The Robust-Algorithm Approach to Fault Tolerance on Processor Arrays: Fault Models, Fault Diameter, and Basic Algorithms

Behrooz Parhami and Chi-Hsiang Yeh

Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA

Abstract

With few exceptions, the two issues of algorithm design and fault tolerance for processor arrays have been dealt with separately, in that algorithm developers have assumed the availability of complete fault-free arrays and fault tolerance techniques have aimed at restoring such complete arrays by reconfiguring faulty ones. We present the design of robust algorithms that run efficiently on complete arrays but are also tolerant of faulty processors/links in a degraded mode. This is a complementary approach in that our algorithms can be used on reconfigurable arrays that tolerate a certain number of faults while maintaining their regularity, with the graceful degradation feature kicking in once the fault tolerance limit of the reconfiguration scheme is exceeded. The fault models considered in this paper comprise of the faulty processors/links being removed from the pool of resources (removal model) or bypassed in their respective rows/columns (bypass model). We discuss the two models, derive tight upper bounds for the fault diameter of the resulting networks, and present building-block algorithms for semigroup computation, parallel prefix computation, data rearrangement, matrix multiplication, and sorting.

Keywords: Adaptive parallel algorithms, Fault tolerance, Faulty meshes, Graceful degradation, Incomplete meshes.

1. Introduction

Robust algorithms, that run correctly in the presence of faults, have been extensively studied in two contexts. One is in adaptive and fault-tolerant routing. The other is in algorithm-based fault tolerance which is based on error detection/correction via problem-specific encoded data. Outside these particular domains, algorithm design and fault tolerance have generally been dealt with separately in that algorithm developers have assumed a fault-free mesh and fault tolerance techniques have aimed at restoring such a complete mesh by reconfiguring a faulty one.

Combined study of fault tolerance and algorithm design issues in parallel systems has been limited [6, 9, 10, 11]. One approach uses embedding of a less powerful architecture into a faulty array; e.g. a cube-connected cycles subgraph within a faulty hypercube [1, 2]. Another is based on an $n \times n$ mesh, containing a potentially large set of faulty elements, emulating a mesh with $\log n$ times as many processors with optimal $O(\log n)$ slowdown [5]. These approaches are theoretical and thus far have not lead to practical robust algorithms.

In this paper, we focus on the design of robust algorithms that can run directly on an incomplete or faulty mesh. We develop robust algorithms for several building-block computations (fan-in, parallel prefix, data routing, matrix multiplication, sorting) that run at almost full speed when the mesh is fault-free or when it contains a small number of faults and that offer gracefully degrading performance as more faults are encountered. Due to space limitation, all theorems and algorithms are given without proof.

2. Architecture and Fault Models

We consider both processor and link faults and assume that faults are detectable by any physically adjacent fault-free processor. Faults are considered permanent and diagnosis is off-line. Our algorithms adapt to fault conditions only in the sense of being able to run on any incomplete configuration (defined later). They do not tolerate faults during execution, but rather rely on prompt fault detection and rollback/restart for run-time fault tolerance.

We focus on $m \times n$ meshes (m rows, n columns, $m \le n$) with north, east, west, south (NEWS) nearest neighbors. We consider two fault models. The first one assumes that faulty nodes/links become unusable and thus removed from the resource pool, leading to gaps or holes in the regular mesh structure (Fig. 1b). We refer to this as (*incomplete*) *removal mesh*. Our second fault model assumes the ability to bypass a faulty node (and associated links) in its row/column (Fig. 1d). This logical fault model, which is what the algorithm designer sees and designs for, is called (*incomplete*) *bypass mesh*.

A removal mesh (Fig. 1b) can be disconnected by as few as 2 faults. However, the disconnected mesh is likely to have a sizable connected component that we use for continued computation. To compute on a removal mesh, we identify a submesh with a reasonably large number of complete subrows and subcolumns. For example, the solid nodes in Fig. 1c represent a 4×3 submesh of the 4×4 removal mesh of Fig. 1b which contains 2 complete (sub)columns and 3 complete subrows, with 6 nodes at their intersections. The submesh is chosen to maximize the number of nodes at the intersections of complete subrows and subcolumns, which are shaded in Fig. 1c. The shaded nodes thus obtained form a *virtual submesh* in which the *virtual neighbors* that are not physically adjacent can communicate via dilated paths.

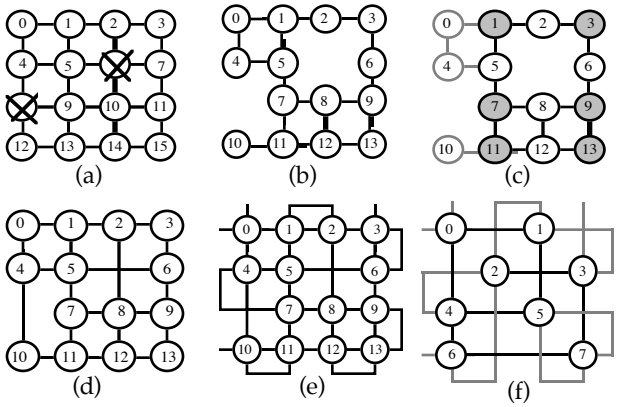

Fig. 1. Incomplete mesh: (a) mesh with 2 faulty nodes, (b) the removal fault model, (c) a 3×2 virtual submesh, (d) the bypass fault model, (e) bypass mesh with snake-like edge links added, (f) bypass mesh with many faults.

For a bypass mesh (Fig. 1d) to become disconnected, a large number of faults must be present. We will at times assume that snakelike edge links are added to our mesh (Fig. 1e). These edge links do not increase the node degree, or adversely affect the regular layout with short wires, but make some of the algorithms significantly more efficient. We require that a fault-free node remaining in operation within a bypass mesh continue to have access to its NEWS ports; although, due to bypassing, these ports in the faulty configuration may lead to processors other than the original neighbors in the fault-free mesh.

The removal fault model is realistic and easy to justify. Once faulty elements become known to fault-free nodes, the latter can simply avoid/ignore any communication to/from the faulty elements. The bypass fault model, on the other hand, may raise some questions about the ability and capacity for bypassing faulty nodes/links. Thus, we devote the rest of this section to justifying the bypass fault model and its advantages over the removal model.

The bypass fault model, though more restrictive than the removal fault model, leads to relatively cleaner and more efficient parallel algorithms. It may thus be worthwhile to include special hardware features in the mesh to guarantee adherence to the bypass fault model. We have previously discussed possible physical implementations that support the bypass model [8]. From among these, the use of separable row/column buses to circumvent faulty elements appears to be the most practical and cost-effective.

Note that once the bypassing mechanism is in place, it can be used not only for getting around faulty elements but also for performance enhancement. For example, one of our robust sorting algorithms is based on moving the data into $n/2$ columns, ordering them on the narrower $m \times n/2$ mesh, and finally distributing them back to their origins. Since the $n/2$ columns used in the middle phase are not necessarily contiguous, the ability to bypass intervening columns leads to faster communications among the nodes involved in executing this phase and, thus, to a more efficient robust sorting algorithm.

Since link faults are handled in much the same way as node faults, henceforth we focus mostly on node faults. If in Fig. 1a, instead of node 6 being faulty, the link from node 2 to node 6 is faulty, then node 6 could belong to a complete (sub)row but not to a complete (sub)column.

3. A Basis for Robust Algorithms

Certain topological properties of removal and bypass meshes are relevant to developing efficient robust algorithms. A row/column of the mesh in which all nodes and links are intact is called a *complete row/column*. Any incomplete mesh is virtually guaranteed to have at least one complete row/column and it has at least $m/2$ ($n/2$) complete rows (columns) with high probability.

We call two adjacent columns *pairwise complete* if there exists at least one path from the top row to the bottom row of the mesh that visits the rows in ascending order and is completely contained within the two columns. Pairwise complete rows are defined analogously (e.g., the middle two rows in Fig. 1b are pairwise complete). Pairwise complete columns can be detected, and their nodes marked, quite easily by a signal traveling down the column pair; when the next available south link is unavailable or is a bypass link, the signal switches to the other column, where it continues its downward propagation. If the signal reaches the bottom row, the two columns are pairwise complete and the nodes on the path of the signal can be used to emulate a complete column. This emulation is more efficient if the emulating path constructed above has a minimum number of bends.

Theorem 1: If columns j and $j+1$ are pairwise complete, then the greedy path building algorithm in which the signal stays in one column until forced to switch, when applied with each of the columns as the starting point, yields a path with the minimum number of bends. ∎

A general method that we use for developing robust algorithms is *virtual submesh emulation*. We develop this method using complete rows/columns (Fig. 2a), but it can be applied with pairwise complete rows/columns (Fig. 2b) or even *blockwise complete rows/columns*, making their tolerance level, and survival probability, quite high.

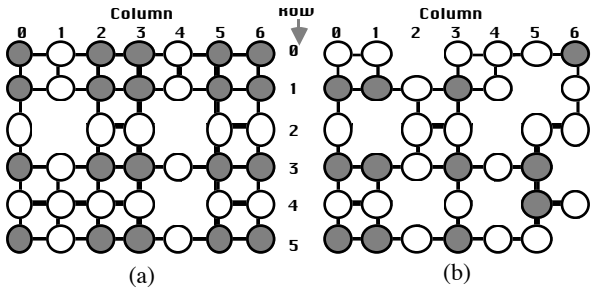

Fig. 2. Virtual submesh examples with (a) complete rows/columns and (b) pairwise complete rows/columns.

Once complete rows/columns have been identified in the entire $m \times n$ mesh or in a smaller $m' \times n'$ submesh, the nodes at their intersections comprise a *virtual submesh*, so named because they may have to communicate with their virtual NEWS neighbors via dilated paths. Typically, such a virtual submesh is composed of a large fraction of the original mn processors (say, $p' = m'n' \geq mn/2$). Thus, one strategy for executing mesh algorithms is to transfer all the data to the nodes of the virtual submesh, perform the algorithm on this virtual complete mesh, and route the results back to the original nodes.

Robust mesh algorithm via virtual submesh emulation

Stage 1 (Data packing): Data items are redistributed from healthy nodes to nodes on the virtual submesh, such that each node holds roughly the same number of items.

Stage 2 (Computation): The virtual submesh emulates the corresponding algorithm of a complete $m' \times n'$ mesh.

Stage 3 (Data unpacking): The computed results are distributed, in proper order, from the virtual submesh back to the healthy processors, if needed.

The *fault diameter* of a network is defined as the diameter of its surviving part after the occurrence of certain faults. The following theorems suggest that the fault diameters of removal and bypass meshes are quite small. While the fault diameter of k-ary n-cube has been studied before [4], we know of no corresponding result for meshes. Extreme cases, such as Fig. 1f, make these results non-trivial.

Theorem 2: The diameter of any connected component of an $m \times n$ removal mesh with f faulty nodes is at most $m+n-2+2f$. This bound is asymptotically tight; i.e., there exist removal meshes whose diameters are $m+n+2f-o(f)$. ∎

Theorem 3: The diameter of any connected component of an $m \times n$ bypass mesh is at most $m+n-2$, regardless of the number f of faults. This bound is tight. ∎

To use virtual submesh emulation, we need to specify the data packing (Stage 1) and complete submesh emulation (Stage 2) strategies. These will be discussed in Sections 5 and 6 of the paper. Data unpacking need not be dealt with separately as it is simply the reverse of Stage 1. Before considering the packing problem, which is a special case of data routing, we discuss data routing in general.

4. Robust Routing on Bypass Meshes

Data routing on faulty (removal) meshes has received a great deal of attention and many algorithms are available for this purpose. Any routing algorithm that works on a removal mesh is trivially applicable to a bypass mesh. Thus, in this section, we focus on algorithms for routing on bypass meshes that are simpler and/or more efficient than those for removal meshes.

One routing strategy is based on complete rows/columns. We assume that each node not on a complete row/column knows in which direction the nearest such row/column can be reached (ties are broken randomly). Routing is then accomplished by the following local decisions within each node (choice of direction is implicit):

Robust data routing algorithm for a bypass mesh

if data is at the destination row and column
then remove data
else if data is in the destination row or column
  then send along the destination row or column
  else if data is in a complete row/column
    then send along the complete row/column
    else send toward a complete row/column

The analysis for routing can be performed in three ways. First, with sparse transfers, we ignore conflicts and base the analysis solely on graph-theoretic routing distances. With $m, n \geq 3$, the worst-case routing distance becomes $m+n-5+\max(m,n)$ or $3n-5$ for an $n \times n$ mesh. In small meshes, it is relatively easy to construct reasonably sized fault patterns that lead to this worst case, but with larger meshes we almost never even get close to this bound. Note that the worst-case distance above is greater than the diameter of the bypass mesh (Theorem 3) since the local routing decisions may be globally suboptimal.

A second analysis method is to estimate the worst-case excess communication load placed on nodes belonging to complete rows/columns. Suppose that in an $n \times n$ mesh, there are $n-2\delta$ complete rows and $n-2\delta$ complete columns, with the 2δ incomplete rows/columns being consecutive. Then, traffic on the complete rows/columns that sandwich the incomplete ones can increase by a factor $\delta+1$. This need not be the case if, once a message is in a complete row/column, it does not necessarily stay there until it gets to the destination column/row. One can modify lines 5- 6 of the algorithm to force switching between various complete rows/columns for traffic balancing. For example, if the number of complete columns among the next few columns is known, turning can be done probabilistically.

An experimental approach shows the performance to be much better than the above upper bounds for random routing problems and random faults. Most packets, e.g. those with source nodes on a complete row/column, are routed along a shortest path and from the remaining packets, most will experience only slight increases in path lengths or routing delays. Fig. 3 shows that the average change in routing delay compared to that of a complete mesh is indeed small; the change can be negative due to some paths becoming shorter in the bypass mesh.

The path selection process discussed for packet routing can be used for wormhole routing as well; the simplicity of the decision algorithm for selecting one of the four outgoing channels in our routing algorithm ensures that wormhole routing can be performed quite efficiently. As in complete meshes, deadlocks can be dealt with in two ways: (1) detection followed by recovery, and (2) avoidance. Methods based on detection and recovery are no different for bypass meshes than for complete meshes. Deadlock avoidance is also quite simple if we use a modified form of the dimension-ordered routing.

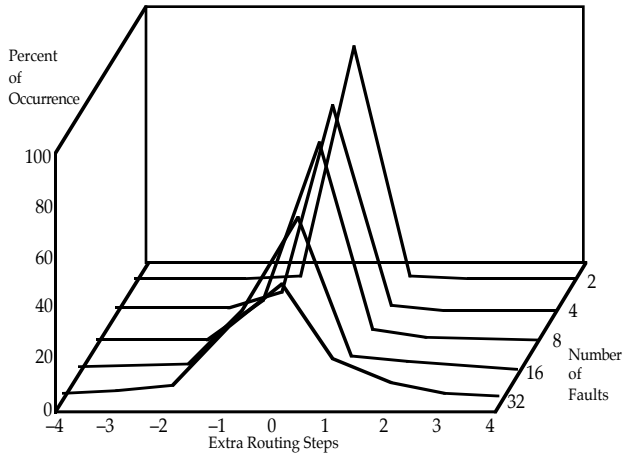

Fig. 3. Distribution of the change in routing time on a 32×32 bypass mesh compared to complete mesh.

In the modified row-first routing algorithm, all messages are routed to the nearest complete row, then to the destination column, and finally to the destination node. The initial column routing phase that takes the message to a nearest complete row cannot lead to deadlock since in any intermediate (incomplete) row, the cross message traffic is limited to those destined for that row and thus require no further direction change. In a similar fashion, other mesh routing algorithms can be adapted for a bypass mesh. Again, the initial column or row routing phase to get the message to a nearest complete row or column will not create additional opportunities for deadlock. The same observation applies to collective communication using either packet routing or wormhole routing methods.

5. Packing Data onto a Virtual Submesh

When each of the $p-f$ healthy processors has one data item, packing consists of sending the data items to the $p' = m'n'$ processors of the virtual submesh such that each processor holds at most $a' = \lceil (p-f)/p' \rceil$ items. The load factor a' satisfies $a' \leq 2$ with high probability and packing, if needed, is often a 1-2 routing problem. When each healthy processor begins with a load factor a, where a is relatively large, then $a' = \lceil a(p-f)/p' \rceil$ will be very close to a.

Packing is simple and very efficient on a bypass mesh. Since the nodes of the virtual submesh belong to complete rows and complete columns, either row-first or column-first routing can pack the data onto the virtual submesh with virtually no conflict.

Since, to our knowledge, no existing adaptive routing algorithm provides a guaranteed upper bound for the time required for our data packing problem with $O(n^{1/2})$ faults on a removal mesh, we propose and analyze an optimal algorithm for this purpose. For simplicity, we restrict our discussion to square $n \times n$ meshes.

Theorem 4: In an $n \times n$ removal mesh with $\Theta(n^{1/2})$ faulty nodes, no algorithm can pack the data to complete rows/columns in $o(n)$ time for arbitrary fault patterns. ∎

The following algorithm packs the data with a guaranteed delay upper bound when the number of faults is $O(n^{1/2})$. For this algorithm, the $n \times n$ mesh is divided into submeshes of size $\Theta(n^{1/2}) \times \Theta(n^{1/2})$. A *working subrow* is simply a complete subrow within a submesh. A *working subcolumn* is a subcolumn that is part of a complete column in the entire mesh. Note that a working subrow can be part of an incomplete row and that there are $\Theta(n^{1/2})$ of each if the sub meshes are made large enough.

Robust data packing on a removal mesh

Phase 1: In each submesh, any data item in a node that is not on a working subrow or a working subcolumn is routed to one of the working subrows/subcolumns.

Phase 2: In each submesh, if a processor that does not belong to a working subrow has more than a small constant number of data items, its data items are routed along working subcolumns to a nearby working subrow.

Phase 3: In each submesh, all data items are spread (approximately) evenly along the working subrows to the nodes at the intersection of working subrows and working subcolumns. At this point, a working subcolumn has $\Theta(n^{1/2})$ data items.

Phase 4: Each data item is sent along the complete column where it resides to the complete row where it will reside following data packing.

Phase 5: Each data item is sent from the complete row where it resides to the desired final position.

Theorem 5: The above robust data packing algorithm can pack data from fault-free processors each having O(1) data items onto the virtual submesh in optimal $O(n)$ time in an $n \times n$ mesh with $O(n^{1/2})$ faulty processors for arbitrary fault patterns. Furthermore, when $f = \Omega(n^{1/2})$ and $f = o(n)$, the data packing time is $O(f^2)$ which is optimal. ∎

6. Simple Parallel Algorithms

In this section we show that semigroup computation, parallel prefix computation, matrix multiplication, and FFT can be performed on incomplete meshes in optimal time. Note, in particular, that the robust algorithms run at full speed when there is no fault, since in this case packing or unpacking becomes unnecessary.

Clearly, the virtual submesh can perform a semigroup computation in $a'+2m+2n-4$ steps. On a bypass mesh, semigroup computation can be performed efficiently without packing or unpacking. Using strict unidirectional communication, a bypass mesh with f faulty nodes can perform a semigroup computation, with no slowdown, in $a+2m+2n-4$ steps, where a is the load factor of each fault-free node. We skip the description of a simple algorithm that uses a complete column for its computation and go directly to an asymptotically optimal algorithm that can use any column j with at least $m/2$ nodes to perform the combining of row results. Any row that does not have a node in column j must send its partial result to a node in that column. There are at most $m-1$ items to be sent which can be pipelined in $O(m+n)$ steps. After combining takes place in column j, the computation result is broadcast to every node in $m+n-2$ steps.

Semigroup computation on a bypass mesh

Phase 1: The operation is applied in a steps to the a values held in each node.

Phase 2: Values in each row are combined in $n-1$ steps, such that a node in Column j, or in another column if there is no node in Column j, obtains the value of the sub-expression corresponding to its row.

Phase 3: A processor not in Column j but holding a partial row result sends its value, tagged with its row number i, to any processor in Column j.

Phase 4: A processor in Column j upon obtaining or receiving a partial result for row i forwards it to the $\lfloor i/2 \rfloor$th fault-free processor in its column.

Phase 5: A fault-free node in Column j that holds two partial results (necessarily coming from consecutive rows) combines them before the final column reduction.

Phase 3 of the algorithm, where some values are sent to the nodes in column j, needs elaboration. One way to do this in $O(m+n)$ steps is to associate a "Column-j" tag with each node. The value of this tag is the direction of a shortest path to some node in Column j. By Theorem 3, the longest such path to Column j has length $m+n-2$. Contention adds no more than $m-1$ steps to the delay.

Assume that the parallel prefix algorithm is to generate s_i in the node initially holding a_i. Again we skip the simpler algorithm based on the use of a complete column, which requires $m+2n+\delta-3$ steps with unidirectional or $m+n-2$ steps with bidirectional communication, and go directly to a more robust, but less efficient, version that runs on any mesh with snakelike edge links as in Fig. 1e.

For clarity, we describe the algorithm for the case $n = p-f$; i.e., when each fault-free node holds one element. Starting with 0 for the topmost node in Column $n-1$, the first m nodes along the column snake are given sequence numbers.

Parallel prefix computation on a bypass mesh

Phase 1: Local prefixes for rows are computed in $n-1$ steps, with the rightmost node of each row holding the partial prefix result for the entire row.

Phase 2: The rightmost node in each row sends its result to a node in Column $n-1$; $O(m+n)$ steps with pipelining.

Phase 3: Upon obtaining or receiving a row-i result, a node in Column $n-1$ shifts it forward or backward along the snakelike path until it occupies the ith fault-free processor. The values may be spread over several columns due to bypassing along the snake.

Phase 4: Parallel prefix computation is performed along the snakelike path in $m-1$ steps.

Phase 5: The node that got its value from Row i sends its result to Row i where it is broadcast and merged with row prefixes. This phase takes $O(m+n)$ steps.

Both FFT and matrix multiplication algorithms can be performed on the virtual submesh in asymptotically optimal time, because the communication requirements of these algorithms are such that messages can be pipelined along the complete rows and columns of the virtual submesh, with the dilation penalty paid only once. Thus, the slowdown for these algorithms is dictated by the packing/unpacking phase. In particular, when the mesh contains $O(n^{1/2})$ faults, both of these algorithms will have optimal total running times based on Theorem 5.

7. Robust Sorting and Selection

For removal meshes, efficient robust sorting algorithms exist [11]. For example, 1-1 sorting on removal meshes, where each fault-free processor begins & ends with 1 item, needs $3n+o(n)$ communication and comparison steps on an $n \times n$ mesh with an arbitrary pattern of $o(n^{1/2})$ faulty nodes. Any removal-mesh algorithm can be used on a bypass mesh with no modification or with minor changes that do not affect the asymptotic time complexity but lead to constant-factor improvements. Thus, in the rest of this section, we focus on alternate methods for bypass meshes.

We have previously reported [9] a shearsort-based robust sorting algorithm for bypass meshes which, instead of guaranteeing sorted order after $\lceil \log_2 m \rceil$ iterations of row and column sorts, places a bound on the number of "dirty rows" (in the terminology of the proof method based on the 0-1 principle), thus allowing sorting to be completed with a limited number of odd-even transposition steps along the row-wise snake in Fig. 1e. However, shearsort is slow and robust shearsort imposes non-trivial performance penalties even with a fairly small number of faults.

A condition that allows us to sort fairly efficiently on a bypass mesh is to have at least $n/2$ complete columns (or $m/2$ complete rows). We assume, for simplicity, that there are f faulty nodes and that $mn–f$ values, initially distributed one per fault-free node, must be sorted (Fig. 4a).

Bypass-mesh robust sorting with $n/2$ complete columns

Phase 1: Values are shifted to $n/2$ complete columns in at most $n/2$ steps such that each node gets two values (one may be the padding value ∞). Prefix computations are performed over the $n/2$ complete columns to determine the number u of ∞ elements at or before each node along the row-wise snake (Fig. 4b)

Phase 2: Any 2-2 sorting algorithm (snakelike order) is used within the rectangular submesh formed by the $n/2$ complete columns. After sorting, the ∞ values reside in the last $\lceil f/2 \rceil$ nodes (Fig. 4c). To rearrange the items so that nodes end up with their initial number of elements, a node routes its smaller (larger) value $\lceil (u–1)/2 \rceil$ ($\lceil u/2 \rceil$) steps forward along the snakelike path (Fig. 4d).

Phase 3: The sorted values are distributed to the entire mesh by shifting along the rows.

To reduce the penalty in row/column sorts stemming from the non-adjacency of complete columns/rows, we use compaction/expansion [11], described below for rows.

Compaction/expansion method to speed up row sorts

Step 1: Row prefix computations yield the number l of good nodes on incomplete columns to the left of a node.

Step 2 (compaction): Shift left in row by l positions.

Step 3: Do row sort on the nodes which have items.

Step 4 (expansion): Sorted values are shifted to the right by the same amount as in Step 2.

In this way, row sort can be performed in $O(n)$ steps. Note that we need to do Step 1 once at reconfiguration time, so row sort requires at most $4f_r$ additional steps, where f_r is the maximum value of l (see Step 1). Since items are shifted back to processors located on complete columns after being sorted within the rows, column sort is done normally. Hence, we can use shearsort for sorting the items on the $m \times n/2$ mesh quite efficiently.

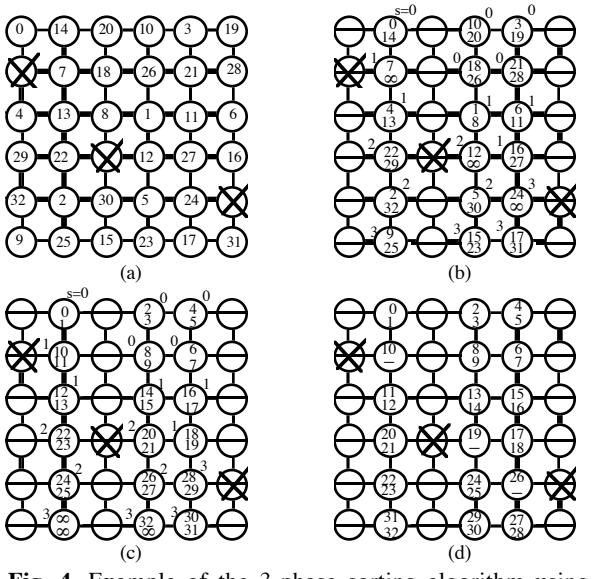

Fig. 4. Example of the 3-phase sorting algorithm using $n/2$ complete columns. (a) The original keys. (b) Keys packed onto 3 complete columns. (c) Sorted keys on the 6×3 mesh. (d) Keys redistributed per the row capacities.

By using row compaction/expansion to emulate the modified Schnorr/Shamir algorithm [Niga95], sorting on an $n \times n$ bypass mesh with $o(n^{1/4})$ faults in a row can be done in $2.5nt_r+3t_c+o(n)$ or $5nt_r+3t_c+o(n)$ steps for bidirectional or strict unidirectional meshes, respectively. More generally, if we compact to n/k complete columns, the number of communication steps approaches $2n+O(kf_r)$ or $4n+O(kf_r)$, respectively, for large k. We see that the performance of the above sorting algorithm is similar to the best known results for a fault-free $n \times n$ mesh.

To select the Mth largest element among N items (e.g. finding the median when $M = \lfloor N/2 \rfloor$), one can use sorting. This is viewed as inefficient on other models of parallel computation as there exist simpler selection algorithms. However, 2-D meshes, even with row/column buses, cannot do much better for selection than for sorting.

8. Conclusions

By designing robust algorithms that run directly on an incomplete bypass or removal mesh, we have demonstrated that certain computations can be performed efficiently, and with graceful degradation, on faulty meshes. These results pave the way for implementing more complex algorithms on incomplete meshes and lead to a novel fault tolerance strategy based on robust algorithms; a strategy that can replace or augment other fault tolerance schemes such as those based on reconfiguration. Extension of our results to higher-dimensional removal meshes is straightforward since virtual submeshes can be defined in much the same way. Higher-dimensional bypass meshes require further study.

Acknowledgments

Our sincere thanks go to Dr. Ching-Yu Hung, who contributed to the proof of Theorem 3 and to routing simulation of Fig. 3, and to Dr. Ding-Ming Kwai, who built counter-examples that helped in correcting an error in Theorem 2.

References

[1] N Bagherzadeh, M Dowd (1995), Computation in faulty stars, *IEEE Trans. Reliability*, vol 44, pp 114-119.

[2] J Bruck, R Cypher, D Soroker (1990), Running algorithms efficiently on faulty hypercubes, *ACM Symp. Parallel Algorithms & Architectures*, pp 36-44.

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

[4] K Day, A Al-Ayyoub (1997), Fault diameter of k-ary n-cube networks, *IEEE Trans. Parallel & Distributed Systems*, vol 8, pp 903-907.

[5] C Kaklamanis et al (1990), Asymptotically tight bounds for computing with faulty arrays of processors, *Symp. Foundations of Computer Science*, pp 285-296.

[6] A Mishra, Y Chang, L Bhuyan, F Lombardi (1995), Fault-tolerant sorting on SIMD hypercubes, *Int'l Parallel Processing Symp.*, pp. 312-318.

[7] M Nigam, S Sahni (1995), Sorting n^2 numbers on $n \times n$ meshes, *IEEE Trans. Parallel & Distributed Systems*, vol 6, pp 1221-1225.

[8] B Parhami (1993), Fault tolerance properties of mesh-connected parallel computers with separable row/column buses, *Midwest Symp. Circuits Systems*, pp 1128-1131.

[9] B Parhamii, C Y Hung (1995), Robust shearsort on incomplete bypass meshes, *Int'l Parallel Processing Symp.*, pp304-311.

[10] J-P Sheu, Y-S Chen, C-Y Chang (1992), Fault-tolerant sorting algorithm on hypercube multicomputers, *J. Parallel & Distributed Computing*, vol 16, pp185-197.

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