RECONFIGURING ARRAYS WITH FAULTS PART I: WORST-CASE FAULTS*

RICHARD J. COLE †, BRUCE M. MAGGS ‡, AND RAMESH K. SITARAMAN §

Abstract. In this paper we study the ability of array-based networks to tolerate worst-case faults. We show that an $N \times N$ two-dimensional array can sustain $N^{1-\epsilon}$ worst-case faults, for any fixed $\epsilon > 0$, and still emulate T steps of a fully functioning $N \times N$ array in $O(T+N)$ steps, i.e., with only constant slowdown. Previously it was known only that an array could tolerate a constant number of faults with constant slowdown. We also show that if faulty nodes are allowed to communicate, but not compute, then an N-node one-dimensional array can tolerate $\log^k N$ worst-case faults, for any constant $k > 0$, and still emulate a fault-free array with constant slowdown, and this bound is tight.

Key words. fault tolerance, array-based network, mesh network, network emulation

AMS subject classifications. 68M07, 68M10, 68M15, 68Q68

1. Introduction.

In a truly large parallel computer, some components are bound to fail. Knowing this, a programmer can write software that explicitly copes with faults in the computer. But building fault tolerance into every piece of software is cumbersome. The programmer would prefer to program a fault-free virtual computer and leave the job of coping with faults to the hardware. Ideally, the emulation of the fault-free computer should entail little slowdown, even if there are many faults in the actual hardware.

The emulation of the fault-free computer consists of two tasks. The faulty computer must emulate the computations performed by the processors of the fault-free computer, and it must emulate the communications between those processors. Emulating the computations does not incur much slowdown. The computation performed by each faulty processor is simply mapped to a fault-free processor. But once the computations are moved around, processors that are neighbors in the fault-free computer may no longer be neighbors in the faulty computer. Thus, there is a risk that the communications will be slowed down. The solution to this problem depends on the communication topology of the computer.

One of the most popular ways to construct a parallel computer is to arrange the processors as a two-dimensional or three-dimensional array. Commercial machines including the Cray T3D [10] and MasPar MP-1 [3] have this topology, as do experimental machines such as iWarp [4] and the J-Machine [18]. In this paper we study the ability of machines like these to tolerate faults. We show, for example, that an $N \times N$ two-dimensional array can sustain $N^{1-\epsilon}$ worst-case faults, for any fixed $\epsilon > 0$, and still emulate a fault-free $N \times N$ array with constant slowdown.

---

* Richard Cole is supported in part by NSF Grants No. CCR-92-02900 and CCR-95-03309. Bruce Maggs is supported in part by an NSF National Young Investigator Award, No. CCR-94-57766, with matching funds provided by NEC Research Institute, and by ARPA Contract F33615-93-1-1330. Ramesh Sitaraman is supported in part by NSF Grant No. CCR-94-10077. This research was conducted while the first author was visiting NEC Research Institute, the second author was employed at NEC Research Institute, and the third author was a student at Princeton University.

† Courant Institute, New York University, New York, NY 10012. (cole@cs.nyu.edu).

‡ School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. (bmm@cs.cmu.edu).

§ Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003. (ramesh@cs.umass.edu).

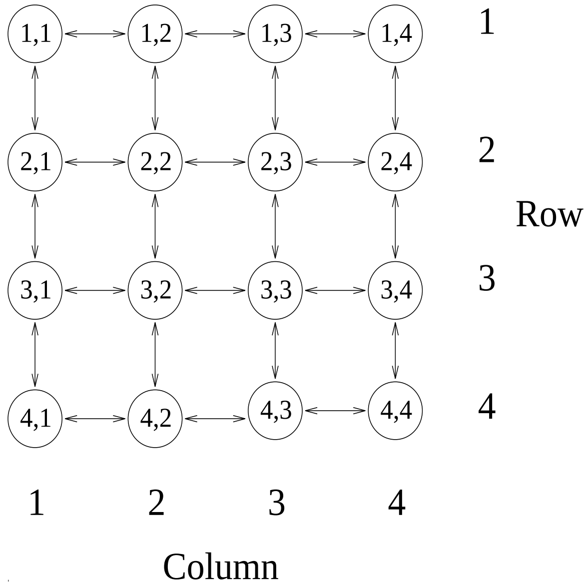

FIG. 1.1. *A* 4×4 *mesh.*

1.1. Arrays and the fault model. A d-dimensional *array* with side-length N consists of N^d nodes, each labeled with a distinct d-tuple $(r_1, r_2, \ldots, r_d)$, where $1 \le r_i \le N$ for $1 \le i \le d$. Two nodes are connected by a pair of oppositely directed edges if their labels differ by 1 in precisely one coordinate. For example, in a 4-dimensional array with side-length 8, nodes $(3,2,4,8)$ and $(3,2,3,8)$ are neighbors, but $(3,2,4,8)$ and $(3,2,3,7)$ are not. A two-dimensional array is also called a *mesh*. A 4×4 mesh is shown in Figure 1.1. For each i, the mesh nodes labeled (i,j), where $1 \le j \le N$, are said to belong to the i^{th} row. For each j, the mesh nodes labeled (i,j), where $1 \le i \le N$, are said to belong to the j^{th} column. Sometimes two nodes are considered to be neighbors if they differ in precisely one coordinate and their values in that coordinate are 1 and N. In this case we say that the array has *wrap-around* edges. A two-dimensional array with wrap-around edges is also called a *torus*. All of the results in this paper hold whether or not the array has wrap-around edges.

The nodes in an array represent processors and the edges represent communication links. We assume that the array operates in a synchronous fashion. At each time step, each node can receive a message from each of its neighbors, perform a simple local computation, and then send a message to each of its neighbors.

In this paper we assume that only nodes fail, that these failures are static, and that their locations are known. We also assume that the faults appear in a worst-case pattern, i.e., that an adversary decides where to put the faults in the network. We allow information about the locations of the faults to be used in reconfiguring the network. We assume that a faulty node can neither compute nor communicate. All of our results can be extended to handle edge failures by viewing an edge failure as the failure of one of the nodes incident on the edge. In Section 4, we use a weaker fault model for one-dimensional arrays by allowing faulty nodes to communicate, but not compute. We observe that even in this weaker fault model linear arrays cannot tolerate as many worst-case faults as two-dimensional arrays. In another paper, we consider random fault patterns. In the case of random faults, we assume that each node fails independently with some fixed probability p.

1.2. Embeddings. The simplest way to show that a network with faults, H, can emulate a fault-free network, G, is to find an embedding of G into H. We call H the *host* network and G the *guest* network. An embedding maps nodes of G to non-faulty nodes of H, and edges of G to non-faulty paths in H. The three important measures of an embedding are its load, congestion, and dilation. The *load* of an embedding is the maximum number of nodes of G that are mapped to any node of H. The *congestion* of an embedding is the maximum number of paths that use any edge of H. The *dilation* is the maximum length of any path. Given an embedding of G into H, H can emulate each step of the computation of G by routing a packet for each edge of G along the corresponding path in H. Leighton, Maggs, and Rao [11] showed that if the embedding has load l, congestion c, and dilation d, then the packets can be routed so that the slowdown of the emulation is $O(l + c + d)$.

In order for an embedding-based emulation scheme to have constant slowdown, the load, congestion, and dilation of the embedding must all be constant. Unfortunately, by placing $f(N)$ faults in an N-node 2- or 3-dimensional array H, where $f(N)$ is any function that is $\omega(1)$, it is possible to force either the load, congestion, or dilation of every embedding of an array G of the same size and dimension to be larger than a constant [7, 8, 12]. Similarly, if $\Theta(N)$ faults are placed in H at random, then with high probability every embedding of G in H will have $\omega(1)$ load, congestion, or dilation. Thus, in order to tolerate more than a constant number of worst-case faults or constant-probability failures, a more sophisticated emulation technique is required.

1.3. Redundant computation. All of the emulations in this paper use a technique called *redundant computation*. The basic idea is to allow H to emulate each node of G in more than one place. This extra freedom makes it possible to tolerate more faults, but it adds the complication of ensuring that different emulations of the same node of G remain consistent over time. The technique of redundant computation was previously used to tolerate faults in hypercubic networks [13], and to construct work-preserving emulations in fault-free networks [6, 9, 15, 16, 17, 21].

1.4. Previous work. A large number of researchers have studied the ability of arrays and other networks to tolerate faults. The most relevant papers are described below.

Raghavan [20] devised a randomized algorithm for solving one-to-one routing problems on $N \times N$ meshes. He showed that even if each node fails with some fixed probability $p \leq .29$, then for almost all random fault patterns, any packet that can reach its destination does so in $O(N \log N)$ steps, with high probability. Mathies [14] improved the $p \leq .29$ bound to $p \approx .4$.

Kaklamanis et al. [8] improved Raghavan's result by devising a deterministic routing algorithm. For almost all random fault patterns, the algorithm guarantees that any packet that can reach its destination does so within $O(N)$ steps. This algorithm can also tolerate worst-case faults. If there are k faults in the network, it runs in time $O(N + k^2)$. Kaklamanis et al. also showed that an $N \times N$ mesh with constant-probability failures or $\Theta(N)$ worst-case faults can sort or route N^2 items, or multiply two $N \times N$ matrices in $O(N)$ time. They also showed that, with high probability, an $N \times N$ mesh with constant-probability failures can emulate a fault-free $N\sqrt{\log N} \times N\sqrt{\log N}$ mesh with $O(\log N)$ slowdown. (Throughout this paper the base of the function log is 2.)

Aumann and Ben-Or [2] used Rabin's information dispersal technique [19] to show that an $N \times N$ mesh H with slack s, $s = \Omega(\log N \log\log N)$, can emulate a fault-free $N \times N$ mesh G with slack s with constant slowdown, even if every node or edge in H fails with some fixed probability $p > 0$ at some point *during* the emulation. (In a slack s computation, each node v in G emulates s virtual nodes. In each *superstep*, v emulates one step of each virtual node, and each virtual node can transmit a message to one of v's four neighbors.) Aumann and Ben-Or assumed that in a single step, an edge in H can transmit a message that is $\log N$ times as large as the largest message that can be transmitted in a single step by G.

Bruck, Cypher, and Ho [5] showed that by adding some spare nodes and edges to a mesh, it is possible for the mesh to sustain many faults and still contain a working fault-free $N \times N$ mesh as a subgraph. In particular, they showed that by adding $O(k^3)$ spare nodes, it is possible to tolerate k worst-case faults, and by adding k spare nodes, for $k = O(N^{2/3})$, it is possible to tolerate k random faults, with high probability. In both cases, the networks have bounded degree. Tamaki [24] showed how to construct an $O(N)$-node network with degree $O(\log\log N)$ with the property that, for any $d \geq 2$, even if every node fails with constant probability, with high probability the network contains a fault-free N-node d-dimensional array as a subgraph. He also showed how to construct a bounded-degree network with the property that even if $N^{(1-2^{-d})/d}$ worst-case faults are placed in the network, the network is guaranteed to contain an N-node d-dimensional array as a subgraph. Ajtai et al. [1] analyzed the technique of adding spare nodes to larger classes of networks that include meshes. In all of these constructions the VLSI layout area requirements of the networks with spare nodes and edges are much larger than those of the arrays that they contain as subgraphs.

Leighton, Maggs, and Sitaraman [13] showed that an N-node butterfly can tolerate $N^{1-\epsilon}$ worst-case faults, for any fixed $\epsilon > 0$, and still emulate a fault-free N-node butterfly with constant slowdown. They proved the same result for the shuffle-exchange network. They also showed that, for any constant $k > 0$, an N-node mesh of trees can tolerate $\log^k N$ worst-case faults and still emulate a fault-free mesh of trees with constant slowdown. Finally, they showed that, with high probability, an N-node butterfly (or shuffle-exchange network) can tolerate constant-probability failures with slowdown $2^{O(\log^* N)}$. Tamaki independently showed that, with high probability, an N-node butterfly can be embedded in an N-node butterfly containing constant-probability node failures with load $O(1)$, congestion $O((\log\log N)^{8.2})$, and dilation $O((\log\log N)^{2.6})$ [22]. In [23], he proved a similar result for a class of networks called cube-connected arrays.

1.5. Our results. In Section 2 we show that an $N \times N$ array can tolerate $\log^k N$ worst-case faults, for any constant $k > 0$, and still emulate T steps of a fault-free array in $O(T + N)$ steps, i.e., with constant slowdown. Previously it was only known that a constant number of worst-case faults could be tolerated with constant slowdown. Section 2 introduces most of the terminology that is used throughout this paper.

In Section 3 we present a method called *multi-scale emulation* for tolerating $N^{1-\epsilon}$ worst-case faults on an $N \times N$ mesh with constant slowdown, for any fixed $\epsilon > 0$. This result nearly matches the $O(N)$ upper bound on the number of worst-case faults that can be tolerated with constant slowdown.

In Section 4 we show that if faulty nodes are allowed to communicate, but not compute, then an N-node one-dimensional array can tolerate $\log^k N$ worst-case faults, for any constant $k > 0$, and still emulate a fault-free N-node linear array only with constant slowdown. We also show that an N-node linear array cannot tolerate more than $\log^k N$ worst-case faults without suffering more than constant slowdown, pro-

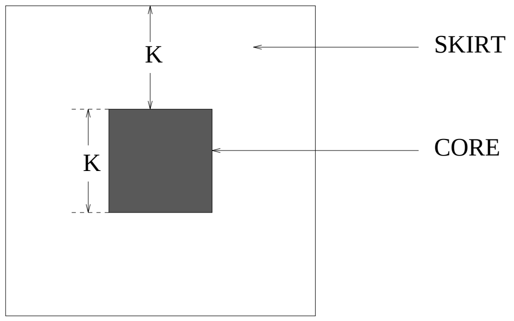

FIG. 2.1. *A finished box*

vided that the emulation is static. In a *static* emulation, each host node a emulates a fixed set $\psi(a)$ of guest nodes. Redundant computation is allowed; a guest node u may belong to $\psi(a)$ and $\psi(b)$ for distinct host nodes a and b. In this case we say that there are multiple *instances* of the guest node u. For each guest time step, host node a emulates the computation performed by each node u in $\psi(a)$. Furthermore, for every guest edge $e = (v, u)$ into u, for each instance u' of u in the host, there is a corresponding instance v' of v at some host node such that for each guest time step *the same* instance v' sends a packet for the edge e to u'. (Note that v' may also send packets to other instances of u.) The emulations that use redundant computation in this paper and in [6, 9, 13, 21] are all static.

2. A simple method for tolerating worst-case faults on the mesh. In this section, we show that, for any constant $k > 0$, an $N \times N$ mesh with $\log^k N$ worst-case faults can emulate any computation of an $N \times N$ fault-free mesh with only constant slowdown. The procedure for reconfiguring the computation around faults consists of two steps. The first is a process by which the faults are enclosed within square regions of the mesh called *boxes*. We call this step the *growth process*. We describe this process in Section 2.1. The next is an emulation technique that maps the computation of the fault-free mesh (the guest) to nodes in the faulty mesh (the host). The boxes grown in the first step determines how the mapping of the computation is done. This process is described Section 2.2. For simplicity, we assume that the mesh has wrap-around edges. This assumption can be easily done away with at the cost of considering some special cases for faults near the border of the mesh.

2.1. The growth process. The growth process grows boxes on the faulty mesh, i.e., the host. There are two types of boxes. The first type is called a *core*. A core has too many faults in it to perform any role in the emulation. The second type is a *finished box*. A finished box can emulate a submesh of the same side-length with constant slowdown. (The side-length of a box or submesh is the number of nodes on each side, i.e., a box or submesh with side-length k has k^2 nodes.) A finished box of side-length $3k$ consists of a core of side-length k surrounded by a *skirt*, which contains no faults, of width k as shown in Figure 2.1. (Like side-length, width is measured in nodes.)

At every stage of the growth process, the algorithm maintains a set of boxes, some of which are cores while others are finished boxes. At the beginning of the growth process, every fault is enclosed in a box with unit side-length which is a core. In every stage of the growth process, we pick a core, say of side-length k, and grow a skirt of width k around it. If the core or the skirt intersects some other core, we merge the two cores to form a new core whose boundary is a smallest square that contains both cores. If the core or the skirt intersects a finished box, we find a smallest square box that contains both the core and the core of the finished box and turn this bounding box into a core. We also remove the finished box from the list of finished boxes. If the skirt does not intersect any other boxes, the newly created box is labeled a finished box. We continue applying these rules until either some core grows to be too large to grow a skirt around it or no core remains and no two finished boxes intersect. For the former outcome, some core must have side-length greater than $N/3$. (Recall that the mesh has wrap-around edges.) In Lemma 2.1 we show that this cannot happen if there are fewer than $(\log N)/2$ faults.

Our rules for growing cores assign each fault to a unique core. Initially, every fault is assigned to the unit-sized core enclosing it. Inductively, when two or more cores are merged to form a new core, every fault assigned to the old cores is now assigned to the new core. A core is said to *contain* all the faults assigned to it. Note that if two cores overlap, it is possible for a fault to be geometrically located inside of a core and yet not be contained by that core.

LEMMA 2.1. *If the number of faults is less than* $(\log N)/2$*, then the growth process terminates with non-overlapping finished boxes.*

Proof. Let $F(k)$ denote the minimum number of faults that a core of side-length k must contain. We show by induction that $F(k) \geq (\log k)/2 + 1$. As the base case, $F(1) = 1$, which satisfies the hypothesis. Assume that we have a core of side-length $k > 1$. This core must have been created by merging two cores according to one of the two merging rules stated previously. Let x and y denote the side-lengths of these two cores. In both cases, $x + y \geq \lfloor k/2 \rfloor + 1$. Using the inductive hypothesis, we have

$$\begin{aligned} F(k) &\geq F(x) + F(y) \\ &\geq (\log x)/2 + 1 + (\log y)/2 + 1 \end{aligned}$$

The values of x and y that minimize the right-hand side of this inequality are $x = \lfloor k/2 \rfloor$ and $y = 1$. Substituting these values, we have

$$\begin{aligned} F(k) &\geq (\log \lfloor k/2 \rfloor)/2 + 2 \\ &\geq (\log k)/2 + 1 \qquad (2.1) \end{aligned}$$

This proves our inductive hypothesis.

Now suppose that there is a core of side-length greater than $N/3$. Then it must contain at least $F(\lfloor N/3 \rfloor + 1)$ faults, which is more than $(\log N)/2$, which is a contradiction. Therefore, the growth process must terminate with a set of non-intersecting finished boxes. □

2.2. The Emulation. In this section, we show that if the growth process terminates with a set of non-intersecting finished boxes, then the host H can emulate the guest G with constant slowdown.

The emulation of G by H is described as a pebbling process. There are two kinds of pebbles. With every node v of G and every time step t, we associate a state pebble (s-pebble), $\langle v, t \rangle$, which contains the entire state of the computation performed at node v at time t. With each directed edge e in G and every time step t, we associate a communication pebble (c-pebble), $[e, t]$, which contains the message transmitted along edge e at time step t.

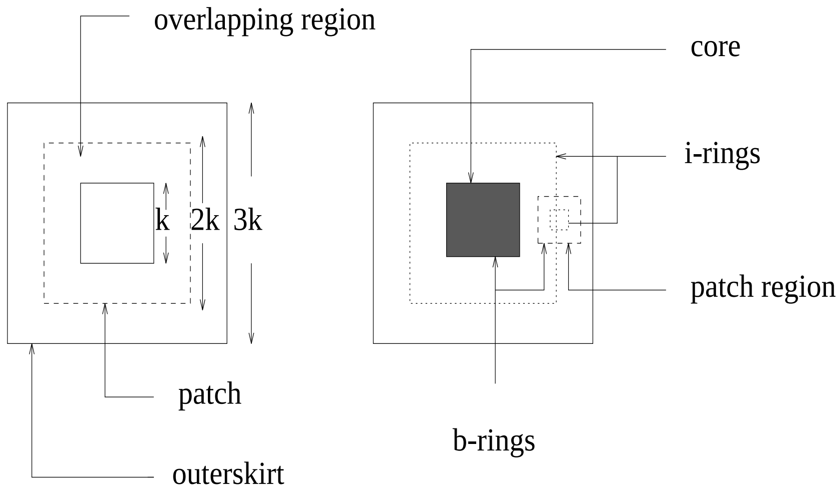

FIG. 2.2. *Mapping the computation inside a finished box. A finished box in H is shown on the right, and the corresponding submesh in G is shown on the left.*

The host H will emulate each step t of G by creating at least one s-pebble $\langle v,t\rangle$ for each node v of G and a c-pebble $[e,t]$ for each edge e of G. A node of H can create an s-pebble $\langle v,t\rangle$ only if it contains s-pebble $\langle v,t-1\rangle$ and all of the c-pebbles $[e,t-1]$, where e is an edge into v. The creation of an s-pebble takes unit time. A node of H can create a c-pebble $[g,t]$ for an edge g out of v only if it contains an s-pebble $\langle v,t\rangle$. The creation of a c-pebble also takes unit time. Finally, a node of H can transmit a c-pebble to a neighboring node in H in unit time. A node of H is not permitted to transmit an s-pebble since an s-pebble may contain a lot of information. All of our emulations are static, i.e., each node of H emulates a fixed set of nodes of G, and for each guest edge $e=(v,u)$ and each instance of u, there is a corresponding instance of v such that for each guest time step, the host node creating s-pebbles for that instance of v sends a c-pebble $[e,t]$ to the host node creating s-pebbles for u. Initially, each node of H contains an s-pebble $\langle v,0\rangle$ for each node v of G that is mapped to it.

Using the growth process of the previous section, we grow a collection of non-intersecting finished boxes on the faulty mesh H. If the faulty mesh H has fewer than $(\log N)/2$ faults, then the growth process will terminate with a set of non-intersecting finished boxes. Every node of H that does not belong to any of the finished boxes will emulate the computation of the corresponding node of G. Every finished box of H will be responsible for emulating the corresponding submesh of G. However, since some of the nodes inside the core of a finished box are faulty, we must make sure that no computation is mapped to them. In fact, there will be no computation mapped to any node inside a core. All the computations will be mapped to the skirt of the finished box which is completely fault-free. Since we would like each finished box to do its share of the emulation with constant slowdown, we need to avoid long communication delays caused by the fact that the core is unusable. As we shall see, we can hide the latency involved in sending c-pebbles long distances across the mesh using a technique called *redundant computation*. In an emulation that performs redundant computation, some nodes of G are emulated by more than one node in H.

The computation of G corresponding to a finished box is mapped with replication to the skirt of that finished box as follows (See Figure 2.2). Suppose that the core has side-length k and the finished box has side-length $3k$. We begin by dividing the submesh of G corresponding to a finished box into two regions, the *patch* and the *outerskirt*. The patch is a square region of side-length $2k$ whose center is also the center of the finished box. The outerskirt consists of the entire submesh of G with a square of side-length k removed from its center. As shown in Figure 2.2, the patch and the host overlap in an annular region of width k. The patch and the outerskirt are mapped to the finished box as follows. The outerskirt is the same size and shape as the skirt of the finished box; every node in the outerskirt is mapped to its corresponding node in the skirt. The patch, which is a square of side-length $2k$, is mapped to a square of side-length $k/2$ called the *patch region* shown in Figure 2.2; the patch region is contained within the skirt of the finished box. This is done in the simplest manner by mapping squares of side-length 4 of the patch to one node of the square in the finished box.

We now observe some properties of the mapping. A *ring* is a set of nodes that form the four sides of a square. The nodes in the finished box to which the border of the patch and the inner border of the outerskirt are mapped form rings in the finished box. We call these rings the *border rings*, or b-rings for short. The nodes on the border of the patch have duplicates in the interior of the outerskirt. Similarly, the nodes on the inner border of the outerskirt have duplicates in the interior of the patch. These duplicate nodes in the interior of the patch and the interior of the outerskirt are also mapped to rings in the finished box. We call these rings the *interior rings*, or i-rings for short.

In order for a node m in H to create an s-pebble for a node v in G, it must receive c-pebbles for each of the edges into v in G. If v is in the interior of the patch or the outerskirt, then the s-pebbles for the neighbors of v are created either by m or by the neighbors of m. In this case the required c-pebbles can be obtained in constant time. The s-pebbles for the neighbors of a node v on the border of the patch or outerskirt, however, may not be created near m in H. In our emulation, every node v on the border receives the c-pebbles for *all* of its incoming edges from its duplicate v' that is mapped to a node m' on one of the i-rings in the finished box. These c-pebbles are first created by the four neighbors of v', then sent to m' (in constant time), then forwarded on to m along a path that we call a *communication path*. Thus, for each node m' on an i-ring, we will need to route a communication path to its duplicate m on a b-ring. Note that these paths are determined off-line, before the start of the emulation. The skirt, which is fault-free, can be used as a crossbar to route the paths with dilation $O(k)$, and constant congestion. For the sake of brevity, the details are omitted.

We now describe the actual emulation. Each node m of H executes the following algorithm which proceeds as a sequence of macro-steps. Each macro-step consists of the following three sub-steps.

1. COMPUTATION STEP: For each node v of G that has been assigned to m, m creates a new s-pebble $\langle v, t\rangle$, provided that m has already created $\langle v, t-1\rangle$ and has received c-pebbles $[e, t-1]$ for every edge e into v.
2. COMMUNICATION STEP: For every node v such that s-pebble $\langle v, t\rangle$ was created by m in the computation step of the current macro-step, and for every edge e out of v, node m creates c-pebble $[e, t]$. If $[e, t]$ is needed by a neighbor m' of m, then m' sends $[e, t]$ to m'.
3. ROUTING STEP: If m lies on an i-ring, then m makes copies of any c-pebbles that were sent to m during the communication step of the current

macro-step. Then (whether or not m lies on an i-ring), for every c-pebble $[e, t]$ at m that has not yet reached its destination, m forwards it one step along its communication path.

LEMMA 2.2. *A macro-step takes a constant number of time steps.*

Proof. At each node of H, there are at most 17 s-pebbles to be updated in the computation step and hence this step takes constant time. Each s-pebble update can cause at most 4 c-pebbles (the outdegree of the node in G) to be sent. If the s-pebble is on the i-ring, it must send 4 additional c-pebbles to its duplicate on the b-ring. Thus the communication step takes only constant time. Since the paths used for routing have constant congestion and since a c-pebble in transit to its destination moves in every macro-step, there are at most a constant number of c-pebbles resident in a node at any time step that have not yet reached their destinations. Therefore, the routing step also takes constant time. □

THEOREM 2.3. *Any computation on a fault-free mesh G that takes time T can be emulated by the faulty mesh H with less than $(\log N)/2$ worst-case faults in $O(T+N)$ time steps.*

Proof. From Lemma 2.1, we know that since the number of worst-case faults in H is less than $(\log N)/2$, the growth process terminates with a set of non-intersecting finished boxes. The computation of G is mapped inside each of these finished boxes as described earlier in this section, and each node m of H performs the emulation algorithm. We will show that only $O(T + N)$ macro-steps are required to emulate a T-step computation of G. The theorem will then follow from Lemma 2.2.

The *dependency tree* of an s-pebble represents the functional dependency of this s-pebble on other s-pebbles and can be defined recursively as follows. As the base case, if $t = 0$, the dependency tree of $\langle v, t\rangle$ is a single node, $\langle v, 0\rangle$. If $t > 0$, the creation of s-pebble $\langle v, t\rangle$ requires an s-pebble $\langle v, t-1\rangle$ and all c-pebbles $[e, t-1]$ such that e is an incoming edge of node v in G. These c-pebbles are sent by some other s-pebbles $\langle u, t-1\rangle$ where u is a neighbor of v in G. The dependency tree of $\langle v, t\rangle$ is defined recursively as follows. The root of the tree is $\langle v, t\rangle$. The subtrees of this tree are the dependency trees of $\langle v, t-1\rangle$ and $\langle u, t-1\rangle$, for all s-pebbles $\langle u, t-1\rangle$ that send c-pebles to $\langle v, t\rangle$.

We now look at the dependency tree of the s-pebble that was created last. Let the emulation of T steps of G take T' time (in macro-steps) on H. Let $\langle v, T\rangle$ be an s-pebble that was updated in the last macro-step. For every tree node s, we can associate a time (in macro-steps) $\tau(s)$ when that s-pebble was created. We choose a *critical path*, $s_T, s_{T-1}, \cdots, s_0$, of tree nodes from the root to the leaves of the tree as follows. Let $s_T = \langle v, T\rangle$ be the root of the tree. s_T requires the s-pebble $\langle v, T-1\rangle$ and c-pebbles $[e, T-1]$. Let ϕ be the function that maps an s-pebble, $\langle v, t\rangle$ to the node in H that contains it. If the s-pebble $\langle v, T-1\rangle$ was created after all the c-pebbles were received then choose s_{T-1} to be $\langle v, T-1\rangle$. Otherwise, choose the s-pebble that sent the c-pebble that arrived last at node $\phi(\langle v, T\rangle)$ to be s_{T-1}. After choosing s_{T-1}, we choose the rest of the sequence recursively in the subtree with s_{T-1} as the root. We define a quantity l_i as follows. If $\phi(s_i)$ and $\phi(s_{i-1})$ are the same node or neighbors in H, then $l_i = 1$. Otherwise, l_i is the length of the path by which a c-pebble generated by s_{i-1} is sent to s_i. From the definition of our critical path, $\tau(s_i) - \tau(s_{i-1})$ equals l_i. This is because a c-pebble moves once along its communication path in every macro-step. Therefore

$$T' = \sum_{0<i\leq T} (\tau(s_i) - \tau(s_{i-1})) = \sum_{0<i\leq T} l_i.$$

Now suppose that some l_i is greater than 1. Then $\phi(s_i)$ must lie on the b-ring of a finished box, and some c-pebble must have taken a communication path of length l_i from a node m on an i-ring of the same finished box. In this case either $\phi(s_{i-1}) = m$ or $\phi(s_{i-1})$ is a neighbor of m. The key observation is that since $\phi(s_{i-1})$ is either on or next to an i-ring, in going down the critical path from s_{i-1} to s_0 we can encounter no more communication paths until we reach an s-pebble embedded in the b-ring, i.e., the values of $l_{i-1}, l_{i-2}, \ldots, l_{\max\{i-q,1\}}$ are all equal to 1, for some $q = \Theta(l_i)$. As $l_i = O(N)$, for all i, $T' = \sum_i l_i = O(T + N)$. □

It is possible to apply the construction described in this section recursively to show that, for any constant $k > 0$, an $N \times N$ mesh can sustain $\log^k N$ worst-case faults and still emulate a fault-free mesh with slowdown. Because a stronger result is proven in Section 3, the proof is omitted.

THEOREM 2.4. *For any constant $k > 0$, an $N \times N$ mesh with $\log^k N$ worst-case faults can emulate T steps of the computation of a fault-free $N \times N$ mesh in $O(T+N)$ steps. with constant slowdown.*

The construction described in this section assumes that large buffers are available at each node in the host to hold c-pebbles that reach their destinations early. Early arrivals can be prevented by slowing down the computation of some nodes and by finding communication paths with the property that all paths within a given finished box have the same length.

3. Multi-scale emulation.

In this section, we show that an $N \times N$ mesh, H, with any set of $N^{1-\epsilon}$ faults, for any fixed $\epsilon > 0$, can emulate any computation of a fault-free $N \times N$ mesh, G, with constant slowdown.

The major difference between the emulation scheme in this section and that in Section 2 is that in this section we allow a finished box to contain smaller finished boxes. In emulating the region of the guest mesh assigned to it, a finished box will in turn assign portions of this computation to each of the smaller boxes that it contains. These smaller boxes might in turn contain even smaller boxes and hence the term *multi-scale emulation*.

For simplicity, we assume that the mesh has wrap-around edges. This assumption can be easily done away with at the cost of considering some special cases for faults near the border of the mesh.

3.1. The growth process.

In this section, we show how to grow boxes on the faulty mesh H. There are two types of boxes: cores and finished boxes. A *core* is not capable of performing any portion of the emulation. A *finished box* consists of a core surrounded by a skirt. An (α-β)-*ensemble* is a collection of possibly intersecting finished boxes. Every finished box B in an (α-β)-ensemble has a distinct round number. The *intersecting region* of a finished box B in the ensemble is defined to be the region formed by nodes that lie both in the skirt of B and in some other finished box with a smaller round number than B. The boxes in an (α-β)-ensemble satisfy the following properties.

1. Every fault in the mesh H is contained in and assigned to the core of some finished box in the ensemble.
2. If the core of a finished box has side-length k, then the width of the skirt of the finished box is $\lfloor \alpha k \rfloor$.
3. The sum of the side-lengths of the finished boxes in the intersecting region of every finished box B is at most β times the width of the skirt of B.

The growth process produces an (α-β)-ensemble of boxes, where $0 < \alpha, \beta < 1$. It proceeds in rounds until there are no more cores left. Initially, every fault is enclosed in a core with side-length $\lceil 1/\alpha \rceil$. Each round produces either a new core or a new finished box. Each new finished box is numbered with the round in which it was created. At the beginning of each round, a core of the smallest side-length (say k) is selected and a skirt of width $\lfloor \alpha k \rfloor$ is grown around it to form a box (call this box B). If $k + 2\lfloor \alpha k \rfloor > N$ then the side-length of B will have to be larger than the size of the mesh itself and this is not possible. If this condition arises the growth process halts and is said to have failed. If this condition does not arise then one of the following steps is executed after which the growth process proceeds to the next round.

Expand Step If the sum of the side-lengths of the finished boxes in the intersecting region of B is more than β times the width of the skirt of B (i.e., $\beta\lfloor \alpha k \rfloor$), then we find the smallest bounding box that contains the core of B as well as the cores of all the finished boxes that intersect the skirt or core of B and turn this box into a new core. The finished boxes whose cores were included in this new core cease to exist, and their faults are assigned to the new core.

Create Step Otherwise, if the sum of the side-lengths of the finished boxes in the intersecting region of B is at most $\beta\lfloor \alpha k \rfloor$, then we declare box B to be a finished box.

Note that in the Expand step the intersecting region of B is computed using the collection of finished boxes that exist during that round.

LEMMA 3.1. *The growth process produces an (α-β)-ensemble of finished boxes, provided that it does not fail.*

Proof. We must show that all three of the properties of an (α-β)-ensemble are satisfied when the growth process does not fail. The growth process must terminate since at each round either the Expand step or the Create step is executed, each of which either increases the side-length of a core without changing the total number of cores or decreases the total number of cores by one. Property 1 is satisfied initially and since the Expand step forms a new core by enclosing a group of old cores, by induction this property will hold after every round. Property 2 is satisfied by construction. Finally, when a finished box B is created in the Create step, the sum of the side-lengths of the finished boxes in the intersecting region of B is at most β times the width of the skirt of B. New finished boxes created in later rounds do not affect this intersecting region since they all have greater round numbers than B. Some of the finished boxes with round numbers less than B may cease to exist due to the application of the Expand Step in some later rounds. However, this can only decrease the sum of the side-lengths of the finished boxes in the intersecting region of B. Thus, Property 3 will be true for all of the finished boxes when the growth process terminates. ∎

THEOREM 3.2. *For any fixed constants β and ϵ, where $0 < \beta < 1$ and $0 < \epsilon < 1$, there is a constant α, where $0 < \alpha < 1$, such that for sufficiently large N, for any set of $N^{1-\epsilon}$ faults in H the growth process grows an (α-β)-ensemble of finished boxes.*

Proof. We must show that for any fixed ϵ and β there is a constant α such that the growth process never fails, i.e., no core of side-length more than $N/(2\alpha + 1)$ is ever created. Then, by using Lemma 3.1, we can infer the theorem.

The key idea is to prove a lower bound, $F(k)$, on the number of faults that any core of side-length k must contain. Let $\delta = \epsilon/2$. We show by induction on k that $F(k) \geq Ak^{1-\delta}$, for some constant A. In order to satisfy the basis of the induction, we will choose A to be small enough that $F(k) \geq Ak^{1-\delta}$ for small values of k. Inductively, suppose that a new core of side-length k is formed in some round. Let x be the side-length of the core selected in this round and let $y_1, y_2, \cdots, y_m$ be the side-lengths of the other cores that were enclosed to form the new core. Since the new core contains these cores, we have $k > x$ and $k > y_i$, for $1 \leq i \leq m$. Since the number of faults in the new core is at least as large as the number of faults in the cores used to form it, using the inductive hypothesis we have

$$\begin{aligned} F(k) &\geq F(x) + F(y_1) + \cdots + F(y_m) \\ &\geq Ax^{1-\delta} + Ay_1^{1-\delta} + \cdots + Ay_m^{1-\delta}. \end{aligned} \tag{3.1}$$

Let y_1 and y_2 be the side-lengths of the two largest cores. If a new core was formed, then it must have been formed in the expand step. Thus, the side-length, k, of the new core is at most $(y_1 + y_2)(1+\alpha) + x(1+2\alpha)$. Thus, to prove the inductive hypothesis, it suffices to show that

$$x^{1-\delta} + \sum_{i=1}^{m} y_i^{1-\delta} \geq [(y_1 + y_2)(1+\alpha) + x(1+2\alpha)]^{1-\delta}. \tag{3.2}$$

Since the cores with side-lengths y_i belong to finished boxes created in earlier rounds, $y_i \leq x$, for all i. Furthermore, since a new core was created, $\sum_{i=1}^{m}(y_i + 2\lfloor \alpha y_i \rfloor) \geq \beta\lfloor \alpha x \rfloor$, which implies that $\sum_{i=1}^{m} y_i \geq \beta\alpha x/6$, for $0 < \alpha < 1$. Thus, there must be a largest index $j \geq 2$ such that $\beta\alpha x/6 \leq \sum_{i=1}^{j} y_i \leq 2x$. Let $y = \sum_{i=1}^{j} y_i$. Then $y \geq y_1 + y_2$. Also, because of the convexity of the function $f(z) = z^{1-\delta}$, $y^{1-\delta} \leq \sum_{i=1}^{j} y_i^{1-\delta} \leq \sum_{i=1}^{m} y_i^{1-\delta}$. Thus, Inequality 3.2 must hold if

$$[y(1+\alpha) + x(1+2\alpha)]^{1-\delta} \leq x^{1-\delta} + y^{1-\delta} \tag{3.3}$$

holds for all y such that $\beta\alpha x/6 \leq y \leq 2x$.

Proving that Inequality 3.3 holds for sufficiently small α requires some elementary (but painstaking) calculations. Let $\lambda = y/x$. In terms of λ, we need to show that for sufficiently small α, the inequality

$$[(1+\alpha)\lambda + (1+2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} \leq 0 \tag{3.4}$$

holds for all λ such that $\beta\alpha/6x \leq \lambda \leq 2$. (As we shall see, $\alpha \leq \min\{(\beta/24)^{1/\delta}, \delta/16\}$ suffices.) There are three cases to consider.

First, suppose that $\lambda \leq \alpha$. In this case, we have

$$\begin{aligned} [(1+\alpha)\lambda + (1+2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} &\leq (1+4\alpha)^{1-\delta} - 1 - (\beta\alpha/6)^{1-\delta} && (3.5) \\ &\leq 4\alpha - \left(\frac{(\beta/6)^{1-\delta}}{\alpha^{\delta}}\right)\alpha && (3.6) \\ &\leq \left(4 - \frac{\beta}{6\alpha^{\delta}}\right). && (3.7) \end{aligned}$$

Line 3.5 is derived using the inequalities $(1+\alpha) \leq 2$, $\lambda \leq \alpha$, and $\beta\alpha/6 \leq \lambda$. Line 3.6 is derived from Line 3.5 using the inequality $(1+4\alpha)^{1-\delta} \leq (1+4\alpha)$. Line 3.7 is derived from Line 3.6 using the inequalities $\alpha < 1$ and $\beta/6 \leq (\beta/6)^{1-\delta}$. For $\alpha \leq (\beta/24)^{1/\delta}$, the right-hand side of Line 3.7 is at most 0.

Second, suppose that $\alpha \leq \lambda \leq 1$. In this case we have

$$\begin{aligned} [(1+\alpha)\lambda + (1+2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} &\leq [1 + \lambda + 3\alpha]^{1-\delta} - 1 - \lambda^{1-\delta} && (3.8) \\ &\leq (1-\delta)(\lambda + 3\alpha) - \lambda^{1-\delta} && (3.9) \\ &\leq 3\alpha + \lambda - \delta\lambda - \lambda^{1-\delta}. && (3.10) \end{aligned}$$

Line 3.8 is derived using the inequality $\lambda \le 1$. Line 3.9 is derived from the Line 3.8 using the fact that $(1+a)^b \le 1+ab$ for any $a \ge 0$ and $0 \le b \le 1$. Line 3.10 is derived from Line 3.9 using the inequality $3\alpha\delta > 0$. Now let $h(\lambda) = 3\alpha + \lambda - \delta\lambda - \lambda^{1-\delta}$. Differentiating with respect to λ, we have $h'(\lambda) = 1 - \delta - (1-\delta)\lambda^{-\delta}$. For $0 < \lambda < 1$, $h'(\lambda) < 0$, and for $\lambda = 1$, $h'(\lambda) = 0$. Thus for $\alpha \le \lambda \le 1$, $h(\lambda)$ takes on its maximum value when $\lambda = \alpha$. Returning to Lines 3.8 through 3.10, we have

$$\begin{aligned}
&(3.11) \qquad [(1+\alpha)\lambda + (1+2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} &\le\ & 4\alpha - \delta\alpha - \alpha^{1-\delta} \\
&(3.12) &\le\ & 4 - \alpha^{-\delta}.
\end{aligned}$$

Line 3.12 is derived from Line 3.11 using the inequalities $\delta > 0$ and $\alpha < 1$. The right-hand side of Line 3.12 is at most 0, provided that $\alpha \le (\beta/24)^{1/\delta}$, as in the first case.

Finally, suppose that $1 < \lambda \le 2$. In this case, we have

$$\begin{aligned}
&(3.13) \quad [(1+\alpha)\lambda+(1+2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} &\le\ & [(1+\lambda)+4\alpha]^{1-\delta} - 1 - \lambda^{1-\delta} \\
&(3.14) &\le\ & (1+\lambda)^{1-\delta}(1+4\alpha)^{1-\delta} - 1 - \lambda^{1-\delta} \\
&(3.15) &\le\ & \lambda^{1-\delta}\left(1+\frac{1-\delta}{\lambda}\right)(1+4\alpha(1-\delta)) - 1 - \lambda^{1-\delta} \\
&(3.16) &=\ & \frac{1-\delta}{\lambda^{\delta}} + \lambda^{1-\delta}\left(1+\frac{1-\delta}{\lambda}\right)(4\alpha(1-\delta)) - 1 \\
&(3.17) &\le\ & 16\alpha - \delta.
\end{aligned}$$

Line 3.13 is derived using the inequality $\lambda \le 2$. Line 3.14 is derived from Line 3.13 using the inequality $1/(1+\lambda) \le 1$. Line 3.15 is derived from Line 3.14 using the fact that $(1+a)^b \le 1+ab$ for any $a \ge 0$ and $0 \le b \le 1$ (in two places). Line 3.17 is derived from Line 3.16 using the inequalities $1/\lambda^{\delta} < 1$, $\lambda^{1-\delta} < 2$, $(1-\delta)/\lambda < 1$, and $1-\delta < 1$. For $\alpha < \delta/16$ the right-hand side of Line 3.17 is at most 0.

The growth process fails only if a core of side-length k is created where $k+2\lceil \alpha k \rceil > N$. In this case $k > (N-2)/(2\alpha+1) > N/(2\alpha+3)$, for sufficiently large N. Such a core must contain at least $F(k) \ge Ak^{1-\delta} \ge A(N/(2\alpha+3))^{1-\delta}$ faults. Recall that $\delta = \epsilon/2$. Thus, for sufficiently large N, $A(N/(2\alpha+3))^{1-\delta} > N^{1-\epsilon}$. So, if there are fewer than $N^{1-\epsilon}$ faults in the mesh, then no such core is created, and the growth process produces an $(\alpha$-$\beta)$-ensemble of finished boxes. $\square$

3.2. Mapping the computation. In this section, we show how to map the computation of the fault-free $N \times N$ mesh G to the faulty $N \times N$ mesh H. The mapping requires that an $(\alpha$-$\beta)$-ensemble of finished boxes be grown in H, for some constants α and β, where $0 < \alpha < 1$ and $0 < \beta < 1$. As it turns out, β can be chosen independently of α and ϵ. So we choose β first. Next, we choose α such that for fixed β and ϵ and any set of $N^{1-\epsilon}$ faults an $(\alpha$-$\beta)$-ensemble of boxes can be grown using the growth process outlined in Section 3.1.

We will use the pebbling terminology introduced in Section 2 to describe the mapping. The mapping is produced by a *mapping process* that progresses iteratively in rounds. Initially, an s-pebble for each node of G is mapped to the corresponding node of H. Like a regular mesh computation, each s-pebble gets c-pebbles from the s-pebbles mapped to the neighboring nodes in H. The mapping process selects a finished box in the ensemble at the beginning of each round in the decreasing order of their round numbers. In each round, the mapping process changes the mapping inside the selected finished box so that no s-pebbles are mapped to the core of that finished box. This is done by removing s-pebbles from the core, duplicating some s-pebbles, and setting up constant-congestion communication paths between duplicated s-pebbles that avoid routing through any finished boxes with smaller round numbers. The mapping process terminates when all finished boxes in the ensemble have been selected.

A mapping of the computation of G to H is said to be *valid* if no s-pebble of G is mapped to a faulty node in H and no communication path between two s-pebbles passes through a faulty node in H. The initial mapping is *not* valid since it maps s-pebbles to faulty nodes of H. After all the rounds are completed, no s-pebble will be mapped to a faulty node and no communication will pass though a faulty node. Thus, the final mapping will be valid.

The computation mapped onto a finished box B is said to be *mesh-like* if each node in B has exactly one s-pebble mapped to it and each s-pebble that is mapped to a node m in B receives a c-pebble from each of the s-pebbles mapped to neighboring nodes of m in B. Our mapping process will ensure that the following invariant will hold true at the beginning of every round.

INVARIANT 3.3. *Suppose that B is a finished box with round number l that is selected at some round. At the beginning of the round, for every finished box B' with round number k, $k \leq l$, either no computation is mapped to B', or a mesh-like computation is mapped to B'. Furthermore, no communication path passes through any node in B'.* The invariant is true at the beginning of the first round since every finished box has a mesh-like computation mapped to it and there are no communication paths. At the end of each round, this invariant will hold true inductively. Later in this section, we will outline the steps involved in a specific round of the mapping process in which the computation within the chosen finished box is remapped. We now show that the invariant guarantees that upon termination the mapping process produces a mapping that does not map computation or communication to faulty processors.

THEOREM 3.4. *The mapping process produces a valid mapping of the computation of G into H.*

Proof. We must show that every faulty node of H has neither an s-pebble mapped to it nor a communication path passing through it in the final mapping produced by the iterative mapping process. A node v of H is said to be *active* at a particular round of the mapping process if it either has an s-pebble mapped to it or has a communication path passing through it in the beginning of this round. A node is said to be *inactive* if it is not active. In the first round, every node in H is active. A key property of the iterative mapping process is that if in some round a node v becomes inactive it remains inactive through the remaining rounds. For a contradiction, suppose that an inactive node v becomes active. Let B be the finished box selected in the last round in which v was inactive. Since only nodes inside B are affected by the remapping, v must be in B. From Invariant 3.3 and the fact that v is inactive, it must be the case that no computation was mapped to B. This means that no computation was remapped in this round, which is a contradiction.

We now show that no faulty node remains active at the end of the mapping proccess. From Property 1 of an $(\alpha\text{-}\beta)$-ensemble of finished boxes, every fault in H is contained in some core of some finished box. Let v be a faulty node in H and let B be the finished box whose core contains this fault. If v is already inactive in some round before B is selected, it will remain inactive through the rest of the rounds. Otherwise, if v is active in the round that B is selected, it follows from Invariant 3.3 that there must be an s-pebble mapped to v but no communication paths passing through v at the beginning of this round. The remapping of computation inside B will remove the computation from v and no new communication path will pass through v. Therefore v becomes inactive and remains that way through the rest of the mapping process. Thus no faulty node remains active at the end of the mapping process. □

3.2.1. Remapping the computation within a finished box. In this section, we show how to remap the computation within a finished box chosen in some round of the iterative mapping process. Let B be a box with round number l and side-length $(2\alpha + 1)k$ that is selected at some round of the mapping process. (In the remainder of this paper we ignore the issue of whether quantities such as $2\alpha k$ are integral.) We assume that Invariant 3.3 is true at the beginning of this round. Later we show that this invariant is true at the end of the round after the remapping. If there is no computation mapped to B at the beginning of the round, no remapping needs to be done and the invariant holds at the end of the round.

The other possibility is that a mesh-like computation is mapped to the nodes of B at the beginning of this round. In this case, the computation is partitioned into two overlapping pieces, the patch and the outerskirt. The precise sizes of the patch and the outerskirt will be specified later. The set of nodes in the finished box to which the border of the patch is mapped forms a ring in the finished box, as does the set of nodes to which the inner border of the outerskirt is mapped. We call these rings the border rings, or b-rings for short. Because the patch and the outerskirt overlap, the nodes on the border of the patch have duplicates in the interior of the outerskirt that perform the same computation. Similarly, the nodes in the inner border of the outerskirt have duplicates in the interior of the patch. These duplicate nodes in the interior of the outskirt and the interior of the patch are also mapped to rings in the finished box. We call these rings the interior rings, or i-rings for short.

The region consisting of nodes in B not more than distance $\alpha k/5$ from the outer border of B is called the *outer region*. The outerskirt will be embedded in this region. Similarly a square box of side-length $3\alpha k/5$ is called the *patch region*. As shown in Figure 3.1, the patch region is located below and to the left of the core, and does not intersect the outer region. The patch will be embedded in this region.

The first step in remapping the computation within B is to place a b-ring and an i-ring in each of the two regions (See Figure 3.1). A *free ring* in the patch region or the outer region is defined to be a ring that does not pass through any finished boxes B' with smaller round numbers than B. The b- and i-rings satisfy the following *ring properties*.

1. The i-ring and the b-ring of the outer region must be free rings. Furthermore, between the i-ring and the b-ring of the outer region there must be $\Theta(k)$ free rings. The same condition must hold for the i-ring and the b-ring of the patch region. Further, the i-ring of the patch region must have side-length $\Theta(k)$.
2. For *any* constant-load embedding of s-pebbles into a side of the i-ring of one region and *any* constant-load embedding of the duplicates of these s-pebbles into the corresponding side of the b-ring of the other region, there must be paths of length $\Theta(k)$ from every s-pebble to its duplicate. These paths must have constant congestion, must be completely contained in B, and must not pass through any finished boxes B' with smaller round numbers than B.

The procedure for finding rings with these properties is outlined in Section 3.4.

Having determined the i-rings and the b-rings, the next step is to determine the size and layout of the patch and the outerskirt. Recall that a mesh-like computation was mapped into box B at the beginning of this round. The size of the outerskirt

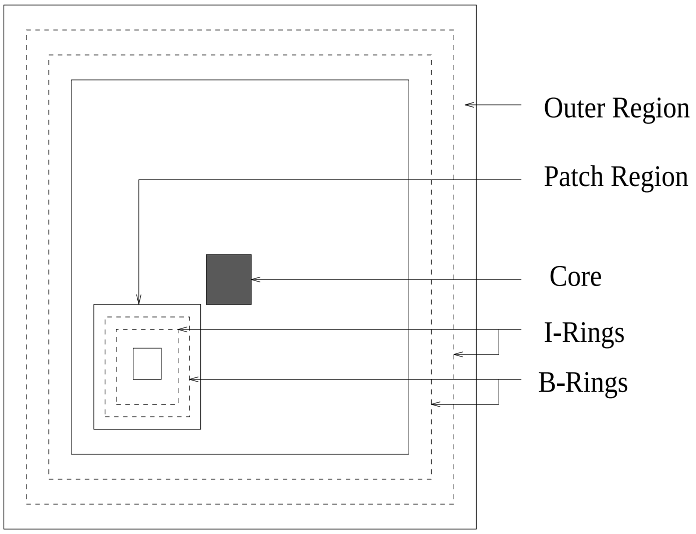

FIG. 3.1. *Layout of finished box B in the host.*

and patch will depend on the choice of the i- and b-rings. The computation mapped between the b-ring of the outer region and the border of B at the beginning of this round forms the outerskirt. The computation mapped within the i-ring of the outer region at the beginning of this round forms the patch. Remapping the outerskirt and the patch to nodes within B must be done with care so that Invariant 3.3 is true at the end of this round.

The outerskirt is embedded in the region of B between the b-ring in the outer region and the border of B. Since the size and shape of the outerskirt is the same as the region in which it is embedded, we simply map each s-pebble in the outerskirt to the corresponding node in that region of B.

The patch must be embedded into the square region enclosed by the b-ring in the patch region. This is trickier since the patch is a constant factor larger in size than the square region in which it is embedded. In particular, we must ensure that each finished box B' with a smaller round number than B that intersects this square region receives a mesh-like computation or receives no computation at all. A column or row in this square region that intersects such a finished box B' will be called a *bad column* or a *bad row*. The remaining rows and columns are said to be *good rows* and *good columns* respectively. Since the sum of the side-lengths of such boxes B' is at most $\beta\alpha k$, if we choose $\beta < 1/10$, then a majority of the $\alpha k/5$ columns and rows will be good. Our embedding is described by two functions ρ and κ such that a node in the i^{th} row and the j^{th} column of the patch is mapped to the $\rho(i)^{th}$ row and the $\kappa(j)^{th}$ column of the square region. The function ρ is selected so that for all i, $\rho(i) \leq \rho(i+1) \leq \rho(i)+1$. Further, for any value of j there are at most a constant number of values of i with $\rho(i) = j$ and if the j^{th} row is bad there is exactly one value of i with $\rho(i) = j$. The function κ is chosen with similar properties for the columns. That such functions ρ and κ exist follows from the fact that the patch is at most a constant factor larger than the square region and that a majority of the rows and columns of the square region are good. This completes the embedding of the s-pebbles to nodes within B.

Finally, the constant-congestion paths between the s-pebbles in the i-ring and their duplicates in the b-ring are set up. These paths must not pass through any finished boxes with round numbers smaller than that of B. These paths can be set up since the i-ring and b-ring satisfy the second ring property. Using these paths, each s-pebble on the b-ring receives c-pebbles from its duplicate on the i-ring.

Now we show inductively that Invariant 3.3 holds at the end of the round in which B was selected.

THEOREM 3.5. *Invariant 3.3 is true after the computation within B has been remapped.*

Proof. We must show that each finished box B' with a smaller round number than B has a mesh-like computation mapped to it or no computation mapped to it at all. All such boxes B' that do not intersect B are not affected by the remapping at all. From ring property 1, we know that no box B' with a smaller round number than B can intersect the b-ring in the outer region. Therefore any intersecting box B' not entirely contained in B must intersect the region only where the outerskirt is embedded. Since the embedding of this region does not change in the course of the remapping, all such boxes B' still have a mesh-like computation mapped to them.

We will now look at boxes B' contained entirely within B. Since the b-rings and i-rings do not pass through B', either B' is contained entirely in the region where the outerskirt is embedded or entirely in the region between the b-ring of the outer region and the core of the box or entirely inside the square region where the patch is embedded. In the first case, the embedding inside B' does not change by the remapping and it continues to have a mesh-like computation mapped to it. In the second case, no computation is mapped to B' and no communication path passes though the nodes in it. In the third case, observe that every row or column of B' is in a bad row or bad column of the square region. Thus ρ and κ map exactly one row and one column respectively of the patch to these rows and columns. Thus a mesh-like computation is mapped to B'. Finally, none of the newly formed communication paths pass through any of the finished boxes B'. □

3.3. The emulation. In this section, we show that if the growth process terminates with a set of non-intersecting finished boxes, then the host H can emulate the guest G with constant slowdown.

In order for a node in H to create an s-pebble for a node v of G, it must receive c-pebbles for each of the edges into v in G. If this s-pebble is in the interior of a patch or an outerskirt, the s-pebbles for the neighbors of v are created either by the same node in H or by the neighbors of that node in H. Thus the required c-pebbles can be obtained in constant time. However, the s-pebbles for the neighbors of an s-pebble on the border of the patch or on the inner border of the outerskirt may not be created nearby in H. But, since the patch and the outerskirt overlap, for every s-pebble on the border of the patch or the inner border of the outerskirt there is a duplicate in the interior of the outerskirt or the patch, respectively. In our emulation, every s-pebble on the border will receive the c-pebbles for all of its incoming edges from its duplicate that is mapped to one of the i-rings in the finished box.

The emulation consists of a series of macro-steps as in Section 2.2.

LEMMA 3.6. *Each macro-step takes only a constant number of time steps to execute.*

Proof. We will prove that the maximum number of s-pebbles mapped to any node of H and the maximum number of communication paths passing through any node of H is a constant when the mapping process terminates. We prove this by induction on the rounds of the mapping process. At the beginning of the first round of the mapping process there is exactly one s-pebble mapped to every node of H and no communication paths pass through any node. So the hypothesis is true at the beginning of the first round. Suppose the hypothesis is true at the beginning of some round. Let the finished box selected at this round be B. There are two possibilities. If there is no computation mapped to B and no communication passing through it no remapping is done and the hypothesis remains true at the beginning of the next round. Otherwise, from Invariant 3.3, there is a mesh-like computation mapped to B and no communication path passes through any of its nodes. Remapping the computation within each finished box B causes at most a constant number of s-pebbles to be mapped to any node within it. Furthermore, the maximum number of paths created in this round that pass through a node in B is a constant. Since no communication path created before this round uses a node in B, the inductive hypothesis is true at the beginning of the next round.

Since there are only a constant number of s-pebbles mapped to any node of H, the computation step takes only constant time. Since each s-pebble can produce at most 4 c-pebbles in the communication step, there are at most a constant number of c-pebbles created at each step by each node. Thus the communication step takes only constant time. Since there are only a constant number of paths passing through every node and since every c-pebble moves in every macro-step and only a constant number of c-pebbles enter a particular path at any macro-step, there can be only a constant number of c-pebbles on a particular path resident at a particular node at a particular time. Thus the routing step also takes only a constant number of time steps. $\square$

THEOREM 3.7. *For sufficiently large N, any computation on an $N \times N$ fault-free mesh G that takes time T can be emulated by an $N \times N$ faulty mesh H with $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$) in time $O(T + N)$.*

Proof. We show that only $O(T + N)$ macro-steps are required to emulate any T-step computation of G. The final result then follows from Lemma 3.6.

Let $B_1, B_2, \cdots, B_m$ be the finished boxes in the descending order of their round numbers and let their side-lengths be $k_1, k_2, \cdots, k_m$. The iterative mapping process produces a series of mappings, $\phi_0, \phi_1, \cdots, \phi_m$, where ϕ_i is the mapping of s-pebbles to H at the end of the i^{th} round. The mapping ϕ_i is obtained from the mapping ϕ_{i-1} by remapping the computation within box B_i. The final mapping generated by the mapping process is ϕ_m. Note that because the guest network can be redundant, it is possible for two distinct s-pebbles s and s' to have the same label $\langle v, t\rangle$, i.e., v is the node of G whose state after t steps of computation is represented by both s-pebbles s and s'. Furthermore, it is possible for different mappings ϕ_i and ϕ_j, $i \neq j$, to map different numbers of s-pebbles to H. For example, for each node v in G and each time step t, ϕ_0 maps only one s-pebble to H. For $i > 0$, however, unless there are no faults in H, ϕ_i maps at least two s-pebbles s and s' with the same label to different nodes in H.

The dependency tree and critical path for the final mapping ϕ_m are defined as in Section 2.2. In general, a sequence of s-pebbles $s_{i,T}, s_{i,T-1}, \ldots, s_{i,0}$ is called a *T-sequence* with respect to a mapping ϕ_i if, for $0 < j < T-1$, $t_{i,j+1} = t_{i,j} + 1$, and either $v_{i,j}$ and $v_{i,j+1}$ are the same node of G, or $v_{i,j}$ and $v_{i,j+1}$ are neighbors in G and under ϕ_i, $s_{i,j}$ sends a c-pebble to $s_{i,j+1}$. For a given mapping ϕ_i and a T-sequence $s_{i,T}, s_{i,T-1}, \ldots, s_{i,0}$, $l_{i,j}$ is 1 if nodes $\phi(s_{i,j})$ are the same node or neighbors in H. Otherwise, $l_{i,j}$ is the length of the communication path by which a c-pebble generated by $s_{i,j-1}$ is sent to $s_{i,j}$. With each mapping ϕ_i we associate one T-sequence. The T-sequence for ϕ_m is the critical path. For $i < m$, the T-sequence for ϕ_i is derived from the T-sequence for ϕ_{i+1}. Let $\langle v_{i+1,j}, t_{i+1,j} \rangle$ be the label of $s_{i+1,j}$. If ϕ_{i+1} does not map $s_{i+1,j}$ to the border of the patch or the outerskirt of box B_{i+1}, then both ϕ_{i+1} and ϕ_i map a single pebble labeled $\langle v_{i+1,j}, t_{i+1,j} \rangle$ to H. In this case, $s_{i,j}$ is the pebble with label $\langle v_{i+1,j}, t_{i+1,j} \rangle$ that ϕ_i maps to H. Otherwise, ϕ_{i+1} maps two duplicate s-pebbles, both labeled $\langle v_{i+1,j}, t_{i+1,j} \rangle$, to box B_{i+1}, but ϕ_i maps only one pebble with that label to B_{i+1}. In this case, $s_{i,j}$ is the one s-pebble with label $\langle v_{i+1,j}, t_{i+1,j} \rangle$ that ϕ_i maps to B_{i+1}.

We now show that for any T-sequence $s_{m,T}, s_{m,T-1}, \cdots, s_{m,0}$ with respect to ϕ_m, $\sum_{0<j\leq T} l_{m,j}$ is $O(T+N)$. For each mapping ϕ_i, we define a series of weights c_i, d_i, and $w_{i,j}$, where $0 \leq i \leq m$ and $0 < j \leq T$. The weights are chosen such that for any value i the following two properties are satisfied.

1. $\sum_j l_{i,j} \leq \sum_j w_{i,j} + \sum_{r \leq i}(c_r + d_r)$.
2. For all j, $w_{i,j}$ is less than some fixed constant.

We will find an upper bound on $\sum_j l_{m,j}$, and hence on T', by finding upper bounds on the $w_{m,j}$ (they will be constant) and on $\sum_{r \leq m}(c_r + d_r)$ (which will be $O(N)$).

Initially, we define $c_0 = d_0 = 0$, and $w_{0,j} = l_{0,j}$, for all values of j. Since ϕ_0 simply maps the s-pebbles of G to the corresponding node of H, every $l_{0,j}$ and hence every $w_{0,j}$ is 1. Thus $\sum_j l_{0,j} = \sum_j w_{0,j} + c_0 + d_0$. Furthermore, for all values of j, $w_{0,j}$ can be bounded from above by a fixed constant.

We choose c_i, d_i, and $w_{i,j}$, $0 < j \leq T$ as follows. Inductively assume that we have determined the weights c_r, d_r, $0 \leq r \leq i-1$, and $w_{i-1,j}$, $0 < j \leq T$, such that the two properties listed above are satisfied. Suppose that ϕ_{i-1} maps no computation onto box B_i. Then no remapping is necessary and so $l_{i,j} = l_{i-1,j}$, for all values of j. In this case we set $w_{i,j} = w_{i-1,j}$, for all j, and set $c_i = d_i = 0$. Otherwise, from Invariant 3.3, ϕ_{i-1} maps a mesh-like computation onto box B_i. The mapping ϕ_i differs from ϕ_{i-1} in that s-pebbles inside box B_i are remapped. Since s-pebbles outside B_i are not affected, $l_{i,j} = l_{i-1,j}$ for all j such that $\phi_{i-1}(s_{i-1,j})$ is not in B_i. We define $w_{i,j} = w_{i-1,j}$ for all such values of j. (Note that if ϕ_{i-1} maps an s-pebble $s_{i-1,j}$ outside B_i but adjacent to its border and maps $s_{i-1,j-1}$ to the border of B_i, then $l_{i,j} = l_{i-1,j}$. This is because the remapping inside B_i will not the change the location of the pebble with the same label as $s_{i-1,j-1}$.)

We now look at s-pebbles mapped inside B_i by ϕ_{i-1}. The new mapping ϕ_i introduces communication paths for s-pebbles $s_{i,j}$ such that $\phi_{i-1}(s_{i-1,j})$ lies on a b-ring of B_i. For all such s-pebbles $s_{i,j}$, $l_{i,j}$ equals the length of the communication path in B_i which is $\Theta(k_i)$, where k_i is the side-length of B_i. For all other s-pebbles $s_{i,j}$, $l_{i,j} = l_{i-1,j}$. We determine the weights $w_{i,j}$ for each s-pebble $s_{i,j}$ such that $\phi_{i-1}(s_{i-1,j})$ is in B_i as follows. We will consider every maximal subsequence, $s_{i-1,h+p}, s_{i-1,h+p-1}, \cdots, s_{i-1,h}$, of the T-sequence such that $\phi_{i-1}(s_{i-1,h+q})$ is in B_i, for $0 \leq q \leq p$. Let I be a set of integers q such that $l_{i,h+q} > 1$. There are 3 cases depending on the value of $|I|$.

If $|I| = 0$ there are no communication paths and for every q such that $0 \leq q \leq p$, $l_{i,h+q} = l_{i-1,h+q} = 1$. Therefore we will define $w_{i,h+q} = w_{i-1,h+q}$ for every $0 \leq q \leq p$, and set $c_i = d_i = 0$.

If $|I| \neq 0$, let $L = \sum_{q \in I}(l_{i,h+q} - l_{i-1,h+q})$, which equals the net increase in the values of $l_{i,h+q}$ in the subsequence.

If $|I| = 1$, there is exactly one communication path. Note that this can happen only if either $\phi_{i-1}(s_{i-1,0})$ or $\phi_{i-1}(s_{i-1,T})$ is in box B_i. This is so because a maximal subsequence of s-pebbles mapped to B_i that contains neither $s_{i-1,0}$ nor $s_{i-1,T}$ must necessarily begin and terminate with s-pebbles mapped to the border of B_i. Therefore such a subsequence must neccessarily use communication paths an even number of times. If $\phi_{i-1}(s_{i-1,0})$ is in B_i make $c_i = L$, where $L = \Theta(k_i)$ (since there is only one path). Otherwise set $c_i = 0$. Similarly if $\phi_{i-1}(s_{i-1,T})$ is in B_i make $d_i = L$. Otherwise set $d_i = 0$. For every $0 \leq q \leq p$, set $w_{i,h+q} = w_{i-1,h+q}$.

If $|I| > 1$, let J be the set of integers q such that $\phi_i(s_{i-1,h+q})$ is in some free ring either in the patch region or in the outer region. Recall that nodes in the free rings are not contained in any finished box B_l, $l > i$. The value of L is $|I|\Theta(k_i)$ since each communication path in B_i is $\Theta(k_i)$ in length. This increase must be distributed evenly among the weights of the s-pebbles $s_{i,h+q}, q \in J$. Thus for all $q \in J$, $w_{i,h+q} = w_{i-1,h+q} + L/|J|$. For any two s-pebbles $s_{i-1,h+q_1}$ and $s_{i-1,h+q_2}$ such that $q_1 < q_2$ and $q_1, q_2 \in I$, the subsequence $s_{i-1,h+q_1}, \cdots, s_{i-1,h+q_2}$ contains at least $\Theta(k_i)$ s-pebbles $s_{i-1,h+q'}$, such that $q' \in J$. This is so because the i-ring and the b-ring of the patch region or the outer region were chosen such that there are $\Theta(k_i)$ free rings between them. This implies that $|J| = \Theta(k_i|I|)$ and thus $L/|J|$ is a constant. For all other $q \notin |J|$, $l_{i,h+q} = l_{i-1,h+q}$ and $w_{i,h+q} = w_{i-1,h+q}$. We also set $c_i = d_i = 0$. After all such subsequences have been dealt with we go to the next iteration.

The weight assignments in all three cases maintain the condition that $\sum_j l_{i,j} \leq \sum_j w_{i,j} + \sum_{r \leq i}(c_r + d_r)$. Further, for all i and j, $w_{i,j}$ is at most a constant. This is so because if the weight of some $s_{i-1,j}$ increases at the i^{th} iteration, i.e., $w_{i,j} > w_{i-1,j}$, then it will never increase again since $\phi_{i-1}(s_{i-1,j})$ is in a free ring of the finished box selected in the i^{th} iteration and hence is not contained in any of the finished boxes with smaller round numbers that will be considered in future rounds. Thus its weight will never change after this iteration. Further, as we saw earlier, the increment $w_{i,j} - w_{i-1,j}$ is also a constant.

We bound $\sum_j l_{m,j}$ by bounding $\sum_j w_{m,j}$ and $\sum_{i \leq m}(c_i + d_i)$. The fact that $w_{m,j}$ is a constant for all j implies that $\sum_j w_{m,j}$ is $O(T)$. We bound the summation $\sum_i (c_i + d_i)$ as follows. The value of c_i or d_i is either zero or $\Theta(k_i)$. Thus $\sum_i c_i$ can be no more than the sum of the side-lengths of all the boxes in the (α-β)-ensemble, i.e., $\sum_i k_i$. We show that this quantity is $O(N)$. By the proof of Theorem 3.2 we know that the core of each B_i has at least $Ak_i^{1-\epsilon/2} > Ak_i^{1-\epsilon}$ faults, where A is a constant. Since each fault is contained in a unique core and there are at most $N^{1-\epsilon}$ faults in the mesh,

$$\sum_i Ak_i^{1-\epsilon} \leq N^{1-\epsilon}.$$

The maximum value of $\sum_i k_i$ that satisfies the above constraint occurs when all but one of the values of k_i equal zero, i.e., when one value of k_i is $\Theta(N)$ and the rest are zero. Thus $\sum_i k_i$ and hence $\sum_i c_i$ is $O(N)$. Similarly, $\sum_i d_i$ can be shown to be $O(N)$. Therefore,

$$\sum_j l_{m,j} \leq \sum_j w_{m,j} + \sum_{i \leq m}(c_i + d_i) = O(T + N)$$

$\square$

3.4. Finding the i- and b-rings. In this section, we show how to find i- and b-rings in a finished box B in H with side-length $(2\alpha + 1)k$ satisfying the two ring properties listed in Section 3.2.

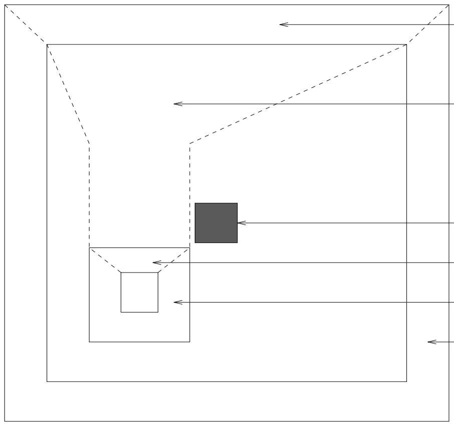

FIG. 3.2. *Finding i- and b-rings in H.*

Recall that the patch region is a square region in B of side-length $3\alpha k/5$ and the outer region is an annular region of width $\alpha k/5$. In the center of the patch region, we place a square of size $\alpha k/5$. (See Figure 3.2). The i-ring of the patch is required to enclose this square. This guarantees that the i-ring of the patch has size $\Theta(k)$. A *trapezoid* is a four-sided figure consisting of two parallel sides and two non-parallel sides. We define the i^{th} column of a trapezoid to be the set of nodes in the trapezoid at a distance i from the longer parallel side of the trapezoid. By joining the corners of the square in the patch region to the respective corners of the patch region, we partition the patch region, excluding the area enclosed by the square, into four trapezoidal regions (One of these regions is shown in Figure 3.2). Similarly, the outer region is also partitioned into four trapezoidal regions by joining each corner of the box B to the corresponding corner of the square forming the inner boundary of the outer region. Each ring in the outer or patch region consists of four sides and each side is a column of one of the trapezoids.

We will define four distinct *zones*, each of which is made up of three parts (A zone is marked with dotted lines in Figure 3.2.) The first part of a zone consists of one of the four trapezoids in the outer region (called the *outer trapezoid*) and the last part consists of the corresponding trapezoid in the patch region (called the *patch trapezoid*). The middle part is called the *fitting* and joins the outer trapezoid to the patch trapezoid (See Figure 3.2). The fitting is either a trapezoidal region or a rectangular region adjoining a trapezoidal region (The patch region is positioned below and to the left of the core so that this is true.). Each of the four sides of a ring in the patch region or the outer region is a trapezoidal column in one of the four zones. Choosing b- and i-rings is equivalent to finding two trapezoidal columns in the patch trapezoid and two trapezoidal columns in the outer trapezoid of each of the four zones. Further, the four trapezoidal columns, one in each zone, that correspond to a particular ring must be chosen so as to have the same column number.

Since it is easier to work with rectangular grids than trapezoids, we will embed each of the four zones into a single rectangular grid R with $\alpha k/5$ rows and $3\alpha k/5$ columns. Note that R is not part of the guest or the host. It is a tool of the construction only. The grid formed by the first $\alpha k/5$ columns of R is called the *outer grid*, the next $\alpha k/5$ columns the *fitting grid*, and the last $\alpha k/5$ columns the *patch grid*. The outer trapezoid, the fitting, and the patch trapezoid of each zone are embedded into the outer grid, fitting grid and patch grid respectively.

Since the outer trapezoid and the patch trapezoid each have $\alpha k/5$ columns, embedding them into their respective grids can be done by embedding the nodes in the i^{th} trapezoidal column of the outer or the patch trapezoid to nodes in the i^{th} column of the respective grid. Any constant-load and constant-dilation embedding will do for our purposes. We describe such an embedding below. The nodes in each trapezoidal column are grouped into $\alpha k/5$ groups such that each group contains some constant number of consecutive nodes of the column. Further, the cardinality of any two groups in a column differ by at most one and for all $i < j$ the cardinality of the i^{th} group is at most the cardinality of the j^{th} group of the same column. For every trapezoidal column, the i^{th} such group is mapped to the i^{th} node of the corresponding column of the grid. The dilation of this embedding is at most 2 and the load is constant.

Note that the four sides that form a ring either in the outer region or in the patch region are embedded into the same column in the outer grid or the patch grid respectively. Therefore a column in the outer or patch grid corresponds to the ring in the outer or patch region that gets mapped to it.

We can use the above technique to embed the fitting as well. The only difference is that since the fitting may have more than $\alpha k/5$ columns, we may have to embed a constant number of columns of the fitting in each column of the fitting grid.

We define regions in R called *obstacles* as follows. The region of R to which a finished box B' (or a portion of it) at a smaller level than B is embedded is defined to be an obstacle. Note that the total perimeter of the obstacles in R is at most some constant times the total perimeter of the intersecting region of B.

A *free column* of R is defined to be one that does not pass through any obstacles. From the correspondences between columns in R and rings in the finished box B, in order to find i- and b-rings in B with the required ring properties, it is sufficient to choose i- and b-columns in R with the following *column properties*.

1. The i-column and the b-column of the outer grid must be free columns. Further, between the i-column and the b-column of the outer grid there must be $\Theta(k)$ free columns. A similar condition must hold for the i-column and the b-column of the patch grid.
2. The nodes of the i-column in one grid can be connected to the b-column of the other grid in any permutation using constant-congestion paths of length $\Theta(k)$ that do not pass through any of the obstacles.

Note that from the definition of the obstacles, if path p in R avoids all obstacles then the paths in the four zones that are mapped to p also avoid all the finished boxes at a smaller level than B.

For technical reasons, we would like the placement of the obstacles in the outer grid, the patch grid, and the entire rectangular grid R itself to be symmetric about the columns in the centers of these respective grids, i.e., the obstacles in the first half of the columns of the grid are a mirror image of the obstacles in the second half of the columns of each these three grids. To satisfy this condition we first copy every obstacle in one half of the outer grid to the other half by reflecting this obstacle about its center column. We do the same for the patch grid and then finally for the entire rectangular grid R. This copying can increase the perimeter of the obstacles by at most a constant factor.

We define a square box in R to be *flowless* as follows.

DEFINITION 3.8. *A square box F of side-length q is said to be flowless if and only if either more than $q/4$ rows or more than $q/4$ columns pass through obstacles.* A column of R that does not intersect any of the flowless boxes is called a *live column*. Note that a live column is also a free column, since a box of size 1 that is not flowless can contain no obstacles.

THEOREM 3.9. *For a small enough value of β, a majority of the columns in the outer grid, fitting grid, and patch grid are live columns.*

Proof. Let f denote the number of columns in R that are not live. We can bound f in terms of the total perimeter of the obstacles in the grid by the following counting argument. Initially, let each non-live column have 1 unit of credit associated with it. The total amount of credit in the system is f. For each non-live column h let the largest flowless box that intersects h be box H. This non-live column distributes its unit of credit evenly to nodes on the perimeters of the obstacles contained entirely in H. After every non-live column has redistributed its unit of credit, the total number of credits in the nodes on the perimeters of the obstacles is still f.

Now we will determine the maximum credit received by any node s on the perimeter of an obstacle. Node s may receive credits from many different non-live columns. First we look at non-live columns with smaller column numbers than the column of s. Let the farthest such column from s that contributes to s be at a distance q from s. The flowless box F that intersects this non-live column and contains s must have side-length at least q. The total perimeter of the obstacles of a flowless box of size at least q must be at least $q/4$, since such a flowless box has at least $q/4$ rows or $q/4$ columns that pass through obstacles. Therefore the contribution of this non-live column is at most $4/q$. Further, note that every non-live column between this non-live column and the column of s can contribute at most $4/q$. This is because box F intersects all these columns and hence the size of the largest flowless box intersecting these columns is at least q. Thus the total contribution to s from non-live columns with smaller column numbers than its own column is at most $q \cdot 4/q$ which equals 4. Similarly the total contribution to s from non-live columns with greater column numbers than its own column can also be bounded by 4. Therefore s receives at most 8 credits.

We will now bound f, the number of columns that are not live. The perimeter of the obstacles is at most some constant c (independent of α and β) times the sum of the side-lengths of the finished boxes in the intersecting region of B, i.e., at most $c\beta\alpha k$. Thus the total number of credits in the nodes on the perimeters of the obstacles is at most $8c\beta\alpha k$, so $f \leq 8c\beta\alpha k$. For $\beta < 1/80c$, the majority of the $\alpha k/5$ columns in each of the outer, fitting, and patch grids are live columns. ☐

3.4.1. Permuting grids. We define a *permuting grid* as follows. An $l \times m$ rectangular grid (i.e., a grid with l rows and m columns) is said to be a permuting grid if

1. Each node on the left side of the grid is connected by a path to a distinct node on the right side of the grid. The paths have constant congestion, do not pass through any obstacles, and each path has length $\Theta(m)$. These paths are called the *horizontal paths.*
2. There are at least $m/4$ paths from nodes on the top side of the grid to nodes on the bottom side of the grid such that the congestion of these paths is also a constant. These paths do not pass through any of the obstacles and each path has length $\Theta(l)$. These paths are called the *vertical paths.*

Note that the horizontal and vertical paths in a permuting grid are required to satisfy different properties.

LEMMA 3.10. *The l nodes on the left side of an $l \times m$ permuting grid can be connected in any permutation to the nodes on the right side of the grid using constant-congestion paths of length $\Theta(m)$ that do not pass through any obstacles, provided that $l = O(m)$.*

Proof. The idea is to use the horizontal and vertical paths in the grid as a crossbar. Each node on the left side of the permuting grid is assigned a vertical path such that no vertical path is assigned more than $4l/m$ nodes. Let π denote the permutation to be routed. A path from node v in the left side to $\pi(v)$ in the right side is routed in three stages. In the first stage, a path is routed from v along the horizontal path originating at v to the node where this horizontal path first meets the vertical path assigned to v. In the next stage, the path goes along this vertical path to the node where this vertical path first meets the horizontal path ending at $\pi(v)$. In the last stage, the path goes along this horizontal path to the destination node $\pi(v)$. It is easy to see that this path has length $\Theta(l + m) = \Theta(m)$.

The total congestion on any node in the grid can be split into a sum of three parts. The congestion of a node due to paths in the first (last) stage is at most the congestion of the horizontal paths and hence is constant. The congestion due to paths in the middle stage is constant since it is at most $4l/m$ times the congestion of the vertical paths. Hence for $l = O(m)$ the net congestion is a constant. $\square$

We choose the i- and b-columns from the live columns in the outer and patch grids. Recall that live columns do not pass through flowless boxes. We choose β as small as is required by Theorem 3.9 so that the majority of the columns in the outer, fitting, and patch grids are live columns. Note that since the obstacles are symmetric about the middle column of the outer grid, the live columns of the outer grid are symmetric about the middle column as well. Similarly the live columns in the patch grid and the entire rectangular grid R are symmetric about their middle columns. The live columns with the smallest column number in the outer grid and the patch grid are chosen to be the i-column of the outer grid and the b-column of the patch grid respectively. The live columns with the largest column number in the outer grid and the patch grid are chosen to be the b-column of the outer grid and the i-column of the patch grid respectively. The i- and b-rings in the finished box B are the rings that correspond to the chosen i- and b-columns. Let the grid between the i-column and b-column in the outer grid be O, the grid between the b-column of the outer grid and b-column of the patch grid be F, and the grid between the b-column and i-column of the patch be P.

THEOREM 3.11. *The grids O, F, and P are permuting grids.*

Proof. First we show that O is a permuting grid. O is an $\alpha k/5 \times m$ grid, for some $m \leq \alpha k/5$. Since i- and b-rings in the outergrid are the leftmost and rightmost live columns, respectively, O contains at least $\alpha k/10 \geq m/2 \geq m/4$ live columns. These live columns can serve as the vertical paths in the grid. Now we grow constant congestion paths that do not hit obstacles from every node in the i-column that forms the left side of O to the corresponding node in the b-column that forms the right side of O. These will serve as the horizontal paths of the permuting grid.

The first step is to grow constant-congestion paths that do not hit obstacles from every node in the i-column to nodes in the middle column of O. Every node in the middle column need not have a path ending in it but every node in the i-column must have a path originating in it. We define a series of square boxes of side length $2^i, 0 \leq i \leq \log_2(\alpha k/5)$ (For simplicity, we assume that $\alpha k/5$ is a power of 2). The left side of every box consists of a set of consecutive nodes in the i-column. All the boxes

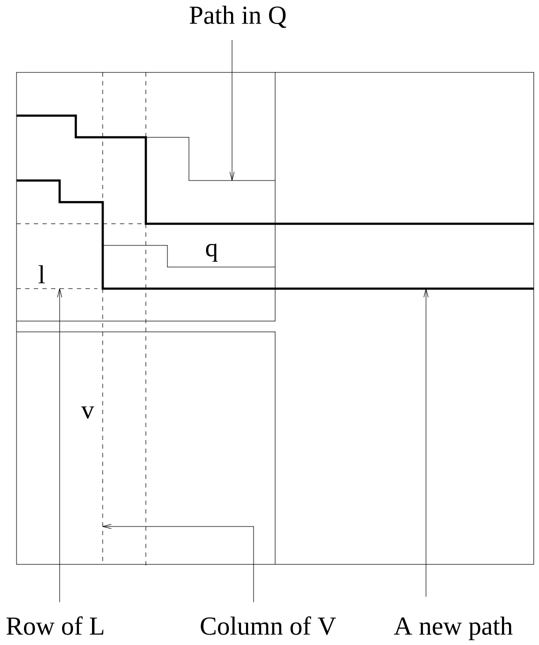

FIG. 3.3. *A new path constructed from path q, column v, and row l.*

of a particular size are numbered consecutively starting from the top and ending at the bottom of the i-column. Boxes of size 1 are the individual nodes in the i-column itself. Given boxes of size 2^i, we obtain boxes of size 2^{i+1} by enclosing every odd numbered box of size 2^i and the succeeding even numbered box with a box of size 2^{i+1}. There will be exactly one box of the largest size and this box encloses O. It is important to note that none of these boxes are flowless since the i-column is a live column.

We grow paths iteratively from smaller sized boxes to larger sized boxes. At the beginning of the ith iteration, we assume that each box of size 2^{i-1} has paths with maximum length $4 \cdot 2^{i-1}$ originating from every node on its left side and ending at some node on its right side. The paths are grouped into sets of 8, such that two paths share an edge only if they belong to the same group. Hence, the congestion of the paths is at most 8. We show how to construct the groups of 8 paths with maximum length $4 \cdot 2^i$ for every box of size 2^i.

For $i = 0$, it suffices to observe that each box of size 1 has no obstacles in it, since these boxes are not flowless. For $i > 0$, each box of size 2^i encloses two smaller boxes of size 2^{i-1} (See Figure 3.3). Let D denote the rectangular box formed by the the first half of the columns of the big box. D encloses both of the smaller boxes of size 2^{i-1}. Since the big box is not flowless, there are at least $(3/4)2^i$ rows that do not hit the obstacles (call this set of rows L) and at least $(3/4)2^i$ columns that do not hit obstacles. Of these $(3/4)2^i$ columns, at least $(1/4)2^i$ columns lie within D (call this set of columns V). Let Q denote the set of paths that are inductively assumed to exist inside both the smaller boxes. We use L and V to extend the paths in Q to nodes on the right side of the big box (See Figure 3.3.) The paths in Q can be ordered sequentially from top to bottom. Likewise the columns in V are ordered sequentially from left to right and the rows in L are ordered sequentially from top to bottom.

First we group the nodes on the left side of the big box into consecutive groups of size 8 each. To the nodes in the ith such group we assign the ith row in L. The paths from the left side of the big box to the right side of the big box are grown sequentially starting from the first group of nodes.

The first group of nodes uses paths in Q until they hit the centermost column in V. Then each of these 8 nodes uses this column in V to reach its assigned row in L. Then it takes this assigned row to reach the right side of the big box (See Figure 3.3). When routing the next group we must make sure that we do not overlap these paths with the paths already routed since this would increase the congestion. Let w be the column in V that was used by the previous group of nodes. Further suppose that the last node in this group turned upward into column w to reach its row in L. In this case, we use the column in V that succeeds w for the current group of nodes. Similarly, if the last node of the previous group turned downward into column w we use the column in V that precedes w for the current group. As before, the paths in the current group follow paths in Q until they hit the chosen column in V and then use this column until their assigned row in L. These paths do not share edges with any of the previous paths. We use this procedure to route paths from all the groups of nodes. Since we have $2^i/8$ columns to the right and to the left of the centermost column in V and since there are at most $2^i/8$ groups of nodes, we will never run out of columns in V. Since the paths outside of a group do not overlap, the congestion is at most 8. The maximum length of any path is at most the maximum length of any path in Q ($4 \cdot 2^{i-1}$ by the inductive hypothesis) added to the maximum length of the newly added portion (at most $2 \cdot 2^i$) which is $4 \cdot 2^i$.

After constructing paths in progressively larger boxes, we will have constructed paths from every node of the i-column to the right side of a square box of size $\alpha k/5$. These paths can be truncated at the middle column of O. Note that the obstacles in O are symmetric about its middle column. From this symmetry, exactly the same paths reflected about the middle column connect every node in the b-column to the same set of nodes in the middle column. Concatenating these two sets of paths, we obtain paths from every node in the b-column to a corresponding node in the i-column of congestion at most 8 and length at most $8\alpha k/5 = \Theta(m)$. Thus O is a permuting grid.

The proof that F and P are permuting grids is similar. ∎

THEOREM 3.12. *The b-columns and i-columns satisfy both of the column properties.*

Proof. The first property is true since the i- and b-column of the outer grid or the patch grid are chosen so that there are $\Theta(k)$ live columns between them. The second property follows from Lemma 3.10 and Theorem 3.11. To connect the nodes of the i-column of the patch grid to the b-column of the outer grid in some arbitrary permutation, we use the permuting grid P followed by the permuting grid F. One of the grids will be used to route the required permutation and the other will route the identity permutation. From Lemma 3.10, the paths obtained have constant congestion and do not pass through obstacles. Furthermore each path is $\Theta(k)$ in length. To connect the nodes from the b-column of the patch grid to the i-column of the outer grid in an arbitrary permutation, we use grid F followed by grid O. ∎

4. A limit on the fault-tolerance of linear arrays. Unlike two-dimensional arrays, one-dimensional arrays are not very fault tolerant. For example, placing $f(N)$ evenly-spaced faults in an N-node linear array splits the array into disjoint pieces of size $N/f(N)$, for any function $f(N)$. Emulating the entire linear array on one of these pieces entails a slowdown of at least $f(N)$. Thus if $f(N)$ grows as a function of N, the slowdown is not constant. However, if we assume a weaker model of faults in which a faulty node cannot perform any computation but can communicate with its neighbors, then the linear array becomes more fault tolerant. In particular, the following theorem shows that an N-node linear array can tolerate $\log^k N$ worst-case faults, for any constant $k > 0$, and still emulate a fault-free N-node linear array with constant slowdown.

THEOREM 4.1. *For any constant $k > 0$, an N-node linear array with $\log^k N$ worst-case faults can emulate T steps of any computation of a fault-free N-node linear array in $O(T + N)$ steps, provided that faulty nodes can communicate with their neighbors.*

Proof. It is straightforward to apply the emulation scheme from Section 2 to the linear array. □

In the remainder of this section we show that an N-node linear array with more than $\log^{O(1)} N$ worst-case faults cannot perform a static emulation of a fault-free N-node array with constant slowdown.

4.1. Bounding the load, congestion, and dilation. For the sake of convenience, we repeat the definition of a static emulation here. In a *static* emulation, a *redundant* guest network $G' = (V', E')$ is embedded in the host H. The redundant network is defined as follows. For every node v in the guest network $G = (V, E)$, there is set of nodes $\pi(v)$ in V'. Each set $\pi(v)$ contains at least one node, and for $u \neq v$, $\pi(v)$ and $\pi(u)$ are disjoint. We call the nodes in $\pi(v)$ the *instances* of v in G'. The network G' is called redundant because it may contain several instances of each guest node. For every node $v' \in \pi(v)$, and every edge (u, v) in E, the redundant network contains a directed edge (u', v'), for some $u' \in \pi(u)$. The embedding maps nodes of G' to non-faulty nodes in the host, and edges of G' to paths in the host. In this section we allow the paths to pass through faulty host nodes.

The host emulates T steps of the guest network's computation as follows. The embedding of G' into H maps a set $\psi(a)$ of nodes of G' to each host node a. Node a emulates each node $v' \in \psi(a)$ by creating an s-pebble $\langle v', t\rangle$ for $1 \leq t \leq T$. An s-pebble $\langle v', t\rangle$ represents the state of node v' at time t. Initially, each node a of H contains s-pebbles $\langle v', 0\rangle$ for $v' \in \psi(a)$. Node a can create an s-pebble $\langle v', t\rangle$ only if it is not faulty and has already created an s-pebble $\langle v', t-1\rangle$, and has received all of the c-pebbles of the form $[e, t-1]$, where e is an edge (u', v') into v'. A c-pebble $[e, t-1]$ represents the communication that v' receives from its neighbor u' in step $t-1$. After creating an s-pebble $\langle v', t\rangle$, a non-faulty node a can create all of the c-pebbles of the form $[g, t]$ for each edge g out of v'. At each host time step a non-faulty host node a can create a single s-pebble (and the corresponding c-pebbles). In this section, and this section only, we assume that any node, faulty or non-faulty, can send and receive one c-pebble on each of its edges at each step. A c-pebble for an edge (u', v') is sent along the path from u' to v' that is specifed by the embedding. Note that a node u' may send c-pebbles to a neighbor v', but receive c-pebbles from a different instance v'' of guest node v.

The following three lemmas show that if a static emulation has slowdown s, then the load and congestion of the embedding of G' into H cannot exceed s, and the average dilation of the edges on any cycle in G' cannot exceed s.

LEMMA 4.2. *Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in Ts steps. Then the maximum load on any host node is at most s.*

Proof. Let l be the load of the embedding. Then some node a in H must emulate

l nodes of G'. For each of these nodes, a must create T s-pebbles. Since a can create at most one s-pebble at each step, the total time is at least lT. Thus, if the slowdown is s, the load can be at most s. ∎

LEMMA 4.3. *Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in Ts steps. Then the maximum congestion on any host edge is at most s.*

Proof. Let c be the congestion of the embedding. Then there is some host edge e through which c paths pass. For each of these paths, T c-pebbles must pass through e. Since e can transmit at most one c-pebble at each step, the total time is at least cT. Thus, if the slowdown is s, the congestion can be at most s. ∎

LEMMA 4.4. *Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in at most Ts steps. Then the average dilation of the edges on any cycle in G' is at most s.*

Proof. Suppose that there is a cycle of length L in G' with dilation D (the dilation of a cycle is the sum of the dilations of its edges). Let $v'_{L-1}, v'_{L-2}, \ldots, v'_0$ denote the nodes on the cycle. For any t, the s-pebble $\langle v'_0, t\rangle$ cannot be created until a c-pebble $[(v'_1, v'_0), t-1]$ arrives at the host node that emulates v'_0. Since a c-pebble can traverse at most one host edge at each time step, the time for the c-pebble to travel from the node that emulates v'_1 to the node that emulates v'_0 is at least the dilation of the edge (v'_1, v'_0). The dilation is also a lower bound on the time between the creation of s-pebbles $\langle v'_1, t-1\rangle$ and $\langle v'_0, t\rangle$. Working our way around the cycle, we see that the time between the creation of s-pebbles $\langle v'_0, t-L\rangle$ and $\langle v'_0, t\rangle$ is at least the dilation of the cycle, D. Thus, for any T that is a multiple of L, the time between the start of the emulation and the creation of s-pebble $\langle v'_0, T\rangle$ is at least TD/L. For $D/L > s$, this pebble is not created until after step Ts, a contradiction. ∎

4.2. Bounding the number of faults. THEOREM 4.5. *For any s, there is a pattern of $h(s)(\log N)^{2s}$ worst-case faults, for any $h(s) > 2^{6s+4}s^{6s+5}$, such that it is not possible for an N-node host linear array with these faults to perform a static emulation of an N-node guest linear array with slowdown s.*

Proof. We begin by placing a layer of $g(s)$ blocks of $f(s)$ consecutive faults in an N-node array so that the number of non-faulty nodes in the gap between each pair of blocks is at most $N/g(s)$. Formulas for $g(s)$ and $f(s)$ will be determined later.

Next we find a block of faults B that some edges of the redundant network must cross. Because the slowdown is s, and at most $N/g(s)$ host nodes lie between any pair of blocks, for $g(s) > s$, it is not possible for the entire emulation to take place in one gap. (If it did, then the load in the gap would be greater than s, which is forbidden by Lemma 4.2.) Since the emulation uses host nodes in at least two gaps, there must be some block B such that some, but not all, of the the guest nodes are emulated on its left, and some, but not all, of the guest nodes are emulated on its right.

Now we find a cycle C in the redundant network G' that crosses B. Let u be a node in the guest network G such that every instance of u in G' on the right side of B receives its left input from the left of B; if there is no such u, then let u be a node in the guest network such that every instance of u on the right side of B receives its right input from the left of B; in the latter case interchange the role of left and right inputs in what follows. Note that since we have chosen B so that the host does not emulate the entire guest on the right side of B, there must be such a node u. Select one of the instances, u', of u and follow the left input edge into u' (i.e., the input edge coming from the node in G' that corresponds to the left neighbor of u in the guest) back to where it came from. It must lead across B to some node v' in G' on the left side of B.

Now follow the left input edge into v' to some other node w' in G' (node w' may be on either side of B). Continue to follow left input edges until reaching a node x' that corresponds to the left endpoint of G. Then follow right input edges until reaching the right endpoint of G, and reverse direction again. Repeat this process until some edge of G' is used twice. When this happens, a cycle C is formed. Furthermore, the cycle C must cross B because it visits every node of G and we know that H does not emulate all of G on one side of B.

The next thing to show is that on one side of B or the other, cycle C visits at least l consecutive nodes of the guest network, where $l > f(s)/2s$, and these nodes are emulated within distance $2sl$ of B in the host. If the slowdown of the emulation is s, then by Lemma 4.4 the dilation of any cycle is at most s times the number of redundant network nodes on the cycle. (The dilation of a cycle or path is equal to the sum of the dilations of the edges on the cycle or path.) Let us define a *segment* to be a maximal subpath of C that begins with an edge that crosses block B, but does not cross B again. Note that every segment either consists of a sequence of right input edges followed by a (possibly empty) sequence of left input edges, or vice versa. Suppose that cycle C crosses block B a total of $2h$ times. Then there are $2h$ segments. Associate with each segment the dilation of the edges on the segment. Note that the average ratio of the dilation of a segment to the number of nodes on the segment must be at most s (since the ratio for the entire cycle C is at most s). Now classify segments into two types: long and short. A short segment is one containing fewer than $f(s)/s$ edges. Since every segment has dilation at least $f(s)$ (due to the first edge on the segment), the ratio of a short segment's dilation to length (number of nodes) is more than s. Since the average ratio over all of the segments is at most s, there must be some long segment whose ratio of dilation to length is at most s. If this segment has more left input edges than right input edges, then discard the right input edges and the nodes that they visit. Otherwise, discard the left input edges. We are left with some set of $l \geq f(s)/2s$ nodes emulated within distance $2sl$ of B. Suppose that there are more left input edges, and let $v'_1, v'_2, \ldots, v'_l$ denote the nodes that were visited on (say) the right side of B, where v_1 is the leftmost node in the guest network. We will call the $2sl$ host nodes on the right of B the *emulation region*. (Note that in the construction of the cycle, we visited v'_l first and v'_1 last.)

Now we show that some communication must pass over the emulation region. Although nodes $v_1, v_2, \ldots, v_l$ are consecutive in the guest network, their instances are not necessarily embedded in the host in consecutive order. Suppose that v'_i is the node embedded the farthest to the right. If $i > l/2$, then the path in the cycle from the left side of B to v'_l to v'_{l-1} and on to v'_i overlaps all of nodes v'_1, $v'_2, \ldots, v'_{l/2}$. On the other hand, if $i \leq l/2$, then the path from v'_i to v'_{i-1} to v'_1 and back across to the left side of B overlaps all of nodes $v'_{l/2+1}, v'_{l/2+2}, \ldots, v'_l$. In either case, we have a set of $l/2$ consecutive nodes in the guest network that the host emulates, and some other edges of G' overlap their emulation with congestion 1.

We now proceed recursively within the emulation region. One last issue that must be dealt with is that some of the $l/2$ nodes that the host is emulating within the emulation region may receive some of their right inputs from outside the emulation region. However, since the embedding has congestion at most s (by Lemma 4.3), at most $2s$ right inputs can enter the emulation region from outside. Thus, there must be a set of at least $(l/2)/2s = l/4s$ redundant network nodes that the host emulates within the emulation region that are consecutive in the guest and receive all of their inputs from within the emulation region. At this point we have placed $g(s)$ blocks of

$f(s)$ faults in the network and we have proved that on one side of one of the blocks, there is an emulation region of size $2sl$ in which at least $l/4s$ consecutive nodes of the guest are emulated, for some $l \geq f(s)/2s$, and some other edges of G' cause congestion 1 in the emulation region. In order to recurse on sets of $l/4s$ guest nodes, where $l \geq f(s)/2s$, we need $f(s) > 8s^2$.

We are now going to place an additional layer of faults in the network. Because we do not know where the emulation region is, we will place faults immediately adjacent to both sides of each of the $g(s)$ blocks of faults in the first layer. Also, because we do not know how large the emulation region is, we will place the faults in patterns of size $2, 4, 8, \ldots, N$ on top of each other. (Note that N is the size of the entire array.) In a pattern of size 2^k, we will place $g(s)$ blocks of $f(s)$ consecutive faults at spacings of $2^k/g(s)$. Thus, in each pattern there are at most $g(s)f(s)$ faults, and there are at most $\log N$ patterns on each side of the blocks in the first layer. The total number of faults in the second layer is $2g^2(s)f(s)\log N$.

The entire emulation region must lie under some pattern P of faults of size 2^k, where $2^k \leq 4sl$. The blocks of faults in this pattern are spaced at a distance of $2^k/g(s)$, which is at most $4sl/g(s)$. In this region, at least $l/4s$ guest nodes are emulated. If the slowdown is at most s, and $(l/4s)/(2sl/g(s)) > s$, then by Lemma 4.2 it is not possible for the entire emulation to be performed entirely between two blocks of faults in this pattern. (Thus, we need $g(s) > 8s^3$.) Arguing as we did for the first layer, we can show that, for some l', on one side of one of the blocks of P, there is an emulation region of size $2sl'$ and a set of least $l'/4s$ nodes that are consecutive in the guest network that receive their inputs from within the emulation region. But now two units of congestion pass over the new emulation region (possibly in opposite directions).

A third layer of faults is now placed in the network. As before, a set of patterns of faults is placed around each block in the second layer. There are $2g(s)^2 \log N$ blocks in the second layer. Thus, there are $4g(s)^3(\log N)^2 f(s)$ faults in the third layer.

By applying $2s+1$ layers of faults, we find an emulation region over which at least $s+1$ units of congestion (in one direction) pass, which is a contradiction by Lemma 4.3. The $(2s+1)$st layer contains $2^{2s}g(s)^{2s+1}(\log N)^{2s}f(s)$ faults. The total number of faults contained in all the $2s+1$ layers is at most twice the number of faults contained in the $(2s+1)$st layer alone, since the number of faults in the ith layer is at least double the number of faults in the $(i-1)$st layer. Thus the total number of faults is at most $2^{2s+1}g(s)^{2s+1}(\log N)^{2s}f(s) = h(s)\log^{2s} N$, where $h(s) = 2^{2s+1}g(s)^{2s+1}f(s)$, $g(s) > 8s^3$ and $f(s) > 8s^2$. □

5. Remarks. The scheme described in Section 2 for tolerating $\log^k N$ faults in an $N \times N$ mesh can be easily generalized to tolerate $\log^k N$ faults in a d-dimensional array with side-length N for any fixed $d > 2$. It seems plausible that the techniques described in Section 3 can also be generalized to arrays of higher dimension.

REFERENCES

[1] M. Ajtai, N. Alon, J. Bruck, R. Cypher, C. T. Ho, M. Naor, and E. Szemerédi. Fault tolerant graphs, perfect hash functions and disjoint paths. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 693–702, October 1992.

[2] Y. Aumann and M. Ben-Or. Computing with faulty arrays. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 162–169, May 1992.

[3] T. Blank. The maspar mp-1 architecture. In *compcon90*, pages 20–24, February 1990.

[4] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J Webb. iWarp, an integrated solution to high-speed parallel computing. In *Proceedings Supercomputing '88*, pages 330–339, November 1988.

[5] J. Bruck, R. Cypher, and C.–T. Ho. Fault-tolerant meshes with small degree. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10, June 1993.

[6] M. R. Fellows. *Encoding Graphs in Graphs.* PhD thesis, Department of Computer Science, University of California, San Diego, CA, 1985.

[7] J. W. Greene and A. El Gamal. Configuration of VLSI arrays in the presence of defects. *Journal of the ACM*, 31(4):694–717, October 1984.

[8] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, and A. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 285–296. IEEE Computer Society Press, October 1990.

[9] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg. Work-preserving emulations of fixed-connection networks. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 227–240, May 1989.

[10] R. K. Koeninger, M. Furtney, and M. Walker. A shared MPP from Cray research. *Digital Technical Journal*, 6(2):8–21, Spring 1994.

[11] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-shop scheduling in $O(\text{congestion} + \text{dilation})$ steps. *Combinatorica*, 14(2):167–180, 1994.

[12] T. Leighton and C. E. Leiserson. Wafer-scale integration of systolic arrays. *IEEE Transactions on Computers*, C–34(5):448–461, May 1985.

[13] T. Leighton, B. Maggs, and R. Sitaraman. On the fault tolerance of some popular bounded-degree networks. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 542–552, October 1992.

[14] T. R. Mathies. Percolation theory and computing with faulty arrays of processors. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 100–103, January 1992.

[15] F. Meyer auf der Heide. Efficiency of universal parallel computers. *Acta Informatica*, 19:269–296, 1983.

[16] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal on Computing*, 15(1):106–119, February 1986.

[17] F. Meyer auf der Heide and R. Wanka. Time-optimal simulations of networks by universal parallel computers. In *Proceedings of the 6th Symposium on Theoretical Aspects of Computer Science*, pages 120–131, 1989.

[18] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-Machine multicomputer: an architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–235, May 1993.

[19] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), April 1989.

[20] P. Raghavan. Robust algorithms for packet routing in a mesh. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 344–350, June 1989.

[21] E. J. Schwabe. On the computational equivalence of hypercube-derived networks. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 388–397, July 1990.

[22] H. Tamaki. Efficient self-embedding of butterfly networks with random faults. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, October 1992. 533-541.

[23] H. Tamaki. Robust bounded-degree networks with small diameters. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 247–256, June 1992.

[24] H. Tamaki. Construction of the mesh and the torus tolerating a large number of faults. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 268–277, June 1994.