated methods that may require more time can also be used to find (somewhat) larger virtual subarrays. Since we only need to reconfigure a faulty array once after a new processor fails or recovers, the degradation of system performance due to such a reconfiguration is negligible in usual cases, regardless of whether a simple or complicated method is used.

2.2. The RACE scheme

In this subsection we present the RACE scheme, which redistributes the data on a faulty network to a virtual subgraph (e.g., an $m_1 \times m_2 \times \cdots \times m_d$ virtual subarray of a faulty array), 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 array algorithms on virtual subarrays.

Let M be the total number of data items in the virtual subarray and a be the *load*, which is the maximum number of items per processor. We then have

$$a = \left\lceil \frac{M}{\prod_{i=1}^{d} m_i} \right\rceil$$

when the data are spread approximately evenly on the virtual subarray. When there are a' data items per healthy and connected processor of the faulty mesh and the number of faults is not large, we usually have load $a = a' + 1$. The load for the virtual submesh in Fig. 1 is 3, assuming $a' = 1$ data item per healthy processor in the faulty mesh. If M data items are input/output to/from each healthy and connected processor, then we in general prefer to select a virtual submesh which minimizes the load a as well as the number of dummy items required. In a typical situation where the number of faults is smaller (e.g., $o(n)$), the load factor is usually increased by a factor of about $1/a'$ only, and most healthy processors (e.g., asymptotically $100 - o(1)\%$) are utilized and only a few healthy processors (e.g., $o(1)\%$) are wasted. Moreover, a virtual subarray with asymptotically the same size (within a factor of $1 + o(1)$) as the entire array can be found using the simple method of Subsection 2.1 that involves fault-free rows and columns only, and is thus very easy to implement.

We assume that a preprocessing stage has identified (perhaps at reconfiguration time) a virtual subgraph to be used. Then the proposed RACE scheme for transforming ordinary algorithms to robust algorithms can be represented in 3 stages:

The emulation scheme

- Stage 1: The data items to be processed are redistributed evenly to the processors on the virtual subgraph such that a processor has at most a items. On the virtual subgraph, 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 subgraph emulates a corresponding algorithm for a fault-free subgraph (e.g., 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 faulty network (e.g., the $n_1 \times n_2 \times \cdots \times n_d$ faulty array).

The virtual subgraph is typically an embedded graph of the same type with a smaller dimension (e.g., an embedded submesh in a faulty mesh or an embedded smaller hierarchical swap network in a faulty hierarchical swap network [17]); but it may also be an embedded graph of a different type than the original faulty graph, such as an embedded virtual subtorous in a faulty n-ary d-cube or an embedded virtual submesh in a faulty torus. An important feature of RACE is that the virtual subgraph used in RACE can be an embedded subgraph with dilation and/or congestion larger than 1. This is different from the strategy used in several previous papers for fault tolerance, which assumes the existence of a smaller complete mesh or an actual CCC subgraph or generalized Fibonacci cube (with dilation, congestion, and expansion all equal to 1) exists in a faulty mesh or hypercube. Note that such a strategy limits the number of faults tolerable (e.g., to as small as 3 in [7]). Our results show that this restriction is not necessary for the robust algorithms to have reasonable performance, at least for fault tolerance in meshes, tori, n-ary d-cubes, and some index-permutation graphs [17]. The embedded subgraphs for RACE should, however, still be carefully designed with certain looser restrictions (such as the ones for VSA) in order to achieve negligible slowdown.

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. 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. 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. Also, many algorithms, such as semigroup and prefix computations [11], can be emulated using SET with small overhead even if packet-switching is employed and the load is as small as 1.

The stepwise emulation technique, combined with the phase synchronization discipline (to be introduced in Section 3), will be used to formulate Theorem 3.1 for fault-tolerant computing on congestion-free virtual subarrays. We can also use techniques other than SET to implement Stage 2 of the RACE scheme. One example is the compaction/expansion technique used in [19]. Efficient implementation of Stages 1 and 3 leads to Theorem 4.2, derived in Section 4. A wide variety of important problems can then be solved efficiently in faulty arrays based on the RACE scheme, either utilizing all healthy processors or only processors on a virtual subarray.

2.3. Deadlock-free $X'Y'$ routing in virtual subarrays

Deadlock-free routing in a virtual subarray can be performed using a simple extension of XY routing. In a 2-D mesh, we first send a packet along the virtual row (i.e., in the X' direction) to which the source node belongs (Step 1), and then send it along the virtual column (i.e., in the Y' direction) to which the destination node belongs (Step 2). As long as there are at least two virtual channels per link, we can avoid deadlocks by using virtual channel 1 for transmitting packets in the X direction during Step 1 or in the Y direction during Step 2, and using virtual channel 2 for transmitting packets in the Y direction during Step 1 or in the X direction during Step 2. It can be verified that no cyclic dependency will arise so that wormhole routing is deadlock-free using this "$X'Y'$ routing" strategy.

$X'Y'$ routing can be easily extended to higher dimensional arrays. To obtain a deadlock-free routing scheme for the entire faulty array, we can use spanning trees to connect the remaining nodes to nodes on the virtual subarray, and then use the virtual subarray as a "backbone" to route packets. Using RACE, the stepwise emulation technique, and $X'Y'$ routing (or the strategy discussed previously that uses virtual channels), most ordinary wormhole-routing or virtual cut-through algorithms that have been developed without considering fault tolerance issues can be transformed into efficient robust algorithms. The details are omitted in this paper. In the following sections, we demonstrate how more complicated algorithms, based on either store-and-forward packet switching, virtual cut-through, and/or wormhole-routing, can be transformed into robust algorithms through virtual subarrays and RACE.

3. Fault-tolerant computing on VSAs

In this section we consider a special application of the RACE scheme where data items are input/output to/from VSA nodes.

3.1 Robust algorithms for congestion-free VSAs

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, where links of some/all dimensions can be used at a single step. We propose the *phase synchronization discipline* and show that such algorithms can be emulated on congestion-free virtual subarrays with negligible slowdown under this discipline and the RACE scheme.

With 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 nodes synchronize at the end of 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 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 *If an algorithm for an array (i.e., mesh or torus) performs an average of S consecutive routing steps along some/all of the dimensions (at the same time under the all-port communication model) 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-node $n_1 \times n_2 \times \cdots \times n_d$ array is $o(n_{min}/d)$, where

$n_{min} = \min(n_1, n_2, \ldots, n_d)$, it is always easy to find at least an $(n_1 - o(n_{min}/d)) \times (n_2 - o(n_{min}/d)) \times \cdots (n_d - o(n_{min}/d))$ congestion-free virtual subarray (which has $N - o(N)$ nodes) with width overhead $o(n_{min}/d)$, leading to the following corollary.

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

Note that the number of faulty processors and links that can be tolerated in Theorem 3.1 and Corollary 3.2 for low-dimensional arrays (i.e., small d) is not small. For example, an N-node 3-D mesh/torus can tolerate $\Theta(\sqrt[3]{N/\log N})$ faults with the size of the virtual subarray being $1 - o(1)$ times that of the entire array.

Communication algorithms, such as unicast, broadcast, total-exchange, and multinode broadcast, are at the heart of many applications [1]. Based on Theorem 3.1 and Corollary 3.2, we can show that these communication tasks as well as a variety of other important algorithms can be executed on congestion-free virtual subarrays with a slowdown factor of $1 + o(1)$ relative to the corresponding completion time in a fault-free array. The following corollary is offered as an example of the type of results that can be obtained from Theorem 3.1 regarding the total-exchange communication task [1] in faulty n-ary d-cubes.

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, 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. This communication time is asymptotically optimal within a factor of $1 + o(1)$.*

Many other important computation problems can also be performed efficiently based on Theorem 3.1. Ascend/descend algorithms, including FFT and permutation routing, form an important subclass of such algorithms and will be investigated in the following subsection for faulty n-ary d-cubes.

3.2 Robust ascend/descend algorithms on faulty n-ary d-cubes

Ascend/descend algorithms [9, 13] require successive operations on data items that are separated by a distance equal to a power of 2. Many applications, such as FFT, bitonic sort, matrix multiplication, and convolution, can be formulated using algorithms in this general category. Reduction (e.g., semigroup computation), parallel prefix computation, and many other algorithms developed for hypercubes and butterfly networks (e.g., broadcast algorithms) can also be formulated as ascend/descend algorithms or their variants.

To execute an ascend algorithm in a fault-free m-ary d-cube, where m is a power of 2, we simply send packets to nodes $1, 2, 4, \ldots, m/2$ hops away along dimension 1, and then send packets to nodes $1, 2, 4, \ldots, m/2$ hops away along dimension 2, and so on. Since the routing path between any source-destination pair in an ascend/descend algorithm belongs to a certain dimension and does not contain any turns, the length of a path in a faulty n-ary d-cube with width overhead $W_{overhead}$ is increased by at most $W_{overhead}$ hops compared to a fault-free m-ary d-cube. From Theorem 3.1, when $W_{overhead} = o(m/\log m)$, the slowdown factor for executing an ascend algorithm on the virtual m-ary d-subcube is only $1 + o(1)$ compared to a fault-free m-ary d-cube. Moreover, we can execute d ascend algorithms concurrently on the virtual m-ary d-subcube by rotating the dimensions.

4. Robust algorithms utilizing the entire faulty array

In Section 3, we have assumed that data items are input/output to/from a virtual subarray. By combining Theorem 3.1 with an algorithm for performing *data redistribution* [16, 17, 18, 19], which moves data from healthy and connected processors of the faulty array to the corresponding processors in the virtual subarray in negligible time, we formulate Theorem 4.2 for fault-tolerant computing and communication on the entire faulty array (in contrast to computing on a virtual subarray, as required by Theorem 3.1).

Theorem 4.1 *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 can be performed in $o(hdn)$ steps, where h is the number of data items per healthy processor.*

Theorem 4.2 *Let T be the total time required for performing an algorithm on the fault-free array, where the algorithm for an array (i.e., mesh or torus) performs an average of S consecutive routing steps along some/all of the dimensions (at the same time under the all-port communication model) without any computation step in between. 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 the virtual subarray is congestion-free, 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 a congestion-free virtual subarray exists and data redistribution can be performed in negligible time. Based on Theorem 4.2 and Corollary 3.3, we can derive the following corollary.

Corollary 4.3 *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 a message can be split into $2dk-o(dk)$ packets with any positive integer k, the time required to perform a single TE task becomes $nN/8+o(nN)$ communication time when d is not a constant, a message requires one time unit for transmission, and a packet requires $1/(2dk-o(dk))$ time unit 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.

5. Conclusion

In this paper, we have proposed the RACE scheme for transforming ordinary algorithms to obtain efficient robust algorithms. Based on RACE, we derived the fastest known robust algorithms for a variety of important problems such as total-exchange and ascend/descend computations on faulty arrays. The techniques used in this paper can also be applied to a variety of other important problems in parallel computation [1, 9, 12] and can be used to derive efficient robust algorithms for other network topologies.

References

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

[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] 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.

[4] Chen, Y.-Y., S.J. Upadhyaya, and C.-H. Cheng, A comprehensive reconfiguration scheme for fault-tolerant VLSI/WSI array processors, *IEEE Trans. Computers,* Vol. 46, no. 12, Dec. 1997, pp. 1363-1371.

[5] 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.

[6] Deconinck, G., V. De Florio, R. Lauwereins, and T.A. Varvarigou, "EFTOS: a software framework for more dependable embedded HPC applications," *Proc. Annual European Conf. Parallel Processing*, 1997, pp. 1363-1368.

[7] Jiang, F.-S., S.-J. Horng, and T.-W. Kao, "Embedding of generalized Fibonacci cubes in hypercubes with faulty nodes," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 8, no. 7, Jul. 1997, pp. 727-737.

[8] 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.

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

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

[11] 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.

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

[13] Preparata, F.P. and J.E. Vuillemin, "The cube-connected cycles: a versatile network for parallel computation," *Communications of the ACM,* vol. 24, no. 5, May 1981, pp. 300-309.

[14] 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.

[15] Varvarigou, T.A., V.P. Roychowdhury, and T. Kailath, "Reconfiguring processor arrays using multiple-track models: the 3-track-1-spare-approach," *IEEE Trans. Computers,* vol. 42, no. 11, Nov. 1993, pp. 1281-1293.

[16] 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.

[17] 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.

[18] 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.

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