Communication and Fault Tolerance in Parallel Computers

Ramesh Kumar Sitaraman

Research Report CS-TR-xxx-93
February 1993

COMMUNICATION AND FAULT TOLERANCE IN PARALLEL COMPUTERS

Ramesh Kumar Sitaraman

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1993

Dedicated to Amma and Aunty

varam buddhir na saa vidyaa
vidyaato buddhir uttama
buddhiheena vinashyanti
yathaa te simhakaarakaa:

Scholarship is less than sense;
Therefore seek intelligence:
Senseless scholars in their pride
Made a lion; then they died.

-From the story "The lion who ate the scholars",
"Tales of the Panchatantra", circa 500 B.C.
Translated from Sanskrit by Arthur W. Ryder.

Abstract

This thesis explores two fundamental issues in the design of large-scale parallel computers: *communication* and *fault tolerance*.

In Chapter 1, we introduce and provide motivation for the problems that we study in this thesis.

Chapter 2 examines several simple algorithms for routing packets on butterfly networks with bounded queues. Among other things, we show that for any greedy queuing protocol, a routing problem in which each of the N inputs sends a packet to a randomly chosen output requires $O(\log N)$ steps, with high probability, provided that the queue size is a sufficiently large, but fixed, constant.

In Chapter 3, we analyze the fault-tolerance properties of several bounded-degree hypercubic networks that are commonly used for parallel computation. Among other things, we show that an N-node butterfly containing $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$) can emulate a fault-free butterfly of the same size with only constant slowdown. Similar results are proved for the shuffle-exchange graph. Hence, these networks become the first connected bounded-degree networks known to be able to sustain more than a constant number of worst-case faults without suffering more than a constant-factor slowdown in performance.

In Chapter 4, we study the ability of array-based networks to tolerate faults. Among other things, we show that an $N \times N$ two-dimensional array can sustain $N^{1-\epsilon}$ worst-case faults, for some fixed $\epsilon < 1$, and still emulate a fully-functioning $N \times N$ array with only constant slowdown.

In Chapter 5, we study a concurrent error detection scheme called Algorithm Based Fault Tolerance (ABFT). Unlike the schemes developed in Chapters 3 and 4 to tolerate permanent faults, the scheme studied in this chapter is primarily aimed at tolerating transient faults in a parallel computer. The main contribution of this chapter is to propose a simple and novel algorithm called RANDGEN to generate data-check relationships. By simply varying its parameters, RANDGEN can produce data-check relationships with a wide spectrum of properties, many of which have been considered important in recent ABFT designs.

Acknowledgments

First and foremost, I would like to thank my mother for her countless sacrifices and her long-term vision. Her deep respect for education planted the seeds for my graduate study.

I would like to thank my advisor Bob Tarjan for his support and encouragement. I am indebted to him for providing me with the ideal conditions to do my work.

Bruce Maggs is a researcher and teacher of extraordinary merit. Bruce has been a guiding influence in my maturing as a researcher. His friendship and sense of humor brought many a cheer.

Working with Tom Leighton has been a great pleasure. I am constantly amazed at how much he can do in an hour! I also thank him for sharing my views on football in an otherwise hostile environment!

I have learned a lot working with Richard Cole. He has taught me by example the value of optimism and tenacity in approaching even the hardest of problems.

Niraj Jha provided me with my first taste of fault-tolerance. I am also grateful to Niraj for giving me a more rounded view of the practice of fault-tolerance.

Bruce, Tom, Richard, and Niraj made direct contributions to this thesis by collaborating with me on various parts.

I have benefited from technical discussions with many of my colleagues. In particular, I would like to mention Satish Rao, Bill Aiello, Bernard Chazelle, and Hsing-Mean Sha.

I am grateful to Bernard for his course on Randomized Algorithms. It gave me many of the tools needed for my thesis work.

I would like to thank my alter ego, Ravi Kuchimanchi, for his deep friendship.

Back home in India, I would like to thank my relatives, especially my Aunty who I know has always deeply cared about me.

I would like to convey my special thanks to my apartment mates S.V. Krishnan and V. Venkataraman for their constant support on the home-front, much beyond the call of friendship.

I would like to acknowledge Joel Friedman and Jin-Yi Cai for their wise counsel. I would like to thank my good friend Hsing-Mean Sha for all our discussions on eastern philosophy. I am grateful to Ben Kao, Anthony Tomasic, Dan Boneh, and Chuck Rose for being such cheery office mates. I would like to thank my many friends at Princeton for making my life here so interesting. I would like to thank Melissa Lawson and Sharon Rodgers for their help with administrivia.

Financial support was provided by a DARPA/ONR Grant under Contract no. N00014-88-K-0459, and DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center-NSF-STC88-09648.

Finally, I would like to thank Bruce and Joel for consenting to be my readers, and Bernard and Andrea LaPaugh for consenting to be my examiners.

Contents

List of Figures

Chapter 1

Introduction

Large scale parallel computing is becoming an indispensable tool in areas of science and technology as diverse as fluid mechanics, climate prediction, and molecular biology. In the coming decade, it may become technologically feasible to build parallel machines with a few thousand integrated circuit chips and a large number of processing elements. However, there are many hardware and software obstacles that must be overcome before parallel computing can realize its full potential. In this thesis, we focus on two fundamental issues that must be understood for the successful design of large-scale parallel computers: *communication* and *fault tolerance*.

1.1 Communication

A parallel computer consists of many processors linked together by an interconnection network. Each processor has its own local memory. Since the processors must cooperate in solving a problem, there is a necessity for processors to exchange information. This is achieved by routing messages (or packets) from processor to processor using the interconnection network.

An interconnection network can be represented as a directed graph. The nodes of the graph represent processors or switches and the directed edges represent communication links. There is a *queue* associated with every edge of the network (See Figure 1). Each queue can hold only a small number of packets at any given time step. Each processor has two buffers: an *output buffer* to store packets that it wants to send and an *input buffer* to store the packets that it receives. (These buffers are not shown in the figure.) Unlike the queues, the buffers can hold a large number of packets. The interconnection network is used to route packets between processors as follows. Suppose processor A produces a packet that needs to be routed to processor C. First processor A stores the packet in its output buffer. The packet is then transmitted from A to C along a directed path in the network. The packet upon reaching C is stored in the input buffer of C until the time when it can be consumed. At each time step, exactly one packet can be transmitted across an edge of the graph. Therefore, a packet may wait in the queue associated with an edge till it can

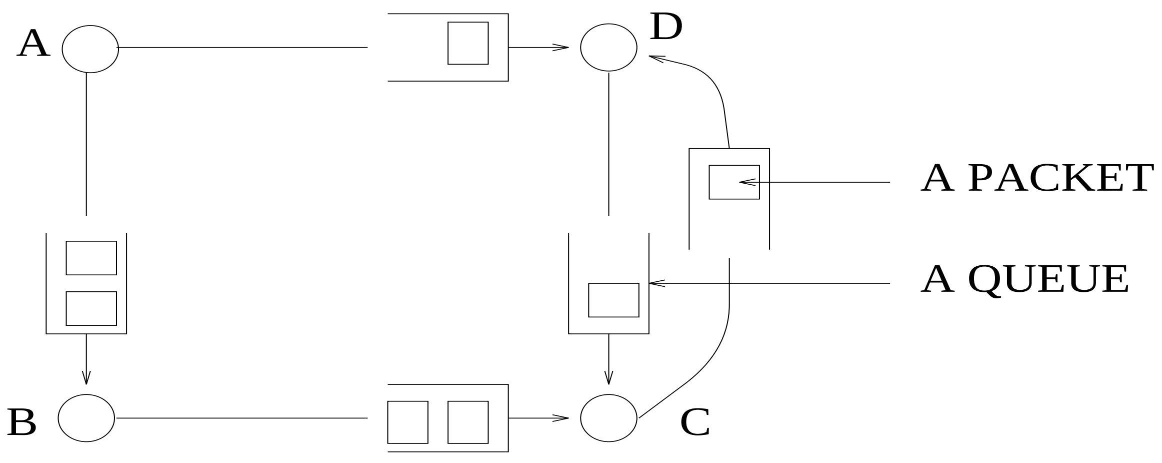

Figure 1: An interconnection network.

use the next edge in its path. The queues of the incoming edges of a switch can be thought of as physically residing in the switch. In the simplest model of a switch, we can assume that all the incoming edges share a single queue located in the switch.

At every time step, a switch uses a procedure known as the *queuing protocol* to select a packet from one of the queues of its incoming edges or from its output buffer, and it transmits this selected packet through the appropriate outgoing edge. (Alternately, a switch may be permitted to transmit one packet along each of its outgoing edges in one time step.) The packets that are not selected by the switch are delayed and have to wait for their turn to be sent. In general, the total delay experienced by a packet is a function both of the other packets in the network and the queuing protocol. While it is convenient to describe the switches in the network as operating in a synchronous fashion, our results on packet routing hold in an asynchronous setting as well.

It has been experimentally observed that communication time rather than computation time plays a major role in determining the time taken for the execution of many parallel programs. Fine-grained parallel programs execute as few as ten instructions in response to a message [Dal87]. This implies that a significant portion of the execution time of these parallel programs is spent in waiting for messages to arrive. Therefore designing efficient, implementable packet routing algorithms is bound to yield high dividends in speeding up parallel programs. In Chapter 2 of this thesis, we study packet routing on a popular interconnection network called the *butterfly network.* We restrict our study to simple classes of queuing protocols that are easy to implement like the First-In First-Out protocol (FIFO). We analyze the maximum delay incurred by a packet under such a queuing protocol for random-traffic as well as worst-case traffic conditions. The conclusion of this work is that there are a variety of simple queuing protocols that provably route packets in an optimal number of time steps (to within a constant factor) in butterfly networks.

1.2 Fault tolerance

The second theme of this thesis is fault-tolerance. The manufacturing technologies involved in building parallel computers are highly complex and are prone to hardware defects. Large systems built with VLSI (Very Large Scale Intergration) or WSI (Wafer Scale Integration) technologies are extremely likely to have failed components at the production stage. This can cause unacceptably low yield rates of fully functioning hardware. (For a report on the sources of such failures and their impact on the yield, please refer to [MR84].) However, if it is possible to *reconfigure* some of the defective chips or wafers and make them usable, the yield rate of *usable hardware* becomes much higher. This has been called *reconfiguration for yield* in the literature [NSS89].

During the course of operation of a large parallel computer, more faults and defects are likely to occur. Reconfiguring the computer to avoid these unreliable or failed units is known as *reconfiguration for reliability* and is aimed at achieving a higher life-time of operation of the parallel computer. In this thesis, we do not differentiate between the two different goals of reconfiguration and our reconfiguration techniques can be used in either setting.

1.2.1 Fault model

Any fault-tolerance technique presumes a *fault model*. Faults can be thought of as occurring at various levels in the circuitry. A popular model for faults at the logic gate level is the *stuck-at* fault model. In this model, a faulty output line of a gate is either permanently stuck at 0 or permanently stuck at 1. However, due to its extremely low-level complexity and its dependence on specific circuit level implementations, this model is not suitable for studying reconfiguration schemes for parallel computers. For our purposes, we require a model in which faults occur at the *functional level* of the parallel computer. In this thesis, we will model all failures as *processor failures* of the parallel computer. For convenience, even the failure of a communication link will be modeled as the failure of one of its adjacent processors. We will assume that a faulty or failed processor cannot perform *any* computation reliably. Further, we will assume that no packets can be routed through a faulty processor.

Next, the fault model must make assumptions about the way the faults are distributed. It is perhaps tempting to assume that each processor fails independently of all others with a failure probability p. This is called the *random fault model*. These assumptions do model to a limited extent what happens in practice, but the reality is much more complex. It has been experimentally observed that defects in a chip or wafer tend to cluster together. This is particularly true of defects caused by extraneous particles since such particles tend to cluster in one geometric area of the chip or wafer. Other correlations introduced by the fabrication method are too complex to model. To overcome these problems, one can assume the *worst-case fault model*. In the worst-case fault model, we make no probabilistic assumptions about the distribution of faults. We simply assume that the faults occur in the worst possible locations making it difficult to reconfigure the chip. While this model overcomes the problems of the random fault model, it has the disadvantage of being overly pessimistic. The worst-case distribution of faults may seldom occur in practice. In this thesis, we will be concerned primarily with worst-case faults though we will derive some results for the random fault model as well.

1.2.2 Reconfiguration

Formally, a *reconfiguration scheme* is a technique that allows a faulty parallel computer to perform any computation of a fault-free parallel computer. Since a faulty parallel computer has only a subset of the resources available to a fault-free parallel computer, it is inevitable that the computations on the reconfigured parallel computer run slower than they would on the fault-free parallel computer. A measure of this degradation in performance is called *slowdown* which is defined to be the minimum value S such that any computation that takes T steps on the fault-free computer can be performed in at most $S \times T$ steps in the reconfigured faulty computer.

In Chapters 3 and 4 of this thesis, we will devise reconfiguration schemes with small slowdowns for commonly used interconnection networks (usually the slowdown is some constant, independent of the size of the network). The reconfiguration schemes presented in these chapters take the form of a general emulation of the fault-free parallel computer on the faulty parallel computer. Hence these reconfiguration schemes are software-independent and make no assumptions about the specific piece of software that is to be run on the faulty computer. Further, we do not assume that the processors compute synchronously. The emulations in this thesis are valid even if the processors work asynchronously.

Most of the networks used in practice fall into two broad categories: *hypercubic networks* and *array-based networks*. In Chapter 3, we study the fault-tolerance of hypercubic networks such as the fat-tree, mesh of trees, butterfly and the shuffle-exchange network. In Chapter 4, we study the fault-tolerance of array-based networks such the linear array and two- and higher-dimensional meshes. The conclusion of these studies is that these commonly used networks can be reconfigured around an unexpectedly large number of faults with only constant slowdown.

It is also possible to talk about a reconfiguration scheme for a *particular task*. For example, if a network is used for packet routing, a reconfiguration scheme is a technique that allows a faulty network to perform routing tasks that can be performed on a fault-free network. A measure of degradation in performance is the ratio of the time taken to perform the task on the faulty network to the time taken to perform the task on the fault-free network. In Chapter 3, we show that hypercubic networks can be used route packets without much degradation in performance even in the presence of many faults. This is of importance since hypercubic networks such as the butterfly are often used solely for their routing abilities.

1.2.3 Algorithm-based fault tolerance

We have so far considered the problem of reconfiguring parallel computers around *permanent faults*. We have assumed that when a processor fails the failure is permanent and the computer is reconfigured to avoid this failed processor. However a common source of unreliability in parallel computers today is *transient faults* [CMS82, IR83, SRHA86, YFA87]. A transient fault is the temporary malfunction of a functional unit of the computer. A transient fault in a processor could introduce errors in its computation. Since these faults are temporary in nature they may not be detected by standard diagnostic algorithms. A primary source of transient faults is noise induced in the circuitry. The current trend in VLSI chip design towards using smaller and smaller voltage levels has the effect of decreasing the margin allowed for noise. This is bound to make transient faults a major reliability issue in the future.

A solution to the problem of transient faults is to perform concurrent error detection (CED). In a CED scheme, an error detection algorithm is run concurrently with the computation. If the error detection algorithm detects an error, steps can be taken to locate the faulty components. In the more likely case that the error is caused by a transient fault, simply re-running the computation may be sufficient. Unlike the software-independent approach in Chapters 3 and 4, different CED schemes are suitable for different kinds of computing tasks. A class of CED schemes called Algorithm-based fault tolerance (ABFT) was proposed for matrix operations like matrix multiplication, matrix triangularization etc. [HA84]. Much research has been done on the ABFT approach recently to extend it to other signal processing applications including the FFT. An abstract graph theoretic model for both analyzing and synthesizing ABFT was proposed in [BA86a]. In Chapter 5, we show how to synthesize ABFT systems in this graph-theoretic model to detect or locate t faults in the system. The method presented in this chapter greatly reduces the overhead required to do error detection/location in comparison with methods known previously.

Chapter 2

Simple Algorithms for Packet Routing

2.1 Introduction

Many commercial and experimental parallel computers, including the NYU Ultracomputer [Got87], the IBM RP3 [PBG+87], the BBN Butterfly [BBN86], and NEC's Cenju [NMT+91], use butterfly networks to route packets between processors. Although many routing algorithms with provably good performance have been devised for butterfly networks [Ale82, LMR88, Pip84, Ran87b, Ran87c, Upf84, Val82, Val90, VB81], simpler algorithms are often used in practice. Typically, packets are sent along shortest paths through the network, and simple queuing protocols such as first-in first-out (FIFO) are used to determine which packets to transmit at each step. In addition, the queues at the switches can usually hold only a small number of packets. The performance of these simple algorithms has proven surprisingly difficult to analyze. For example, the only previously known upper bound on the time required for each input of an N-input butterfly network with constant-size FIFO queues to route a packet to a random destination was $O(N)$. In this chapter, we show that the expected time is actually $\Theta(\log N)$. We also analyze the performance of several other simple algorithms for routing on butterflies with bounded queues.

2.1.1 Butterfly networks

An example of an N-input butterfly ($N = 8$) with depth $\log N = 3$ is shown in Figure 2. *All logarithms in this thesis are to base 2.* The edges of the butterfly are directed from the node in the smaller numbered level to the node in the larger numbered level. The nodes in this directed graph represent switches, and the edges represent communication links. We use the terms node and switch interchangeably in the rest of the chapter. Each node in a butterfly has a label $\langle l, c_0 \cdots c_{\log N - 1} \rangle$, where the *level*, l, ranges from 0 to $\log N$, and the *row*, $c_0 \cdots c_{\log N-1}$, is a $\log N$-bit binary string. The switches on level

This chapter describes joint research with Bruce Maggs [MS92].

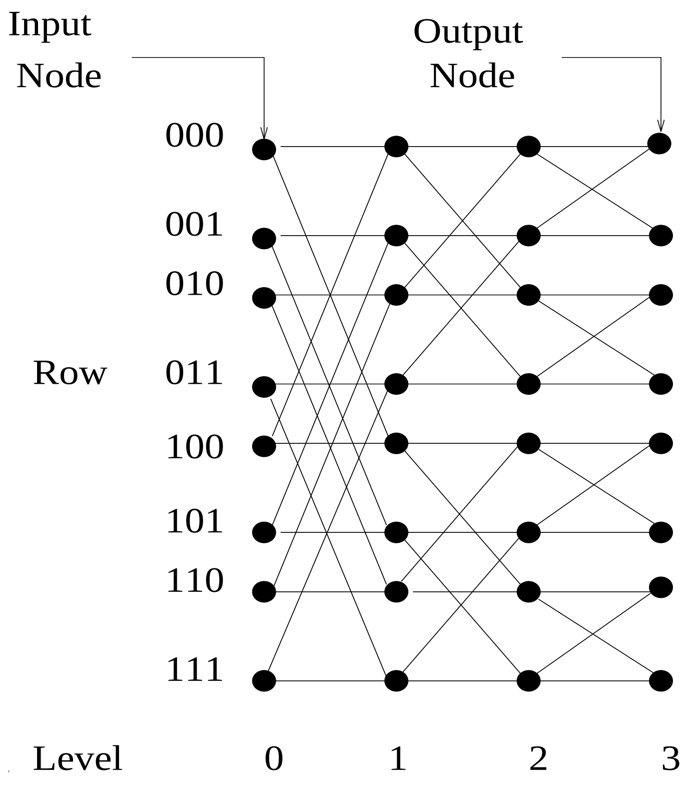

Figure 2: An 8-input butterfly network.

0 are called *inputs*, and those on level $\log N$ are called *outputs*. For $l < \log N$, node $\langle l, c_0 \cdots c_l \cdots c_{\log N-1}\rangle$ is connected to node $\langle l+1, c_0 \cdots c_l \cdots c_{\log N-1}\rangle$ by a *straight* edge, and to node $\langle l+1, c_0 \cdots \overline{c_l} \cdots c_{\log N-1}\rangle$ by a *cross* edge. (The notation $\overline{c_l}$ denotes the complement of bit c_l.) At each time step, each switch is permitted to transmit one packet along each of its outgoing edges.

In a butterfly network, packets are typically sent from the inputs on level 0 to the outputs on level $\log N$. If each input sends a single packet, we say that the network is *lightly loaded.* A specific type of routing problem of interest is the *permutation routing problem.* In a permutation routing problem, each input of the butterfly sends exactly one packet to some output of the butterfly and each output receives exactly one packet from some input of the butterfly. If each input sends $\log N$ packets, we say that the network is *fully loaded.* One of the nice properties of the butterfly is that there is a unique path of length $\log N$ between any input and any output, and there is a simple rule for finding that path. When a packet with origin $\langle 0, a_0 \cdots a_{\log N-1}\rangle$ and destination $\langle \log N, d_0 \cdots d_{\log N-1}\rangle$ reaches level l, it passes through the node labeled $\langle l, d_0 \cdots d_{l-1} a_l \cdots a_{\log N-1}\rangle$. If $d_l = a_l$, then it takes the straight edge to $\langle l+1, d_0 \cdots d_l a_{l+1} \cdots a_{\log N-1}\rangle$. Otherwise, it takes the cross edge to the same node. This path selection algorithm is called *source oblivious* [BH85] because, at each node, the next edge taken by a packet depends only on its current location and its destination, and not on its source, or on the locations or paths taken by any of the other packets. All of the routing algorithms discussed in this chapter are source oblivious.

2.1.2 Queuing protocols

This chapter studies two broad classes of queuing protocols: greedy protocols and non-predictive protocols. Many easily implementable as well as conceptually simple queuing protocols like FIFO and fixed-priority scheduling are greedy as well as non-predictive. In a *greedy* queuing protocol, at each step, each switch with one or more packets in its queue selects a packet, and then sends it to the next level, unless the queue that the packet wishes to enter is already full. A switch is not prohibited from sending more than one packet at each step, provided that they use different edges. In a *non-predictive* queuing protocol [Lei92, Section 3.4.4][Ran87b], at each step, each switch selects one packet from its queue *without examining the destinations of any of the packets in its queue*, and sends the packet to the next level, unless the queue that it wishes to enter is full. If the queue is full, then the switch must select the same packet at the next step. The switch is not permitted to examine the destinations of any other packets until the selected packet has been successfully transmitted. Non-predictive protocols are also greedy.

2.1.3 Previous work

The first important butterfly routing algorithm is due to Batcher [Bat68], who showed that an N-input butterfly network can sort, and hence route, N-packets in $O(\log^2 N)$ steps.

The next breakthrough came more than a decade later when Valiant [Val82, VB81] observed that any permutation routing problem can be transformed into two random problems by routing the packets first to random intermediate destinations, and then on to their true destinations. He also showed that an N-node hypercube (or N-input butterfly) can route N packets to random destinations (or from random origins) in $O(\log N)$ time using queues of size $O(\log N)$, with high probability. As a consequence, the hypercube or butterfly can route any permutation in $O(\log N)$ time, with high probability.

Valiant's result was improved in a succession of papers by Aleliunas [Ale82], Upfal [Upf84], Pippenger [Pip84], and Ranade [LMRR, Ran87c]. All of these papers use Valiant's idea of first routing to random intermediate destinations. Aleliunas and Upfal increased the number of packets that can be routed in $O(\log N)$ time. They developed the notion of a *delay path* and showed how to route N packets on an N-node shuffle-exchange graph and $N \log N$ packets on an N-input butterfly network, respectively, in $O(\log N)$ steps, using queues of size $O(\log N)$. Pippenger devised an ingenious algorithm for routing with bounded size queues. He showed how to route $N \log N$ packets on a variant of the butterfly in $O(\log N)$ steps with queues of size $O(1)$. Finally, Ranade developed a simpler algorithm for routing with bounded queues that could also efficiently combine multiple packets with the same destination. As a consequence of Ranade's algorithm, it is possible to simulate one step of an $N \log N$-processor CRCW PRAM on an N-input butterfly in $O(\log N)$ steps. Neither Pippenger's algorithm nor Ranade's algorithm are greedy.

Recently, Stamoulis and Tsitsiklis [ST91] have analyzed the average delay and queue size in hypercubes and butterflies with unbounded queues in which packets with random destinations are generated according to a Poisson process. They show that if the load factor on the network is less than one, then the network is stable in the steady state, the average delay is $O(\log N)$, and the average queue size is $O(1)$.

Although the performance of greedy algorithms in butterflies with bounded queues has proven difficult to analyze, attempts have been made to approximately model [SS89] or empirically determine [Tsa90] their performance.

2.1.4 Our results

In Section 2.2 we show that for any greedy queuing protocol, a routing problem in which each input in an N-input butterfly sends a single packet to a randomly chosen output requires $O(\log N)$ steps, with high probability, provided that the queue size is a sufficiently-large fixed constant. Previously, only the trivial upper bound of $O(N)$ was known. An intriguing problem left open in this section is a bound on the number of steps taken by a greedy queuing protocol when the butterfly is fully-loaded.

In Section 2.3 we show that for any deterministic non-predictive queuing protocol, there exists a one-to-one routing problem (permutation) that requires $\Omega(N/q \log N)$ time to route, where q is the maximum queue size. Previously, no lower bound greater than $\Omega(\sqrt{N})$ was known. The $\Omega(\sqrt{N})$ bound is based on the congestion and is independent of the way the packets are scheduled. This section shows that greater delays can occur due to the way packets interact in the network.

Section 2.4 presents a simple, but non-greedy, algorithm for routing a random problem on a fully-loaded butterfly with bounded-size queues in $O(\log N)$ steps, with high probability. The algorithm is simpler than the previous algorithms of Ranade and Pippenger because it does not use ghost messages, it does not compare the ranks or destinations of packets as they pass through a switch, and it cannot deadlock.

Finally, in Section 2.5 we analyze routing algorithms that drop packets when there is contention. Examples of machines that drop packets are NEC's ATOM switch [SNS$^+$89] and the BBN Butterfly [BBN86]. The BBN Butterfly algorithm has been studied by Kruskal and Snir [KS83] and Koch [Koc88]. Koch showed that for a random problem the number of packets that succeed in locking down paths from their origins to their destinations is $\Theta(N/\log^{\frac{1}{q}} N)$, where q is the maximum number of packets that any wire can support. By routing the packets to randomly (but not independently) chosen intermediate destinations, we show that for *any fixed permutation* the expected number of packets that reach their destinations is $\Omega\left(N/\log^{\frac{1}{q}} N\right)$.

2.2 Greedy queuing protocols

In this section, we will study the performance of greedy queuing protocols. In Section 2.2.1, we analyze the average case behavior of any routing algorithm with a greedy queuing protocol. We show that if every input sends a packet to a randomly chosen output, then the time required for all of the packets to reach their destinations is $O(\log N)$. In Section 2.2.2, we show how any specific permutation routing problem on the butterfly can be routed in $O(\log N)$ steps using Valiant's idea of splitting a routing problem into two random routing problems.

2.2.1 Average case behavior

We first define a few terms. A *delay tree* is a rooted tree that is a subgraph of the butterfly. Its root is a level 0 node and the tree contains a (directed) path, which we call the *spine*, from the root to a node in level $\log N$ of the butterfly. The tree "grows out" from the spine and there is a unique directed path in the tree from the root to each node in the tree. A *full node* is defined to be a node through which the paths of at least q packets pass, where q is the maximum size of the queue in each node. Note that in the course of the routing, a full node may never have a full queue since the packets could arrive at different times. However a non-full node can never have a full queue. A *full delay tree* is a delay tree for which every node of the tree that is not on the spine is a full node. The number of *packets on a delay tree* is defined to be the sum over all nodes of the tree of the total number of packets passing through each node. Note that this number is different than the number of *distinct packets on a delay tree*. In the former, if a particular packet hits (i.e., passes through) many nodes of a tree it is counted many times in the sum. The significance of the above definitions becomes clear in Theorem 2.2.1 below.

Theorem 2.2.1 *The maximum delay of any packet is less than or equal to the maximum of the number of packets on a full delay tree.*

Proof: Let the path of some packet p, from its source to destination, be denoted by P. Now consider the maximal full delay tree with the path P as its spine and the source of the packet p as its root. We will refer to this maximal full delay tree as the maximal full delay tree of packet p. We will bound the delay of p by the number of packets on its maximal full delay tree. Since the tree is maximal, every non-tree node that is a neighbor of a tree node is not a full node. We will now show that at each time step t until packet p reaches its destination, some packet in its maximal full delay tree moves. At every time step t there are 3 cases.

a. The packet p moves.

b. Some other packet queued at the same node as p moves.

c. No packet queued at the same node as p moves.

The first two cases are simple. Since the queuing protocol is greedy, case **c** necessarily means that the packet selected by the node to be sent at time step t could not move because the queue in the node, say n, that it wanted to enter was full. Note that node n belongs to the maximal full delay tree since it has at least q packets passing through it. Now if some packet in node n moved at time step t we are done. If not we look at the packet selected by node n and repeat the argument again. Note that case **c** cannot apply at the leaves of tree since it does not have any neighbors with full queues. So we must encounter either case **a** or **b** before we leave the tree. Therefore the delay of packet p is at most the number of packets on its maximal full delay tree. Thus the maximum delay of any packet is at most the maximum of the number of packets on a full delay tree. □

We will use the following property of the butterfly network in the proofs in this section.

Observation 2.2.2 *A packet can enter a tree contained in the butterfly at exactly one point and once the packet leaves the tree it can never return to it.*

We also state without proof a Chernoff type bound [AV79] and [Rag90, p. 56] and a result due to Hoeffding [Hoe56].

Lemma 2.2.3 (Hoeffding) *Let X be the number of successes in r independent Bernoulli trials where the probability of success in the i^{th} trial is p_i. Let S be the number of successes in r independent Bernoulli trials where the probability of success in each trial is $p = \frac{1}{r}\sum_{1\leq i\leq r} p_i$. Then $E(X) = E(S) = rp$, and, for α such that $\alpha E(S) \geq E(S) + 1$,*

$$\Pr[X \geq \alpha E(X)] \quad \leq \quad \Pr[S \geq \alpha E(S)] \tag{1}$$

Lemma 2.2.4 *Let S be the number of successes in r independent Bernoulli trials where each trial has probability p. The $E(S) = rp$, and, for $\alpha > 2e$,*

$$\Pr[S \geq \alpha E(S)] \quad \leq \quad 2^{-\alpha E(S)} \tag{2}$$

Theorem 2.2.5 *Let constant q be the maximum queue size. Then the maximum delay of any packet is at most $\gamma \log N$ with probability at least $1 - \frac{1}{N}$, for sufficiently large but constant γ and q.*

Proof: We will show that if there is a packet with large delay, then there must be a delay tree with a large number of packets on it, which in turn we will show to be an unlikely event. Assume that some packet has a delay $\gamma \log N$ or more. Consider the maximal full delay tree of this packet. Let D denote the number of packets on this maximal full delay tree. By Theorem 2.2.1, we know that $D \geq \gamma \log N$. Also since every non-spine node of this delay tree is necessarily a full node, the maximum number of nodes of this maximal full delay tree is at most $\frac{D}{q} + \log N$. Let n be a node on any level l of the butterfly. The average number of packets passing through n is 1, because there are 2^l possible packets that can pass through this node and each of these packets has a probability of 2^{-l} of passing through it. Therefore the expected number of packets on this delay tree is at most $\frac{D}{q} + \log N$. The gist of the proof is to show that the number of packets on this tree is clustered around its mean and therefore the tree is unlikely to have D packets on it, for sufficiently large constants q and γ.

The number of *hits* made by a packet on a delay tree is the number of nodes of the tree through which the packet passes. Let us divide the hits on a delay tree into two types : b-hits (for big hits) which are hits made by packets that make at least c hits on the tree, and s-hits (for small hits) which are hits made by packets that make fewer than c hits on the tree, where c is some constant. It must be the case that either the total number of b-hits on some delay tree is greater than or equal to $\frac{D}{2}$ (call this event E_b) or the total number of s-hits on some delay tree is greater than or equal to $\frac{D}{2}$. The latter possibility also implies that there are at least $\frac{D}{2(c-1)}$ distinct packets hitting some delay tree (call this event E_s), since each packet making s-hits can make at most $c-1$ hits on the tree. Thus, the probability that some packet has delay d is at most $\Pr(E_b) + \Pr(E_s)$. The intuitive reason as to why b-hits are unlikely is as follows. If you imagine packets running backwards in time from destination to source, once a packet enters the tree, it can remain in the tree at the next step only if it takes the unique edge to its ancestor in the tree. So, at every step, it has approximately a probability of $\frac{1}{2}$ of making another hit. This exponentially decreasing probability for making more and more hits gives us the bound. Thus, this bound uses the tree structure in a crucial way, unlike the bound we will prove for E_s which is valid for any set of $\frac{D}{q} + \log N$ nodes.

Bounding the big hits: Let us suppose that event E_b occurs, i.e., there exists a delay tree of size at most $\frac{D}{q} + \log N$ with a total of at least $\frac{D}{2}$ b-hits. To bound the probability of this event we will enumerate all the possible ways it can happen. The maximum value that D can take is $N \log N$, since each packet can contribute at most $\log N$ hits and there is a total of N packets. Therefore, the number of ways of choosing D is at most $N \log N$. The number of ways of choosing the root for the delay tree is N. A binary tree of size at most $\frac{D}{q} + \log N$ can be represented by indicating the number of children (no children, left son only, right son only, both sons) in breadth-first-search order. Thus the total number of ways of choosing the delay tree is at most $N 4^{\frac{D}{q} + \log N}$. The number of different packets causing these b-hits is at most $\frac{D}{c}$, since each packet causes at least c hits and there are a total of at most D hits on the tree. Let us assume that there is some arbitrary fixed ordering of the nodes in the tree, e.g., the breadth-first-search ordering of the tree. We will now pick a nondecreasing sequence of $\frac{D}{c}$ nodes in the tree, $n_1, n_2, \cdots, n_{\frac{D}{c}}$. Note that each node of the tree can occur more than once in this sequence. Node n_i is the last node on the tree through which the i^{th} packet passed. The number of ways of choosing this sequence is at most

$$\binom{\frac{D}{q} + \log N + \frac{D}{c}}{\frac{D}{c}} = \binom{\frac{D}{q} + \log N + \frac{D}{c}}{\frac{D}{q} + \log N}$$

Let node n_i of the sequence be at level l_i of the butterfly. For every n_i, we now associate a non-negative integer h_i denoting the number of hits made by a packet p_i before leaving node n_i. The number of ways of distributing at most D hits over $\frac{D}{c}$ elements of the sequence is at most $\binom{D + \frac{D}{c}}{\frac{D}{c}}$. We can ignore n_i with $h_i = 0$ in this. Since the packet must have made exactly h_i hits before leaving the tree at n_i, the number of choices for p_i is $2^{l_i - h_i}$. Here we have used Observation 2.2.2. The total number of ways of choosing packets for all elements in the sequence is at most $\prod_i 2^{l_i - h_i}$. (We are overcounting a little since packets have to be distinct.) Now, we have chosen a particular tree, a sequence of nodes n_i and the associated packets p_i. The probability that all the packets p_i pass through their corresponding nodes n_i is simply the product of the probabilities that each individual packet p_i passes through node n_i which equals $\prod_i 2^{-l_i}$. (We can multiply probabilities because each packet chooses its path independently.) Putting it all together, we have

$$\begin{aligned}
\Pr(E_b) &\leq N\log N \cdot N4^{\frac{D}{q}+\log N} \cdot \binom{\frac{D}{q}+\log N+\frac{D}{c}}{\frac{D}{q}+\log N} \\
&\quad \cdot \binom{D+\frac{D}{c}}{\frac{D}{c}} \cdot \prod_i 2^{l_i-h_i} \cdot \prod_i 2^{-l_i} \\
&\leq N^5 2^{2\frac{D}{q}} \cdot \left(\frac{(\frac{D}{q}+\log N+\frac{D}{c})e}{\frac{D}{q}+\log N}\right)^{\frac{D}{q}+\log N} \cdot \left(\frac{(D+\frac{D}{c})e}{\frac{D}{c}}\right)^{\frac{D}{c}} \cdot 2^{-\sum_i h_i} \\
&\leq 2^{5\frac{D}{\gamma}} \cdot 2^{2\frac{D}{q}} \cdot \left(\left(1+\frac{q}{c}\right)e\right)^{\frac{D}{q}+\frac{D}{\gamma}} \cdot 2^{\log((c+1)e)\frac{D}{c}} \cdot 2^{-\frac{D}{2}} \qquad (3)
\end{aligned}$$

since $D \geq \gamma \log N$ and $\sum_i h_i \geq \frac{D}{2}$. Note that the multiple of D in the exponent of the first four factors decreases with an increase in the values of c, q and γ. So for some suitably large values for the constants c, q and γ the expression in Equation 3 is at most 2^{-kD} for constant $k > 0$. Now by increasing the value of γ further if necessary we can use the fact that $D \geq \gamma \log N$ to bound the value of this expression (and hence $\Pr(E_b)$) to be at most $\frac{1}{2N}$.

Bounding the small hits: Let us suppose event E_s occurs, i.e. there is a tree of size at most $\frac{D}{q} + \log N$ with at least $\frac{D}{2(c-1)}$ different packets hitting the tree, for some value of $D \geq \gamma \log N$. The number of ways of choosing a value for D is at most $N \log N$. The number of ways of choosing such a tree is at most $N4^{\frac{D}{q}+\log N}$. Let X denote the total number of distinct packets hitting a tree of size at most $\frac{D}{q} + \log N$. X is a sum of N boolean random variables, $X_i, 1 \leq i \leq N$. Each X_i is 1 if the packet originating at input i hits the tree and 0 otherwise. The expected number of distinct packets on the tree is at most the expected number of packets on the tree. Therefore, $E(X) \leq \frac{D}{q} + \log N$. Using Lemmas 2.2.3 and 2.2.4, we have

$$\begin{aligned}
\Pr(E_s) &= N\log N \cdot N4^{\frac{D}{q}+\log N} \cdot \Pr\left(X \geq \frac{D}{2(c-1)}\right) \\
&\leq N\log N \cdot N4^{\frac{D}{q}+\log N} \cdot 2^{-\frac{D}{2(c-1)}} \qquad (4)
\end{aligned}$$

as long as $\alpha = \frac{\frac{D}{2(c-1)}}{E(X)} > 2e$. Using the fact that $D \geq \gamma \log N$, we have

$$\alpha \geq \frac{\frac{D}{2(c-1)}}{\frac{D}{q}+\log N} \geq \frac{q\gamma}{2(c-1)(q+\gamma)} \qquad (5)$$

Let c_0, q_0, and γ_0 be values of c, q, and γ respectively for which $\Pr(E_b)$ was shown to be at most $\frac{1}{2N}$. We choose the values of c, q, and γ such that *both* $\Pr(E_b)$ and $\Pr(E_s)$ are at most $\frac{1}{2N}$ as follows. First we choose $c = c_0$. Next we choose constants q and γ such that

$q = \gamma$. Let τ denote the value of q and γ. We choose τ such that $\tau \geq \max(q_0, \gamma_0)$. We make $\alpha > 2e$ by choosing τ large enough such that the right-hand side of Equation 5 which equals $\tau/4(c-1)$ is greater than $2e$. Furthermore, τ is chosen large enough such that

$$4^{\frac{D}{q}} \cdot 2^{-\frac{D}{2(c-1)}} = 4^{\frac{D}{\tau}} \cdot 2^{-\frac{D}{2(c-1)}} \leq 2^{-j\frac{D}{c-1}},$$

for some constant $j > 0$. Since $D \geq \gamma \log N$,

$$2^{-j\frac{D}{c-1}} \leq 2^{-j\frac{\gamma \log N}{c-1}} = 2^{-j\frac{\tau \log N}{c-1}}$$

Finally, the value of τ is made large enough such that

$$\Pr(E_s) \leq N \log N \cdot N 4^{\log N} \cdot 2^{-j\frac{\tau \log N}{c-1}} \leq \frac{1}{2N}$$

Note that since $c = c_0$, and $q = \gamma = \tau \geq \max(q_0, \gamma_0)$, $\Pr(E_b)$ is at most $\frac{1}{2N}$ for the chosen values of c, q, and γ. It now follows that the probability that a packet has delay d greater than $\gamma \log N$ is at most $\frac{1}{2N} + \frac{1}{2N}$ which equals $\frac{1}{N}$. □

2.2.2 Routing a fixed permutation

The results of Section 2.2.1 deal with the routing delay of an average routing problem. What can we say about routing a fixed permutation? We can show that we can route any fixed permutation in $O(\log N)$ steps with high probability using Valiant's idea of routing in two phases. In Phase A, each packet is routed from its source in level 0 to a random intermediate destination in level $\log N$. For simplicity, we will assume that the butterfly network has wrap-around, i.e., each node in level $\log N$ is identified with the node in level 0 in its row. The packets are queued up at the end of Phase A and in Phase B each packet is routed to its actual destination.

Theorem 2.2.6 *Any fixed permutation can be routed using Valiant's paradigm such that the delay is $O(\log N)$ with probability $\geq 1 - \frac{2}{N}$.*

Proof: Phase A is precisely the same problem as that studied in Section 2.2.1. In Phase B, each packet is routed from its intermediate destination to its final destination. For convenience, we will denote the level of its final destination as 0 and that of the intermediate destination as level $\log N$. This phase is a little different from the one we studied in that the starting points are random while the destinations are fixed. But the same proofs for the delay will work with small modifications. It is perhaps best to imagine the packets running backwards from level 0 (final destinations) to random nodes in level $\log N$ (intermediate destinations). In the proof for bounding the b-hits, the sequence n_i will now represent switches through which packets that hit the tree *entered* the tree (running backwards in time). The number of ways of associating a packet with n_i in level l_i is 2^{l_i}. The probability that the packet makes h_i hits is now $2^{-(l_i+h_i)}$, since it must leave the tree at the unique ancestor of n_i in level $l_i - h_i + 1$. The rest of the calculation is the same as before. The proof for bounding the s-hits is identical. □

2.3 Difficult permutations

In this section, we prove that for any deterministic non-predictive queuing protocol, there exists a permutation that requires $\Omega(N/q \log N)$ time to route on a butterfly network. Previously, the best lower bound for routing on a butterfly with queues of any size was $\Omega(\sqrt{N})$. The $\Omega(\sqrt{N})$ bound is proved by observing that certain permutations, such as the bit-reversal permutation, force $\Omega(\sqrt{N})$ packets to pass through a single switch [Lei92, Section 3.4.2]. (It is also not very difficult to prove that if the queue size is not bounded, then $O(\sqrt{N})$ is an upper bound on the time to route any permutation using any greedy protocol.) Because the $\Omega(\sqrt{N})$ bound is based on congestion only, it applies to any queuing protocol. The results in this section indicate that the manner in which packets are scheduled can potentially cause much greater delays. The proof involves a careful examination of the interaction of the packets as they route through the network.

To simplify the presentation in this section, we will assume that each switch has a single queue, and that, at each step, its two neighbors at the previous level may each send a packet into the queue provided that, at the beginning of the step, the queue held at most q packets. We call q the *queue threshold* of the switch. Since a queue can receive 2 packets when it already has q, it may contain as many as $q+2$ packets, but no more.

Theorem 2.3.1 *For any deterministic non-predictive queuing protocol, there exists a permutation π that requires $\Omega(N/q \log N)$ steps to route on a butterfly with queue threshold q.*

Proof: The proof is by induction. We will assume that there are two edges leading into each butterfly input, and we begin by computing the time, $t_d(r)$, required for a depth d butterfly to accept $r/2$ packets on each of the 2^{d+1} edges into its inputs. (For simplicity, we assume without loss of generality that r and q are even.) We will assume that at time step 1 and at each time step thereafter, 1 packet is available for transmission along each of these edges until $r/2$ packets have crossed the edge. Furthermore, we will assume that each output switch can transmit one packet at each step.

We begin by examining a 1-input butterfly, which consists of a single switch, s. Suppose that at the beginning of time step 1, the queue at switch s is empty. We would like to know how long takes for s to receive $r/2$ packets from each of its incoming edges, where $r > q$. On time steps 1 through q, s receives one packet along each of its two incoming edges. During steps 2 through q, s transmits one packet at each step. Thus, after q steps, $2q$ packets have been received, $q-1$ have been transmitted, and the queue contains $q+1$ packets. Since the queue is full, s does not receive any packets on step $q+1$, but it does transmit one. Thereafter, s receives two packets on every other step, and transmits one packet on every step, until a total of r packets have been received, which occurs on step $q+(r-2q) = r-q$. Thus, $t_0(r) = r-q$.

Next, let us compute the time required for each input of a depth-d butterfly to receive $r/2$ packets along each of its incoming edges. In order for an input to receive r packets, it must transmit at least $r-(q+2)$ packets. Using the assumption that the queuing protocol is nonpredictive, we will choose the paths of these $r-(q+2)$ packets so as to maximize the delay. Since a switch cannot look at a packet's destination until it has been selected for transmission, we can wait until a packet has been selected, and then decide if it should take a cross edge or a straight edge to the next level. The first $(r-(q+2))/2$ packets selected by each input switch $\langle 0, c_0c_1\cdots c_{d-1}\rangle$ will be sent to the switches labeled $\langle 1, 0c_1\cdots c_{d-1}\rangle$. These switches are the inputs of a depth-$(d-1)$ sub-butterfly. The second $(r-(q+2))/2$ packets will be sent to the depth-$(d-1)$ sub-butterfly whose inputs are labeled $\langle 1, 1c_1\cdots c_{d-1}\rangle$.

The inputs of the first sub-butterfly start accepting packets at step 2. By induction, the time required for each input to receive $r-(q+2)$ packets is $t_{d-1}(r-(q+2))$. Thus, the first sub-butterfly receives packets during steps 2 through $t_{d-1}(r-(q+2))+1$. In the meantime, no packets are sent to the inputs of the second sub-butterfly. The first packets arrive there on step $t_{d-1}(r-(q+2))+2$, and continue to arrive until step $2t_{d-1}(r-(q+2))+1$, at which point each input has received $r-(q+2)$ packets. Thus, $t_d(r) = 2t_{d-1}(r-(q+2))+1$. Solving this recurrence yields

$$\begin{aligned} t_d(r) &\geq 2^d t_0(r-(q+2)d) \\ &\geq 2^d(r-(q+2)(d+1)). \end{aligned}$$

The lower bound on $t_d(r)$ that we have just derived requires $r > (q+2)(d+1)$ packets to pass through each butterfly input. In a permutation routing problem, however, only one packet originates at each input. In order to use the bound, we will force r packets through each input of an N/r^2-input sub-butterfly that spans levels $\log r$ through $\log N - \log r$. We call this sub-butterfly the *busy sub-butterfly*. It has depth $d = \log N - 2\log r$. Each input of this sub-butterfly is the root of a depth-$\log r$ complete binary tree whose leaves are butterfly inputs on level 0. Call these trees the *input trees*. Each output is the root of a $\log r$-depth complete binary tree whose leaves lie on level $\log N$. Call these trees the *output trees*. All of these trees are completely disjoint. The r packets that originate at the leaves of an input tree will all be sent through the root of that tree. Each output of the busy sub-butterfly receives exactly r packets. These packets are distributed among the r leaves of the corresponding output tree so that they each receive exactly one packet. Note that between levels $\log r$ and $\log N - \log r$, the only switches and edges used for routing are those in the busy sub-butterfly.

All that remains is to choose appropriate values of r and d. From the construction of the busy sub-butterfly, we know that $d = \log N - 2\log r$. In order for our lower bound on t_d to be greater than zero, we need $r > (q+2)\ (d+1)$. Choosing $r = 2(q+2)(\log N + 1)$ yields $t_d(r) \geq 2^d(q+2)(\log N+1) = (N/r^2)(q+2)(\log N+1) = N/(4(q+2)(\log N+1))$. Thus the delay is $\Omega(N/q\log N)$. □

Note that the maximum number of packets passing through any node (the congestion) for the worst-case permutation constructed in this section is only $O(\log N)$. This implies that there are other more complex routing algorithms like that of Ranade [Ran87c] which can route this permutation in $O(\log N)$ steps!

2.4 A simple routing algorithm

In this section we present a simple, but non-greedy, algorithm for routing on butterfly networks. With high probability, the algorithm requires $O(k + \log N)$ time to route packets with random destinations, where k is the number of packets that originate at each input. The algorithm is simpler than the algorithms of Pippenger [Pip84] and Ranade [Ran87c] because it does not use ghost messages, it does not compare the ranks or destinations of packets as they pass through a switch, and it cannot deadlock. Unlike the algorithm of Ranade, however, it does not combine packets with the same destination.

The routing algorithm begins by breaking the packets into *waves*. Each input contributes one packet to each wave. The waves of packets are separated by waves of *tokens*. Unlike the ghost messages in Ranade's algorithm, a token carries no information other than its type, which requires $O(1)$ bits to represent.[1] Initially, there are k packets at each input and a token is placed between each pair of successive packets, and after the last packet. For $0 \leq i \leq k-1$, the ith packet at each input is assigned to wave $2i$, and the ith token is assigned to wave $2i+1$. Thus, the packets belong to the even waves, and the tokens to the odd waves. Throughout the course of the routing, the algorithm maintains the following important invariant. For $i < j$, no packet or token in the jth wave leaves a switch before any packet or token in the ith wave. Furthermore, packets within the same wave pass through a switch in the increasing order of their row numbers of origin. (A row $c_0 \cdots c_{\log N-1}$ is viewed as a binary number where c_0 is the *lower* order bit.)

A switch labeled $\langle l, c_0 \cdots c_{\log N-1} \rangle$ has two edges into it, one from the switch labeled $\langle l - 1, c_0 \cdots c_{l-2} 0 c_l \cdots c_{\log N-1} \rangle$, and the other from the switch labeled $\langle l-1, c_0 \cdots c_{l-2} 1 c_l \cdots c_{\log N-1} \rangle$. We call the first edge the *0-edge*, and the other the *1-edge*. At the end of each of these edges is a first-in first-out queue that can hold q packets. We call these queues the 0-queue and the 1-queue, respectively.

The behavior of each switch is governed by a simple set of rules. By "forward" a packet or token we mean send it to the appropriate queue in the next level. If that queue is full, the switch tries again in successive time steps until it succeeds. A switch can either be in 0-mode or in 1-mode and is initialized to be in 0-mode. In 0-mode, a switch forwards packets in the 0-queue in FIFO fashion, until a token is at the head of the 0-queue. It then changes to 1-mode. In 1-mode, a switch forwards packets in the 1-queue in FIFO fashion, until a token is at the head of the 1-queue as well. Now the switch waits until both the queues at its outgoing edges have room to receive a token and then simultaneously sends one token to each of them. After doing this, the switch changes back to the 0-mode.

Note that at each step a switch is required to perform only $O(1)$ bit operations in order to determine which packet, if any, to send out. In the algorithms of Pippenger and Ranade, the switches must perform more complicated operations, such as comparing the destinations of two packets as they pass through a switch. In the succeeding sections, we show that our algorithm requires $O(k + \log N)$ steps, which is asymptotically optimal.

[1]Tokens are used in a similar fashion in a bit-serial algorithm for routing on the hypercube in [ALMN91]. It turns out, however, that tokens are not really needed in that algorithm. Ranade's proof of the equivalence of different queuing disciplines [Ran87b] implies that a first-in first-out queuing protocol will suffice.

2.4.1 Delay sequences

The proof that the algorithm requires $O(k + \log N)$ time uses a delay sequence argument similar to those in [ALMN91, LMRR, Ran87c]. A (w, f)-delay sequence consists of four components: a path P from an output to an input; a sequence $s_1, \ldots, s_w$ of w, not necessarily distinct, switches which appear in order on the path; a sequence $h_1, \ldots, h_w$ of w distinct packets and tokens; and a non-increasing sequence of wave numbers $r_1, \ldots, r_w$. The path P may trace any edge of the network in either direction. When the path traces an edge from some level l to level $l+1$, we call the edge a *forward* edge. The number of forward edges in the path is denoted by f. The length, L, of P is equal to the distance from an output to an input ($\log N$) plus two times the number of forward edges on P, $L = \log N + 2f$. We say that a delay sequence *occurs*, if, for $1 \leq i \leq w$, packet or token h_i belongs to wave r_i, and passes through switch s_i. The following lemma shows that if some packet is delayed, then a delay sequence must have occurred.

Lemma 2.4.1 *If some packet arrives at its destination at step* $\log N + d$ *or later, then a* $(d + (q-2)f, f)$*-delay sequence must have occurred, for some* $f \geq 0$*. Furthermore, no two tokens in the sequence belong to the same wave.*

Proof: Before we begin the proof, we need some definitions. Let the *lag* of a switch s at time t on level l be $t - l$. Also, let the *rank* of a packet h be a 2-tuple consisting of h's wave number and the row number of the input in which it originated. Ranks are compared by first comparing wave numbers, and then, if there is a tie, comparing row numbers. A row $c_0 \ldots c_{\log N - 1}$ is viewed as a binary number where c_0 is the low order bit. Note that each packet has a distinct rank. Every token belonging to the same wave has the same rank. This rank is strictly less than all the packets in the wave above it but strictly greater than the packets in the wave below it. Note that ranks are used only as a tool for the analysis and not by the algorithm itself.

The algorithm maintains several important invariants. As mentioned before, the packets and tokens leave each switch in order of non-decreasing wave number. Furthermore, each edge transmits exactly one token from each odd wave. Finally, within an even wave, the packets that arrive at a switch via its 0-edge have smaller ranks than the packets that arrive via its 1-edge. As a consequence, each switch sends out packets and tokens in order of strictly increasing rank.

The delay sequence begins with the last packet to arrive at its destination. Suppose that some packet h_1 arrives at its destination, s_1, at step τ_1, where $\tau_1 \geq \log N + d$. Then s_1 has lag at least d at step τ_1. We will construct the delay sequence by spending lag points. We begin the sequence with h_1, s_1, and r_1, where r_1 is the wave number of h_1. Next, we follow h_1 back in time until the step at which it was last delayed.

In general, suppose that we have followed some packet or token h_i back in time from some switch s_i at time step τ_i until it was last delayed, at some switch s'_{i+1} at time step τ_{i+1}. As we follow h_i back in time, the nodes that h_i passes through are added to path P. Because h_i is delayed at s'_{i+1} at step τ_{i+1}, the lag at s'_{i+1} at step τ_{i+1} is one less than the lag of s_i at step τ_i. There are three possible reasons for the delay of h_i at switch s'_{i+1}.

First, if s_{i+1} selects another packet or token, h_{i+1}, to send instead of h_i, then h_{i+1} must have a strictly smaller rank than h_i. In this case, h_{i+1}, $s_{i+1} = s'_{i+1}$, and r_{i+1} are inserted into the sequence, where r_{i+1} is the wave number of h_{i+1}. We then follow h_{i+1} back in time until it was last delayed. We have spent one lag point, and inserted one packet or token into the sequence.

Second, if s'_{i+1} doesn't send h_i because the queue at the end of one of its outgoing edges is full, then we extend the path, P, forward along that edge to the switch at its head, s''_{i+1}. The lag of switch s''_{i+1} at time τ_{i+1} is two less than the lag of s_i at step τ_i. However, the queue must contain a total of q packets and tokens, all of which have smaller rank than h_i. We insert these packets and tokens into the sequence. We then follow the packet or token at the head of the queue back in time until it was last delayed.

If neither of these cases is true, it must be the case that in switch s'_{i+1} at time τ_{i+1} either of the following occurs.

(a) h_i is a packet and it is at the head of the 1-queue and the 0-queue is empty, or

(b) h_i is a token and it is at the head of one of the queues and the other queue is empty.

In either case, we go back to the switch at the tail of the empty queue at the previous time step. Note that we do not loose any lag by this process. We continue to do this as long as we can find an empty queue at the current switch. Suppose we do it m times and we are at a switch s''_{i+1} at time $\tau_{i+1} - m$. Switch s''_{i+1} has packets or tokens at the heads of both of its queues but did not send anything through one of its edges at time $\tau_{i+1} - m$. If one of the heads of its queues is a packet, we add it and switch s''_{i+1} to the delay sequence and continue to follow this packet back in time. Note that in case (a), this packet belongs to the same wave as h_i but has rank strictly less than h_i since the first edge we followed back from s'_{i+1} is a 0-edge. In case (b), the packet belongs to a wave earlier than that of token h_i and hence has a strictly smaller rank. In either case, we have added a packet of strictly smaller rank for the cost of one lag point. Now suppose that both the heads of queues are tokens. The only reason the tokens were not sent at time $\tau_{i+1} - m$ is that one of the outgoing edges of s''_{i+1} has a full queue. In this case we extend the path P forward to the switch at the head of the queue, and insert all of the packets and tokens in that queue into the delay sequence and follow the packet or token at the head of the queue back in time. Now we have added q packets and tokens for the cost of two lag points.

For each lag point spent, at least one new packet or token is inserted into the delay sequence. Furthermore, for each forward edge on the path P, an additional $q - 2$ packets and tokens are inserted. Let f be the number of forward edges on P. Since we had d lag points to spend, we must insert at least $d + (q - 2)f$ packets and tokens. Since we are inserting packets or tokens in strictly decreasing order of rank, at most k of these can be tokens. The length of P is $\log N + 2f$. □

2.4.2 Bunched delay sequences

We have now established that if some packet is delayed, then a delay sequence occurs. To simplify the rest of the argument, we will restrict our attention to delay sequences in which the packets can be partitioned into *bunches* of size b such that all of the packets in each bunch pass through the same switch on the sequence and have the same wave number. We call such a delay sequence a *bunched* delay sequence. The following lemma shows that if a delay sequence occurs, then a bunched subsequence also occurs.

Lemma 2.4.2 *If a $(d+(q-2)f, f)$ delay sequence occurs, then a (bg, f) bunched delay sequence occurs, where*

$$g = \left\lceil \frac{d+(q-2b)f - bk - (b-1)\log N}{b} \right\rceil .$$

Proof: Suppose that a $(d+(q-2)f, f)$ delay sequence occurs. We will describe an algorithm for finding a bunched subsequence.

Starting at the first switch on the sequence, s_1, form a bunch of size b of packets with wave number $2(k-1)$. If successful, then form another bunch of packets with wave number $2(k-1)$. Otherwise, if there are fewer than b remaining packets with wave number $2(k-1)$, then there are two cases to consider. First, if there are other packets on the sequence that pass through s_1, then discard the remaining packets with wave number $2(k-1)$, and begin forming bunches out of packets with the next smaller even wave number. Since the wave number can decrease at most k times, this case can happen only k times. Each time, we may discard as many as $b-1$ packets from the original delay sequence. Second, if no other packets on the sequence pass through s_1, then move on to the second switch, s_2. This case can happen at most $\log N + 2f$ times, since the path has length $L = \log N + 2f$. As in the first case, we may discard $b-1$ packets from the original sequence.

Since the original sequence contains at least $d+(q-2)f-k$ packets, and we discard a total of at most $k(b-1)+(\log N+2f)(b-1)$ packets, at least $d+(q-2b)f-bk-(b-1)\log N$ packets are placed in bunches. Thus, there are at least $g = \left\lceil \frac{d+(q-2b)f-bk-(b-1)\log N}{b} \right\rceil$ bunches. $\square$

2.4.3 The counting argument

We are now in a position to prove that, with high probability, every packet reaches its destination within $O(k+\log N)$ steps.

Theorem 2.4.3 *For any c_2, there exists constants c_1 and $q > 0$ such that the probability that any packet is delayed for more than $d = c_1(k+\log N)$ steps is at most $1/N^{c_2}$, where k is number of packets per input of the butterfly.*

Proof: To prove this theorem we will enumerate all possible bunched delay sequences, and show that it is unlikely that any of them occurs.

The number of different bunched delay sequences is at most

$$N \cdot 4^L \cdot \binom{L+g}{g} \cdot \binom{g+k}{g} \cdot \prod_{i=1}^{g} \binom{2^{d_i}}{b}, \tag{6}$$

where d_i is the level of the switch through which the ith bunch passes. The factors in this product are explained as follows. There are N choices for the output switch at which the path P in the delay sequence originates. At each of the first L switches on the path, there are at most 4 choices for the next switch on the path. There are at most $\binom{L+g}{g}$ ways of choosing the g (not necessarily distinct switches) on the path that the g bunches pass through, and at most $\binom{g+k}{g}$ ways of choosing (not necessarily distinct) wave numbers for the g bunches. Finally, given a switch with level d_i, and wave number w, there are $\binom{2^{d_i}}{b}$ ways of choosing b packets with wave number w that can pass through the switch.

Whether or not a particular delay sequence occurs depends entirely on the random destinations chosen by the packets in the delay sequence. It is important to note that every packet on the delay sequence is distinct. Therefore the events regarding any two packets on the delay sequence are independent. Thus the probability that all of the packets pass through their corresponding switches is $\prod_{i=1}^{g} \frac{1}{2^{bd_i}}$, since each of the b packets in the ith bunch has probability $1/2^{d_i}$ of passing through any particular switch on level d_i.

We can bound the probability that *any* delay sequence occurs by summing the probabilities of each individual delay sequence occurring, which is equivalent to multiplying (6) by $\prod_{i=1}^{g} \frac{1}{2^{bd_i}}$. Using the inequality $\binom{x}{y} \leq (ex/y)^y$ to bound $\binom{L+g}{g}$ and $\binom{g+k}{g}$, and using $\binom{x}{y} \leq x^y/y!$ to bound $\binom{2^{d_i}}{b}$, the product is at most

$$2^{3\log N+4f} \cdot (e(L+g)/g)^g \cdot (e(g+k)/g)^g \cdot (1/b!)^g,$$

where $g = \left\lceil \frac{d+(q-2b)f-(b-1)k-(b-1)\log N}{b} \right\rceil$. First, we choose b such that $b! \geq 16e^2$. By making q large compared to b (but still constant), and d large compared to $b(k+\log N)$ (but still $c_1(k+\log N)$, where c_1 is a constant), we can make g larger than $L = \log N + 2f$, k and $3\log N + 4f$. In this case, the product is at most $(8e^2/b!)^g \leq 2^{-g}$. By making g large enough, we can make this product less than $1/N^{c_2}$, for any constant c_2. □

2.5 Algorithms that drop packets

In this section, we consider queuing protocols that resolve contention by dropping packets. Two examples of machines that use this kind of protocol are the BBN Butterfly [BBN86] and the NEC ATOM switch [SNS+89].

The ATOM switch routes packets in a store-and-forward manner. At every time step, each switch examines the head of its input queue and forwards a packet to the appropriate output queue. If a queue receiving packets is already full then it discards packets in excess of its maximum queue size. The BBN Butterfly operates in the circuit-switching paradigm. Each packet tries to lock down a path between its source and destination. We will assume that each edge of the butterfly can sustain up to q such paths. This means that some packets may not be able to lock down paths to their destinations. In both of these queueing protocols, a natural question to ask is how many of the packets reach their destinations. The ATOM switch has not been studied in this context before. Kruskal and Snir [KS83] and Koch [Koc88] studied the average case performance of the BBN Butterfly algorithm. Koch showed that if each packet independently chooses a random destination then the expected number of packets that get through is $\Theta(N/\log^{\frac{1}{q}} N)$. However there are permutations that arise from natural problems in which only $O(\sqrt{N})$ packets get through. To combat this we show how to route any *fixed* permutation in either of the above mentioned queuing protocols such that the expected number of packets that reach their destinations is $\Omega\left(N/\log^{\frac{1}{q}} N\right)$. As an aside, this section also provides an elementary proof of the fact that the expected number of packets that get through for a random routing problem on the butterfly is $\Omega\left(N/\log^{\frac{1}{q}} N\right)$. As mentioned earlier, this was first proved by Koch [Koc88].

The idea for routing any fixed permutation is based on Valiant's idea of routing to random intermediate destinations. Consider two back-to-back butterflies, i.e. two butterflies whose level $\log N$ nodes are identified. The source of the packets is level 0 of one of the butterflies and each packet has a destination in level 0 of the other butterfly. In the first stage, the packets route from level 0 to level $\log N$ of the first butterfly. In the second stage of the routing, the packets route from level $\log N$ to level 0 of the second butterfly. In the first stage we use a scheme for sending packets to random but not independent destinations. Ranade[Ran87a] was the first to use this scheme in order to reduce the amount of the randomness needed to send packets to intermediate destinations in a packet switching algorithm. The scheme is as follows. At time step i every level i switch receives two packets, one from each of its incoming butterfly edges. The switch selects a random outgoing edge for one of the packets and routes the other packet through the remaining outgoing edge. Therefore in the first stage no packets are dropped. In the second stage, every packet is routed from this intermediate destination to its actual destination in level 0 of the second butterfly. In this stage, packets are dropped according to the rules of BBN Butterfly routing or that of the ATOM switch. We will assume that each packet picks a random rank from 1 to $r = \log^{\frac{1}{q}} N$. When packets need to be dropped, packets with the least rank are dropped in favor of those with a higher rank. We will now show that the expected number of packets that reach their destinations is $\Omega\left(N/\log^{\frac{1}{q}} N\right)$.

Let n be a node at level l of the second butterfly. Consider any k packets whose final destinations are reachable from this node. We bound the probability that all k packets pass through this node.

Lemma 2.5.1 *The probability that any k packets all pass through a node n in level l of the second butterfly is at most $\frac{1}{2^{lk}}$.*

Proof: Let the node in the first butterfly that corresponds to node n be n'. Let the sub-butterfly from level l to $\log N$ of the first butterfly that contains n' be B (See Figure 3). Note that the k packets pass through the given node n if and only if all of these packets pass through some node of sub-butterfly B. Consider the sources of the k packets in level 0 of the first butterfly and the unique shortest paths from each of the sources to sub-butterfly B. If any two of them intersect before reaching the sub-butterfly these two packets cannot both hit sub-butterfly B, since at the node of intersection only one packet can take the path to the sub-butterfly. If no two paths intersect, then the probability of each packet hitting B is independent of the others and equals $\frac{1}{2^l}$. Thus in this case the probability of all of

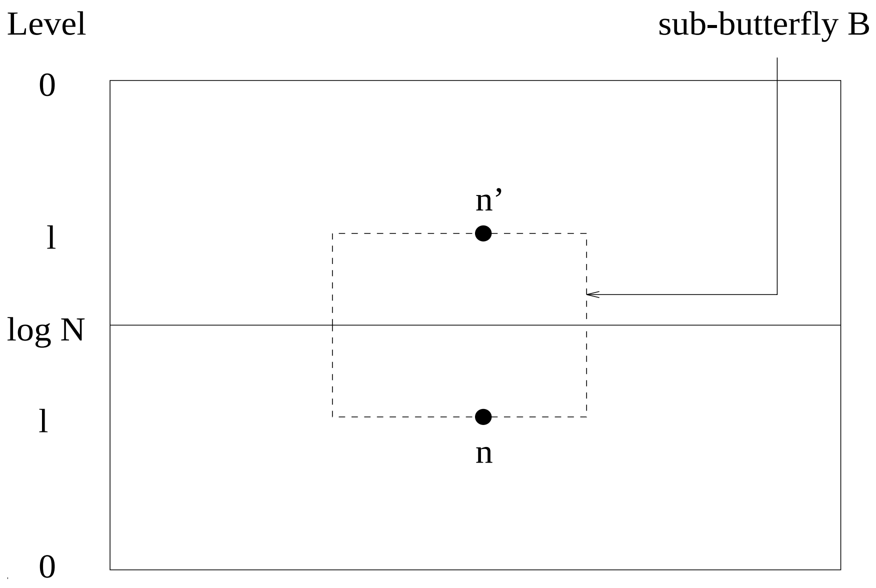

Figure 3: Sub-butterfly B.

them hitting the sub-butterfly is exactly $\frac{1}{2^{lk}}$. Therefore given any k packets the probability of all k of them passing through the given node or equivalently hitting the sub-butterfly is at most $\frac{1}{2^{lk}}$. □

Theorem 2.5.2 *The expected number of packets that reach their destinations is* $\Omega\left(N/\log^{\frac{1}{q}} N\right)$.

Proof: Consider the path of a particular packet in the second butterfly. We will now evaluate the probability that this packet reaches its destination. Note that with probability $\frac{1}{r}$ this packet will have the highest rank r. In this case, this packet can be dropped only if there is a node on its path with at least q packets going through it, all with rank r. We will now show a lower bound on the probability that there exist no such q packets. First let's bound E_q, the expected number of q-tuples of packets incident on a node at level l of the second butterfly.

$$E_q \leq \binom{2^l}{q} \frac{1}{2^{ql}} \leq \frac{1}{q!},$$

since there are $\binom{2^l}{q}$ ways of choosing q packets that can pass through a node on level l, and by Lemma 2.5.1, the probability that these packets actually pass through the node is at most $1/2^{ql}$.

The expected total number of q-tuples incident on some node on the path is at most $\log N/q!$, since the path has length $\log N$. The expected number of such q-tuples with all packets having rank r is at most $\log N/(q!r^q)$ which equals $1/q!$, since $r = \log^{\frac{1}{q}} N$. Since $1/q! \leq 1$, the probability that no such q-tuple exists anywhere on the path of the packet is at least $1 - 1/q!$. (A slightly larger choice for r would make the arguments work for $q = 1$.) This implies that the packet reaches its destination with a probability of at least $(1 - 1/q!)(1/r)$, since the probability that the packet gets rank r is $1/r$. Therefore the expected number of packets to reach their destinations is at least $(1 - 1/q!)(N/r)$ which is $\Omega\left(N/\log^{\frac{1}{q}} N\right)$. □

The proof that the expected number of packets that reach their destinations is $\Omega\left(N/\log^{\frac{1}{q}} N\right)$ also holds for a random routing problem in which each packet chooses independently a random destination. Lemma 2.5.1 is true because the probability that a packet passes through a node n in level l is $1/2^l$. Every packet chooses its path independently and hence the probability that all of them pass through the node exactly equals $1/2^{lk}$. The rest of the proof is the same as before. Koch [Koc88] has observed that the expected number of packets that get through is not affected by the rule that is used to decide which messages to keep and which messages to drop, as long as the destinations of the packets are not used to make this decision. Therefore, for this problem, random ranks are necessary only as a tool for analysis and any other non-predictive rule would exhibit the same average case behavior.

2.6 Open questions

The most vexing problem left open by this chapter is to determine the average number of time steps required to route on a fully-loaded N-input butterfly with constant-size FIFO queues. If fewer than $\Omega(\log N)$ packets may be queued at a node, then the only known upper bound on the time to route is $O(N \log N)$. This trivial upper bound is proven by showing that after $\log N$ steps, at least one packet arrives at the outputs at every time step until the routing is completed. Simulations show that the true time is closer to $O(\log N)$.

Another open question concerns the algorithm of Section 2.4 for routing on a fully-loaded butterfly with constant size queues. We know from Section 2.2 that a single wave of packets with random destinations can be routed using a greedy queuing protocol in $O(\log N)$ time, but when the waves are pipelined, as in Section 2.4, the analysis requires us to use a simple, but not greedy, protocol to route each wave. It would be interesting to show that even if each individual wave was routed with a greedy protocol, the total time to route $\log N$ waves was $O(\log N)$.

Chapter 3

Fault Tolerance of Hypercubic Networks

3.1 Introduction

In this chapter, we analyze the effect of faults on the computational power of hypercubic networks. Hypercubic networks are a loosely defined class of networks which includes commonly used networks such as the hypercube, shuffle-exchange network, butterfly, mesh of trees, and the fat-tree. Hypercubic networks have logarithmic diameter and all the networks that we mentioned excepting the hypercube have bounded degree. (A network is said to have bounded degree if the maximum number of edges out of any node is a constant independent of the network size.) The main objective of our work is to devise methods for circumventing faults in hypercubic networks using as little overhead as possible, and to prove lower bounds on the effectiveness of optimal methods. Of particular concern to us are the bounded-degree hypercubic networks that we mentioned earlier.

We will now summarize the relevant aspects of the fault model described in Section 1.2.1. We consider both the worst-case fault model and the random fault model, and we always assume that faulty components are totally disabled (e.g., a faulty node cannot be used to transport a packet of data through the network). We also assume that the faults in the network are static and detectable and that information concerning the location of faults can be used when reconfiguring the network to circumvent the faults. For simplicity, we restrict our attention to node failures since an edge fault can always be simulated by disabling the node at either end of the edge.

As described in Section 1.2.2, we will be primarily concerned with the amount by which a collection of faults can slow down some computation in the network. For example, if a butterfly network is being used for packet routing, we will be concerned with how much longer it takes a faulty butterfly to deliver all of the packets than it takes a fault-free butterfly to perform the same task. More generally, we will be interested in the length of time it takes an impaired network to emulate a single step of a fault-free network of the

This chapter describes joint research with Tom Leighton and Bruce Maggs[LMS92a, LMS92b].

same size and type. In particular, we define the *slowdown* caused by a set of faults in a network G to be the minimum value of S such that any computation that takes T steps on G when there are no faults can be performed in at most $S \cdot T$ steps on G when faults are present. One of our main goals will be to understand the relationship between slowdown and the number of faults for commonly-used networks. In particular, we will prove bounds on the number of faults that can be tolerated without losing more than a constant factor in speed.

We will use two methods for emulating a fault-free network G on an isomorphic but faulty network H. The first approach is to embed G into H so that the nodes of G are mapped to non-faulty nodes of H, and so that the edges of G are mapped to non-faulty paths in H. The goal is to find an embedding with minimum load, congestion, and dilation. The *load* of an embedding is the maximum number of nodes of G that are mapped to any single node of H. The *congestion* of an embedding is the maximum number of paths that pass through any edge e of H. The *dilation* of an embedding is the length of the longest path.

The load, congestion, and dilation of the embedding determine the time required to emulate each step of G on H. Every node of H has to emulate the computation of l nodes of G. Therefore, a computation step of G can be emulated on H in $O(l)$ steps. A communication step of G is emulated on H by routing packets along the paths in H that correspond to edges in G. Every packet can be delayed by at most c other packets at a node. Therefore every packet must reach its destination in $O(c \cdot d)$ steps. Therefore a communication step of G is emulated on H in $O(c \cdot d)$ steps. Thus, if there is an embedding of G in H with congestion c, load l, and dilation d, it is easy to see that H can emulate any computation of G with slowdown $O(l + c \cdot d)$. A stronger result due to Leighton, Maggs, and Rao [LMR88] is that it is possible for H to emulate any computation of G with a smaller slowdown of $O(l + c + d)$. Since we will be mostly interested in embeddings for which the congestion, load, and dilation are all constant (independent of the network size), we could use either result to obtain a constant-slowdown emulation of G on H.

In Section 3.2, we will show how to embed a fault-free N-input butterfly into an N-input butterfly containing $\log^{O(1)} N$ worst-case faults using constant congestion, load, and dilation. (In other words, we will show how to reconfigure an N-input butterfly around $\log^{O(1)} N$ worst-case faults so that the resulting degradation in performance is at most an $O(1)$ factor in speed.) A similar result will also be proved for an N-node mesh of trees. Hence, these networks can tolerate $\log^{O(1)} N$ worst-case faults with constant slowdown.

Previously, no connected bounded-degree networks were known to be able to tolerate more than a constant number of worst-case faults without suffering more than a constant-factor loss in performance. Indeed, it was only known that

1. any embedding of an N-node (2 or 3-dimensional) array into an array of the same size containing more than a constant number of worst-case faults must have more than constant load or dilation [GE84, KKL$^+$90], and

2. the N-node hypercube can be reconfigured around $\log^{O(1)} N$ worst-case faults with constant load, congestion, and dilation [AL91].

The embeddings that we use in Section 3.2 are level-preserving, i.e., nodes in a particular level of the fault-free network are mapped to nodes on the same level of the faulty network. We take a significant step towards proving the limitation of embedding techniques for the emulation of these networks by showing that no level-preserving embedding strategy with constant load, congestion, and dilation can tolerate more than $\log^{O(1)} N$ worst-case faults. Whether or not there is a natural low-degree N-node network (the hypercube included) that can be reconfigured using (not necessarily level-preserving) embedding techniques around more than $\log^{O(1)} N$ faults with constant congestion, load, and dilation remains an interesting open question.

In Section 3.3, we shift our attention to the routing capabilities of hypercubic networks containing faults. First we prove in Section 3.3.1 that an N-input butterfly with f worst-case faults can support an $O(\log N)$-step randomized packet routing algorithm for the nodes in $N - O(f)$ rows of the butterfly. The ability of the butterfly to withstand faults in this context is important because butterflies are often used solely for their routing abilities. Previously, it was known that expander-based multibutterfly networks can tolerate large numbers of worst-case faults without losing their routing powers [ALM90, LM92], but no such results were known for butterflies or other hypercubic networks. A corollary of this result is that an N-input butterfly with αN worst-case faults (for some small constant α) can support an $O(\log N)$-step randomized routing algorithm for a majority of its nodes. Note that the number of faults is optimal to within a constant factor, since it is possible to partition an N-input butterfly into components of size $O(\sqrt{N} \log N)$ with a total of N faults placed in level $(\log N)/2$. In Section 3.3.2 we show that butterflies with faults can also be used for circuit switching. In particular, we show that even if an N-input $O(1)$-dilated Beneš network contains $N^{1-\epsilon}$ worst-case faults (for any $\epsilon > 0$), there is still a set of $N - o(N)$ inputs I and a set of $N - o(N)$ outputs O such that for any one-to-one map $\pi : I \mapsto O$ it is possible to route edge-disjoint paths from i to $\pi(i)$ for all $i \in I$. This result substantially improves upon previous algorithms for fault-tolerant circuit switching in Beneš networks [OT71, SR80] which dealt with a constant number of faults by adding an extra stage to the network.

In Section 3.4, we use the fault-tolerant routing algorithm from Section 3.3.2 to show that an N-input butterfly with $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$) can emulate a fault-free butterfly of the same size with only constant slowdown. A similar result is proved for the shuffle-exchange network. These results are stronger than the reconfiguration results proved in Section 3.2 because the number of faults tolerated is much larger. The approach used in Section 3.4 differs from the embedding-based approaches in Section 3.2 in that a single node of the fault-free butterfly will be emulated by (possibly) several nodes in the faulty butterfly. Allowing redundant computation provides greater flexibility when embedding one network in another (thereby attaining greater fault tolerance) but also adds the complication of ensuring that replicated computations stay consistent (and accurate) over time. This technique was previously used in the context of (fault-free) work-preserving emulations of one network by another [Fel85, KLM+89, Sch90].

The techniques developed in Section 3.4 also have applications for hypercubes. For example, we can use the techniques to show than an N-node hypercube with $N^{1-\epsilon}$ worst-case faults can emulate any normal algorithm with constant slowdown. (The set of normal algorithms include FFT, bitonic sort, and other important ascend-descend algorithms [Lei92].) Previously, such results were known only for hypercubes containing $\log^{O(1)} N$ faults [AL91, BCS90, BCS92]. Whether or not an N-node hypercube can tolerate more than $\log^{O(1)} N$ faults with constant slowdown for general computations remains an important unresolved question.

In Section 3.5, we show that even if each node in an N-input butterfly fails independently with probability $p = 1/\log^{(k)} N$, where $\log^{(k)}$ denotes the logarithm function iterated k times, the faulty butterfly can still emulate a fault-free N-input butterfly with slowdown $2^{O(k)}$ with high probability. For $k = \Theta(\log^* N)$ the node failure probability is constant, and the slowdown is $2^{O(\log^* N)}$, which grows very slowly with N. Whether or not this result can be improved and whether or not there is a bounded-degree network that can sustain constant-probability random faults with only constant expected slowdown remain interesting open questions. It is likely that the methods in Chapter 4 can be used to tolerate constant-probability random faults in two- or higher-dimensional arrays with constant expected slowdown (See Conjecture 4.3.9). Until very recently, no results along these lines were known for the butterfly (unless routing is allowed through faulty nodes [Ann89], which simplifies matters substantially). Tamaki [Tam92a] has recently discovered an emulation scheme with slowdown $O((\log\log N)^{8.2})$. He has also introduced a class of bounded-degree networks called cube-connected arrays [Tam92b], and showed that an N-node network in this class with constant-probability random faults can emulate itself with expected slowdown approximately $\log\log N$. (These networks can also tolerate up to $\log^{O(1)} N$ worst-case faults with approximately $\log\log N$ slowdown.)

3.1.1 Additional previous work

There is a substantial body of literature concerning the fault-tolerance of communication networks. We do not have the space to review all of this literature here, but we would like to cite the papers that are most relevant. In particular, [AL91, Ann89, BCLR92, HLN87, HLN89, KKL+90, LSGH87, Rab89, Tam92b] show how to reconfigure a network around faults so that it can emulate a fault-free network of the same size and type. References [AAB+92, BCH91, DH90, DH91] show how to design a network H that will contain G as a subnetwork even if H contains some faults. Algorithms for routing messages around faults appear in [AS82, ALM90, AB91, HLN89, KKL+90, LM92, Lin92, Lyu90, OT71, Rab89, SR80]. The fault-tolerance of sorting networks is studied in [AU90, LMP91]. Finally, [BCS90, WC92, WCM91] show how to perform certain computations in hypercubes containing faults.

3.1.2 Network definitions

In this section, we will review the structure of some of the networks that we study in this chapter. For a description of the butterfly network, please refer to Section 2.1.1. Our results will hold whether or not the nodes on levels 0 and $\log N$ of each row are assumed to be the same node. Another popular network, the hypercube, is related to the butterfly. An N-node *hypercube* is obtained by labeling each node with a distinct $\log N$-bit binary number and connecting with an edge any two nodes whose binary numbers differ in exactly one bit. Another network closely related to the butterfly is the *shuffle-exchange network*. A $\log N$-dimensional shuffle-exchange network has N nodes, each node of which is labeled with a distinct $\log N$-bit binary number. Two nodes labeled u and v are linked by an edge if either u and v differ in precisely the last bit (exchange edge) or if u is a left or right cyclic shift of v (shuffle edge). Another popular interconnection network on which many fast parallel algorithms have been devised [Lei92] is the *mesh of trees network*. The nodes of the mesh of trees network are arranged in the form of an $N \times N$ grid with N rows and N columns. The nodes in each row or column form the leaves of a distinct complete binary tree.

A *circuit-switching network* is used to establish edge-disjoint paths (circuits) between its inputs and outputs. Each switch in a circuit-switching network has two incoming and two outgoing edges. The incoming edges can be connected to the outgoing edges in one of two ways: either crossing or straight through. The switches at the first level of the network are called the *input switches*. The switches at the last level are called the *output switches*. The two edges into each input switch are called *input edges*, or *inputs*. The two edges out of each output switch are called *output edges*, or *outputs*. By setting the switches, each input edge can be connected to an output edge via a path through the network. A circuit-switching network with N inputs and N outputs is said to be *rearrangeable* if for any one-to-one mapping π from the inputs to the outputs, it is possible to construct edge-disjoint paths in the network linking the i^{th} input to the $\pi(i)^{th}$ output, for $1 \leq i \leq N$. A classical example of a rearrangeable network is the Beneš network (See Figure 4). The Beneš network consists of back-to-back butterflies. A $\log N$-dimensional Beneš network has $2N$ inputs and $2N$ outputs. Its switches are arranged in $2\log N + 1$ levels of N switches each. The levels of a Beneš network are also referred to as stages. The first and last $\log N + 1$ levels each form a $\log N$-dimensional butterfly. Level $\log N$ is shared by these butterflies. We will refer to the nodes (switches) in level 0, $\log N$, and, $2\log N$ as the *input nodes*, *middle nodes*, and *output nodes* respectively. A *b-dilated* Beneš network is a network obtained by replacing each edge of the Beneš network by b parallel edges, and by replacing each 2×2 switch by a $2b \times 2b$ switch.

3.2 Emulation by Embedding

In this section, we show how to embed a fault-free binary tree, butterfly, or mesh of trees network into a faulty version of itself with constant load, congestion and dilation. As noted in the introduction, finding a constant load, congestion, and dilation embedding is the simplest way of emulating arbitrary computations of a fault-free network on a faulty version of itself with only constant slowdown. We will first consider embedding a complete binary tree in a complete binary tree with faults only at its leaves. This result will also hold for fat-trees [Lei85, GL89] with faults at the leaves. We use this result to find reconfigurations of butterflies and meshes of trees in which faults may occur at any node. The primary result

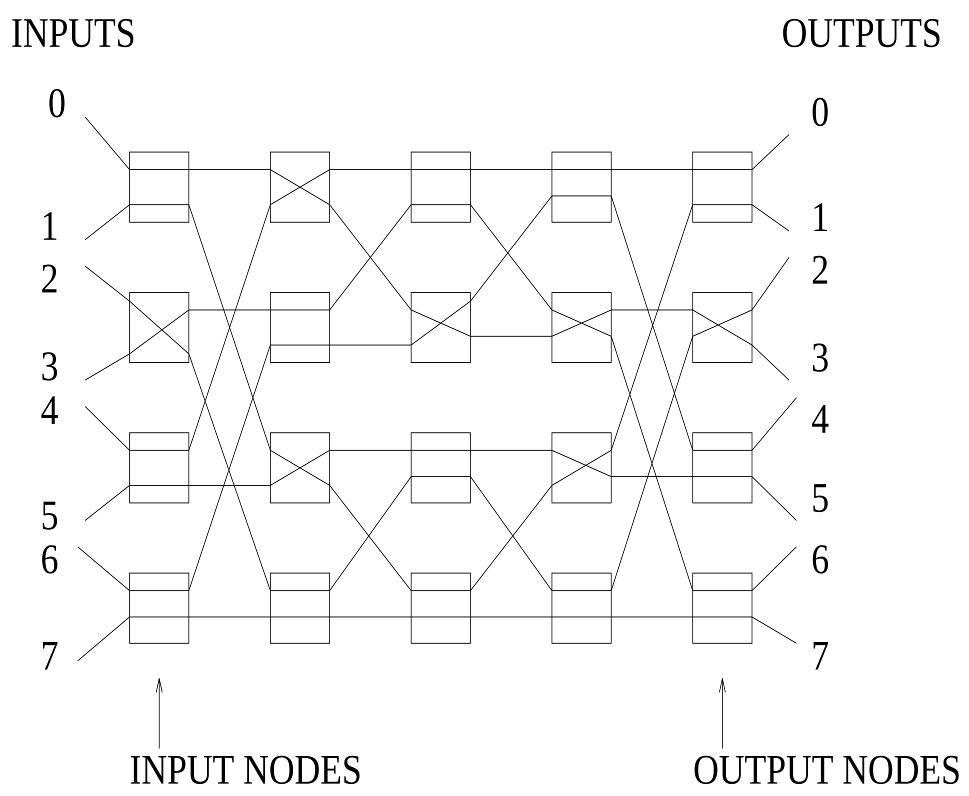

Figure 4: An 8-input (2-dimensional) Beneš network.

of this section is that an N-node butterfly or mesh of trees network can tolerate $\log^{O(1)} N$ worst-case faults and still emulate itself with only constant slowdown.

3.2.1 The Binary Tree

Define $S(n,b)$ for $n \geq b \geq 0$ by the recurrence

$$S(n,b) = S(n-1,b) + S(n-1,b-1) + 1$$

for $n > b > 0$ with boundary conditions $S(n,0) = 0$ and $S(n,n) = 2^n - 1$, for $n \geq 1$. The following lemma provides a useful asymptotic bound on the growth of $S(n,b)$ for large n.

Lemma 3.2.1 *For all $n \geq b \geq 1$, $\binom{n}{b} \leq S(n,b) \leq \binom{n+b}{b}$. Hence, $S(n,b) = \Theta(n^b)$ for any constant $b > 0$.*

Proof: The proof is by induction on n and follows trivially from the fact that $\binom{x}{y} = \binom{x-1}{y} + \binom{x-1}{y-1}$, for all $x > y > 0$. $\square$

Theorem 3.2.2 *Given an N-leaf complete binary tree T with a set of at most $S(\log N, b) = \Theta(\log^b N)$ worst-case faults at the leaves, it is possible to embed a fault-free N-leaf complete binary tree T' in T so that*

1. *nodes on level i of T' are mapped to non-faulty nodes on level i of T, for $0 \le i \le \log N$,*
2. *the congestion and the load of the embedding are at most 2^b, and*
3. *the dilation of the embedding is 1.*

Proof: The proof is by induction on $n = \log N$ and b. For $n = 0$, the tree is a single node, and the number of faults, $S(0,0)$, is zero. Now suppose that we are given a complete binary tree T with 2^n leaves of which a set of at most $S(n,b)$ are faulty. If both 2^{n-1}-leaf subtrees of T have at most $S(n-1,b)$ faulty leaves, then we use the result inductively in both subtrees. If one 2^{n-1}-leaf subtree (say the left subtree) has $S(n-1,b)+1$ or more faults, then by definition of $S(n,b)$ the other subtree (the right subtree) has at most $S(n-1,b-1)$ faulty leaves. Hence we can use induction to embed a 2^{n-1}-leaf complete binary tree on the right subtree using dilation 1 and load and congestion 2^{b-1}. By doubling the congestion and the load, we can embed two 2^{n-1}-leaf complete binary trees in the right subtree. This means that we can embed T' in T with dilation 1 and load and congestion 2^b (using only the root and the right subtree of T in this case). □

This result can be extended to a class of binary-tree-like structures called *fat-trees*. A fat-tree of depth n is specified by a sequence of numbers $m_0, m_1, \cdots, m_n$, where $m_n = 1$. (Typically $m_0 \ge m_1 \ge \cdots \ge m_n$.) A fat-tree of depth 0 is a single node, which is both the root node and the leaf node of the tree. A fat-tree of depth n is constructed as follows. At the root of the fat-tree there is a set of m_0 nodes. The left and right subtrees are identical and are constructed recursively. Each is a fat-tree of depth $n-1$ with number sequence $m_1, \cdots, m_n$. The root nodes of the fat-tree are connected to the root nodes of the left subtree with any number of edges in an arbitrary fashion (multiple edges are allowed). An isomorphic set of edges is used to connect the root nodes of the tree to the root nodes of the right subtree.

Corollary 3.2.3 *A $\log N$-depth fat-tree can be embeded in a level-preserving fashion in an isomorphic fat-tree with $S(\log N, b)$ worst-case faults at its leaves with load and congestion 2^b and dilation 1.*

Proof: Associate the nodes of the fat-tree with nodes of an N-node complete binary tree as follows. Associate all root nodes of the fat-tree with the root of the complete binary tree. Recursively associate the nodes of the left (right) subtree of the fat-tree with the nodes of the left (right) subtree of the complete binary tree. Note that every leaf of the fat-tree is associated with a unique leaf of the complete binary tree. Given a fat-tree F with faulty leaves, let T be a complete binary tree whose leaf is faulty iff the corresponding leaf in F is faulty. A fault-free complete binary tree T' can be embedded in T by embedding subtrees of T' into subtrees of T using the procedure given in Theorem 3.2.2. The same embedding can be used to embed a fault-free fat-tree, F' in F by embedding the corresponding subtrees of F' into the corresponding subtrees of F. The dilation and load are the same as that of the complete binary tree embedding. □

Corollary 3.2.4 *A* $\log N$*-dimensional butterfly can be embedded in a level-preserving fashion in an isomorphic butterfly with* $S(\log N, b)$ *worst-case faults at level* $\log N$ *with load and congestion* 2^b *and dilation 1.*

Proof: A $\log N$-dimensional butterfly is a fat-tree of depth $\log N$ with $m_i = 2^{\log N - i}$. The leaves of the fat-tree are the nodes in level $\log N$ of the butterfly. □

3.2.2 The mesh of trees and the butterfly

We can use the previous result to show that the mesh of trees and the butterfly network can tolerate $\log^{O(1)} N$ worst-case faults with constant slowdown, even when a fault can occur at *any node* of the network. The proof uses, together with Theorem 3.2.2, the fact that both the mesh of trees and the butterfly can be viewed as a special kind of product graph. We will call the leaves of a tree and the nodes in level $\log N$ of a butterfly *external* nodes. Given a graph G with $2^n = N$ external nodes, the *external product graph* of G (denoted PG) is defined as follows. Make $2N$ copies of the graph G, $G_{i,j}$, for $i = 1, 2$ and $1 \leq j \leq N$. Number the external nodes of each copy from 1 to N. Now identify the k^{th} external node in $G_{1,j}$ with the j^{th} external node in $G_{2,k}$, for all $1 \leq j, k \leq N$. (By "identify" we mean make them the same node.) The resulting graph is the external product graph of G. Two important networks can be expressed as external product graphs. When the graph G' is a tree the graph PG' is a mesh of trees network. If G' is a butterfly, then PG' is a butterfly with twice the dimension. We now show that if we can tolerate faults in the external nodes of G, then we can tolerate faults anywhere in PG.

Theorem 3.2.5 *If a graph* G' *can be embedded in a level-preserving fashion with load* l*, congestion* c*, and dilation* d *in an isomorphic graph* G *with* f *worst-case faults located in its external nodes, then it is possible to embed the product graph* PG' *in a level-preserving fashion with load* l^2*, congestion* c^2*, and dilation* d *in an isomorphic graph* PG *with* $f/2$ *worst-case faults located in any of its nodes.*

Proof: Let PG and PG' be made of up of graphs isomorphic to G, $G_{i,j}$ and $G'_{i,j}$, respectively, for $i = 1, 2$ and $1 \leq j \leq N$. Let CG and CG' also be graphs isomorphic to G. The j^{th} external node of CG is declared to be faulty iff either $G_{1,j}$ or $G_{2,j}$ contains a fault. If PG has $f/2$ faults, then CG has at most f faults (since each external node appears in two graphs $G_{1,j}$ and $G_{2,k}$). Let Φ be a level-preserving embedding of the fault-free graph CG' into CG with load l, congestion c, and dilation d and define ϕ so that Φ maps the j^{th} external node of CG' to the $\phi(j)^{th}$ external node of CG. We embed PG' into PG by mapping $G'_{i,j}$ to $G_{i,\phi(j)}$ using Φ, for $i = 1, 2$ and $1 \leq j \leq N$. It follows from the definition of faults in CG that $G_{i,\phi(j)}$ is fault-free. Therefore no nodes of PG' are mapped to faulty nodes of PG. We need to verify that every external node of PG' is mapped to a unique node of PG. The k^{th} external node of $G'_{1,j}$ is the same as the j^{th} external node of $G'_{2,k}$. The former is mapped to the $\phi(k)^{th}$ external node of $G_{1,\phi(j)}$ and the latter to the $\phi(j)^{th}$ external node of $G_{2,\phi(k)}$. These nodes are the same node of PG. The dilation of the mapping is d. The number of copies $G'_{i,j}$ of PG' mapped to any particular $G_{i,\phi(j)}$ is at most l. Each copy can put l nodes onto any particular node of $G_{i,\phi(j)}$. Therefore the load is at most l^2 and the congestion is at most c^2. □

For simplicity, we state the result for a 2-dimensional mesh of trees. However the same techniques can be used to show that any constant-dimensional mesh of trees can tolerate $\log^{O(1)} N$ worst-case faults with only a constant slowdown.

Theorem 3.2.6 *A $2\log N$-dimensional butterfly can be embedded in a butterfly of dimension $2\log N$ containing $S(\log N, b) = \Theta(\log^b N)$ worst-case faults in a level-preserving fashion with load and congestion 2^{2b} and dilation 1.*

Proof: The proof follows from Corollary 3.2.4 and Theorem 3.2.5. □

Theorem 3.2.7 *An $N \times N$ mesh of trees can be embedded in a level-preserving fashion in an $N \times N$ mesh of trees containing $S(\log N, b) = \Theta(\log^b N)$ worst-case faults with load and congestion 2^{2b} and dilation 1.*

Proof: The proof follows from Theorems 3.2.2 and 3.2.5. □

The results of this subsection can also be formulated using the fact that the butterfly and the mesh of trees can be expressed as the Layered Cross Product [EL92] of two complete binary trees (or variations thereof) [Aie92].

3.2.3 Limitations on level-preserving embeddings

We do not know whether or not Theorems 3.2.2, 3.2.6, and 3.2.7 can be improved if the level-preserving constraint is removed. However, we can show that the bounds in Theorems 3.2.2, 3.2.6, and 3.2.7 are tight if the embedding is forced to be level-preserving.

Given an N-leaf binary tree T with faults at its leaves, an arrow diagram has arrows drawn from some nodes of T to their siblings. We define a b-legal arrow diagram as follows:

1. On any path from the root to a faulty leaf, there is an arrow from a node on the path to a node not on the path (outgoing arrow).
2. On any path with no outgoing arrow, there can be at most b incoming arrows.

Let $T(n, b)$ be the maximum number of faults that can be placed at the leaves of a 2^n-leaf binary tree (where $2^n = N$) without making it impossible to construct a b-legal arrow diagram for the tree. We bound the value of $T(n, b)$ as follows.

Lemma 3.2.8 *For all $n \geq b \geq 1$, $T(n, b) \leq T(n-1, b) + T(n-1, b-1) + 1 = O(n^b)$.*

Proof: We will first show that $T(n, 0) = 0$. If there are no faults in the tree, then a tree with no arrows is a 0-legal arrow diagram. However, even if the tree has only one fault it necessarily has at least one arrow. If the tree has at least one arrow, then there must be path from the root of the tree to a leaf having at least one incoming arrow and no outgoing arrow. Such a path can be recursively constructed as follows. Choose the arrow from node

m' to its sibling m such that m is the node with an incoming arrow closest to the root of the tree. The constructed path is the path from the root of the tree to m concatenated with the path constructed recursively in the subtree rooted at m. (If there is no arrow in the subtree rooted at m a path from m to any leaf of the subtree will be sufficient.) Thus, a tree with a fault cannot have a 0-legal arrow diagram. Hence $T(n,0) = 0$. $T(n,n) = 2^n - 1$, since if we allow n incoming arrows without an outgoing arrow we can place an incoming arrow on every node of the path from the non-faulty node to the root. For $n > b > 0$, we show that

$$T(n,b) \leq T(n-1,b) + T(n-1,b-1) + 1.$$

We show this by proving that there is a way of placing $T(n-1,b)+T(n-1,b-1)+2$ faults at the leaves such that either there must be at least $b+1$ incoming arrows on some path without any outgoing arrows or there must be a faulty leaf with no outgoing arrow in its path. We place $T(n-1,b)+1$ worst-case faults in the left subtree and $T(n-1,b-1)+1$ worst-case faults in the right subtree. Assume that it is possible to place arrows in the tree such that every path to a faulty leaf has an outgoing arrow and every path from the root to a leaf which has no outgoing arrows has at most b incoming arrows. We look at the placement of arrows in the left subtree. Since there are more than $T(n-1,b)$ faults, there must be a path from the root of this subtree to a leaf which has $b+1$ incoming arrows and no outgoing arrows or there must be path from the root of this subtree to a fault with no outgoing arrow. Either of these cases implies that the root of the left subtree must have an arrow from itself to its sibling. Now look at the right subtree. It cannot be the case that there is a path from the root of the right subtree to a faulty leaf with no outgoing arrow, since then there will be no outgoing arrow for the path from the root of T to this fault. Further, no path from the root of the right subtree to a leaf of the right subtree can have more than $b-1$ incoming arrows without having an outgoing arrow, since otherwise there will be a path from the root of the tree to that leaf with more than b incoming arrows without an outgoing arrow. Thus the right subtree must be $(b-1)$-legal. However, the right subtree has more than $T(n-1,b-1)$ worst-case faults. This is a contradiction. The recurrence we have for $T(n,b)$ is similar to what we had for $S(n,b)$ in Section 3.2.1 and is $O(S(n,b))$. Using Lemma 3.2.1, $T(n,b)$ is $O(n^b)$. □

Theorem 3.2.9 *For any constants, c, l, and d, there is a constant k such that there is a way of placing $\log^k N$ faults in the leaves of an N-leaf complete binary tree T such that there is no level-preserving embedding of an N-leaf fault-free complete binary tree T' in T with congestion c, load l, and dilation d.*

Proof: We are given an $N = 2^n$ node complete binary tree T with faults at its leaves and its fault-free version T'. Let $k = \lceil d + (l-1)2^{d-1} \rceil$. We will choose a worst-case set of faults in T of cardinality $\Theta(\log^k N)$ such that this fault pattern has no k-legal arrow diagram. Clearly this is possible, since $T(n,k)+1$ is $O(\log^k N)$. Suppose, for contradiction, that there is an embedding of T' to T with the property that the nodes in level i of T' are mapped to nodes in level i of T, and no nodes of T' are mapped to faulty nodes of T, with dilation (d), congestion (c) and load (l). Annotate the tree T with arrows as follows. For any two siblings in the tree, draw an arrow from the sibling whose subtree has a smaller number of leaves of T' mapped to it to the sibling that has a greater number of leaves mapped to it. If the number of leaves mapped to each of the two subtrees is equal then no arrow is drawn.

We will now show that the annotated tree is b-legal, for some b less than or equal to k. The path from the root of a tree to any faulty leaf must have an outgoing arrow, since no node of T' is mapped to a faulty leaf. Let b be the maximum number of incoming arrows on a path without an outgoing arrow. We will ignore the last d levels of the tree. Therefore, there is a path in T, $p_0, p_1, \cdots, p_{n-d}$, where p_0 is the root and p_i is some node in level i of the tree, which has at least $b - d$ incoming arrows without any outgoing arrows. Let l_i, $0 \leq i \leq n-d$, denote the average number of leaves of T' that are embedded into each leaf of the subtree of T rooted at node p_i. Clearly, l_0 is 1. If there is no incoming arrow into node p_k, then the split of leaves of T' is even and hence $l_k = l_{k-1}$. Suppose there is an incoming arrow into node p_k from its sibling ${p_k}'$, then l_k is greater than l_{k-1}. Further, l_k is at least $l_{k-1} + 2^{-d+1}$. To see why, consider the subtrees of T' rooted at level $k + d - 1$. The nodes in each of these subtrees can be mapped entirely within either the subtree rooted at p_k or entirely within the subtree rooted at ${p_k}'$ but never to the nodes in both. The reason is that it is not possible for a node of T' to be mapped to one of the sub-trees (say p_k) to reach a node mapped at an adjacent level in the other subtree (say ${p_k}'$) with dilation d or less. Thus subtree p_k must have at least $2^{n-k-d+1}$ more leaves of T' mapped to it than subtree ${p_k}'$. Thus, since there are at least $b-d$ arrows, l_{n-d} must be at least $1 + (b-d)2^{-d+1}$. Note that there is at least one leaf in the subtree rooted at p_{n-d} which has load at least l_{n-d}. Therefore, l_{n-d} is at most l. This implies that b is at most $d + (l-1)2^{d-1}$, i.e., at most k. But there can be no k-legal arrow placement for the fault pattern chosen for T. This is a contradiction. □

Theorem 3.2.10 *For any constants, c, l and d, there is a constant k such that there is a way of choosing $\Theta(\log^k N)$ faults in an N-input butterfly B such that there is no level-preserving embedding of an N-input butterfly B′ in B with congestion c, load l, and dilation d.*

Proof: The proof is similar to that of Theorem 3.2.9. Let B be a butterfly with faults and B' be the fault-free version of B. We can associate a tree T with B as follows: The root of T represents the entire butterfly B. Its children represent the two sub-butterflies of dimension $\log N - 1$ (between levels 1 and $\log N$). Each child is subdivided recursively until each leaf of the tree T represents a unique node in level $\log N$ of the butterfly B. We will choose the same set of worst-case faults in the leaves of T as in Theorem 3.2.9. The faulty nodes of B will be the nodes in level $\log N$ of B that correspond to the faulty leaves of T. (Note that faults are not needed on any other level of the tree.) Given a level-preserving embedding of B' into B with load l, congestion c and dilation d, we can produce a b-legal placement of arrows in T in a manner similar to the previous proof. Given two siblings m and m', draw an arrow from m' to m if the there are more nodes in level $\log N$ of B' mapped to the sub-butterfly of B represented by tree node m than the sub-butterfly represented by tree node m'. Let m and m' be on level j of T. As before, due to dilation considerations, it is true that smaller sub-butterflies of B' spanning levels $j+d-1$ to n must be mapped entirely within the sub-butterfly of B represented by m or within the sub-butterfly represented by m' but never to both. The rest of the proof is similar to Theorem 3.2.9. □

Theorem 3.2.11 *For any constants, c, l and d, there is a constant k such that there is a way of choosing $\Theta(\log^k N)$ faults in an N by N mesh of trees M such that there is no level-preserving embedding of an N by N mesh of trees M' in M with congestion c, load l, and dilation d.*

Proof: The proof is similar to that of Theorem 3.2.10. Let M be a mesh of trees with faults and M' be the fault-free version of M. The nodes in level $\log N$ of M and M' are arranged in the form of a 2-dimensional N by N mesh. We will refer to these nodes as the mesh nodes. We can associate a tree T with the mesh nodes of M as follows: The root of T represents the entire mesh. Divide the mesh vertically into two equal parts and let each child represent one of the halves. At the next level of the tree divide each of the halves horizontally into two equal parts. Divide alternately, either horizontally or vertically, until you reach individual mesh nodes which are each represented by a unique leaf of the tree. We will choose the same set of worst-case faults in the leaves of T as in Theorem 3.2.9. The faulty nodes of M will be the mesh nodes of M which correspond to the faulty leaves of T. Given a level-preserving embedding of M' into M with load l, congestion c and dilation d, we can produce a b-legal placement of arrows in T in a manner similar to the previous proofs. Given two siblings m and m', draw an arrow from m' to m if the there are more mesh nodes of M' mapped to the sub-mesh of M represented by tree node m than the sub-mesh represented by tree node m'. As before, due to dilation considerations, it is true that smaller sub-meshes of M' must be mapped entirely within the sub-mesh of B represented by m or within the sub-mesh represented by m' but never to both. The rest of the proof is similar to Theorem 3.2.9. □

3.3 Fault-tolerant routing

In this section, we present algorithms for routing around faults in hypercubic networks. Section 3.3.1 presents algorithms for routing packets in butterfly networks with faulty nodes, while Section 3.3.2 presents algorithms for establishing edge-disjoint paths between the inputs and outputs of a Beneš network with faulty switches.

3.3.1 Fault-tolerant packet routing

In this section, we show how to route packets in an N-input butterfly network with f worst-case faults. The routing problem studied in this section is slightly different from that studied in Chapter 2. In this section, every node of the butterfly can send and receive packets whereas in Chapter 2 only the nodes in level 0 (input nodes) can send packets and only the nodes in level $\log N$ (output nodes) can receive packets. A widely studied class of routing problems is the class of *permutation routing problems*. A permutation routing problem between a set of nodes in the network has exactly one packet originating at every node in this set. Each such packet needs to be routed to a destination node in the set, and no two packets share the same destination.

The permutation routing problem on the nodes of a butterfly as defined in this section is equivalent to the routing problem in which every input node of the butterfly sends $\log N$ packets and every output node receives $\log N$ packets. The latter problem was referred to as routing a fully-loaded butterfly in Chapter 2. The reason why they are equivalent is as follows. Each packet originating in some node of the butterfly could first route from its node of origin to the level 0 node in the same row. Thus every level 0 node receives $\log N$ packets from the nodes in its row. Each packet then routes to the level $\log N$ node in the row of its destination. Each level $\log N$ node receives $\log N$ packets destined for the nodes in its row. Finally, each packet routes along the row of its destination from the level $\log N$ node to its actual destination. The time taken to route along a row in the first and the last stages of the routing is $O(\log N)$. Routing each packet from the level 0 node in the row of its origin to the level $\log N$ node in the row of its destination is equivalent to routing a fully-loaded butterfly.

In this section, we show that there is some set of $N - O(f)$ rows such that it is possible to route any permutation between the nodes in these rows in $O(\log N)$ steps, with high probability. This is result is comparable to the result for fault-tolerant routing in a multi-butterfly[LM92]. A special case of this result is that when f equals αN (for any $\alpha < 2/9$) we can route arbitrary permutations between a majority of nodes in the butterfly. Note that this is optimal within constant factors since N faults in level $\log N/2$ can bisect the butterfly into many small components.

We start by describing Valiant's algorithm [Val82] for permutation routing in a butterfly without faults. It will be convenient for us to view the packets in this scheme as being routed on a bigger network with $4\log N + 1$ levels called the *virtual network*. Between level 0 and level $\log N$, and between levels $3\log N$ and $4\log N$, nodes are connected only by straight edges. Between levels $\log N$ and $3\log N$ the network is a pair of back-to-back butterflies isomorphic to the Beneš network. In analogy with the Beneš network the nodes in levels $\log N$, $2\log N$ and $3\log N$ are called input nodes, middle nodes and output nodes respectively. The virtual network can simulated by a single butterfly by embedding the virtual network onto the butterfly with load and congestion equal to 4 and dilation equal to 1 as follows. We fold the virtual network in an accordion-like fashion at levels $\log N$, $2\log N$, and, $3\log N$. The folded network can be embedded in a one-to-one fashion onto to the butterfly since one is isomorphic to the other. Note that each node of the folded network corresponds to four nodes of the virtual network. Thus each node of the butterfly has four nodes of the virtual network embedded onto it. Note that the nodes in levels 0, $2\log N$ and $4\log N$ of the virtual network are embedded onto nodes in level 0 of the butterfly. The nodes in $\log N$, and $3\log N$ of the virtual network are embedded onto nodes in level $\log N$ of the butterfly. Each node of the butterfly simulates all the nodes of the virtual network that are mapped to it. It is easy to see that the butterfly can simulate one step of the virtual network in at most four time steps.

Valiant's algorithm for routing in a fault-free virtual network is as follows. In Stage 0, a packet goes down its row to the input node in that row (say m). In Stage 1, the packet goes from m to a random middle node (say m''). In Stage 2, the packet goes from m'' to an output node m' in the row of its destination. In Stage 3, the packet goes from the row of m' to its destination. Valiant showed that these paths, which have length at most $4 \log N$, also have congestion $O(\log N)$ with high probability. In leveled networks such as the butterfly, as long as the (leveled) paths of the packets are selected such that the maximum length is $O(\log N)$ and congestion $O(\log N)$, a Ranade-type queuing protocol [Ran87c] can be used to route the packets in $O(\log N)$ steps [LMR88]. Therefore it will be sufficient to derive high probability bounds on the maximum length and congestion of the paths of a routing scheme.

In a faulty butterfly, we would like to use an algorithm like Valiant's to route packets between the "good" nodes. Let F be the set of worst-case faults on the butterfly and let $f = |F|$. As before, we will view the packets as being routed in the virtual network. Any node of the virtual network embedded to a faulty node of the butterfly is considered faulty. Since Stages 0 and 3 require a fault-free row, any node in a row with a fault is declared to be *bad*. Furthermore, in Stage 1, every packet needs a sufficient number of random choices of middle nodes. For any input (or output) node m, let $REACH(m)$ be defined to be the set of middle nodes reachable from m using fault-free paths of length $\log N$. If $|REACH(m)| < 4N/5$ for any input (or output) node m, then we define m and all other nodes in its row to be bad. Any node not defined as bad is defined to be *good*. Note that there are no faults in rows containing good nodes, and every good input (output) node can reach at least $4N/5$ middle nodes via fault-free paths.

We will now show that only $O(f)$ rows contain bad nodes. This follows from the fact that only f rows can contain faults and from the fact that $|REACH(m)| \geq 4N/5$ for all but $O(f)$ input nodes m. The latter fact is proved by setting $t = N/5$ in the following lemma, which will also be used in Section 3.3.2.

Lemma 3.3.1 *In an N-input butterfly with f worst-case faults, at least $N - fN/t$ nodes in level 0 can each reach at least $N - t$ nodes in level $\log N$ via fault-free paths of length $\log N$, for any $t \leq N$.*

Proof: For each node i in level 0, let n_i represent the number of nodes in level $\log N$ that i cannot reach. If the lemma were false, then we would have

$$\sum_{i=1}^{N} n_i \geq \left(\frac{fN}{t} + 1\right)(t+1).$$

A fault at any level of the butterfly lies on precisely N paths from nodes in level 0 to nodes in level $\log N$. Hence $\sum_{i=1}^{N} n_i \leq fN$. Combining the inequalities yields $fN \geq (fN/t+1)(t+1)$ which is a contradiction. Hence the lemma must be true. □

In order to route any permutation between the good nodes, we use Valiant's algorithm except that in Stage 1, we randomly select a middle node m'' from $REACH(m)$

$\cap REACH(m')$. (This step requires us to store at each node m a table containing information about $REACH(m) \cap REACH(m')$ for each node m'.) Since m and m' are good input and output nodes $|REACH(m) \cap REACH(m')|$ is at least $3N/5$. We will now prove that the paths selected in this manner have congestion $O(\log N)$ with high probability. This also implies that the routing will complete in $O(\log N)$ steps with high probability [LMR88].

Theorem 3.3.2 *The paths selected have length* $4 \log N$ *and the congestion of the paths is* $O(\log N)$ *with probability at least* $1 - 1/N^k$ *(for any constant k).*

Proof: The lengths of the paths are clearly $4 \log N$ since the paths traverse the butterfly four times, once in each stage. We bound the congestion as follows. Every good node sends and receives one packet. The congestion of any node in Stages 0 and 3 is trivially at most $\log N$. Consider a node s in level l of the butterfly in Stage 1. There are $2^l \log N$ packets that could pass through this node. A packet passes through this node iff it selects as a random middle node, one of the $2^{\log N - l}$ middle nodes reachable from this node. Note that the set of possible choices of middle nodes for any input node m and output node m' is $REACH(m) \cap REACH(m')$. Since both m and m' are good nodes, the cardinality of both $REACH(m)$ and $REACH(m')$ is at least $4N/5$. This implies that the cardinality of $REACH(m) \cap REACH(m')$ is at least $3N/5$. Thus the probability that the chosen middle node is reachable from s is at most $2^{\log N - l}/(3N/5)$. Therefore the average number of packets passing through a node in Stage 1 is at most $2^l \log N 2^{\log N - l} 5/(3N)$ which is $5 \log N/3$. We can use Chernoff bounds [Rag90] to show that the number of packets through s in Stage 1 is $O(\log N)$ with probability at least $1 - o(1/N^k)$. The calculation for a node in Stage 2 is analogous. Thus the congestion is $O(\log N)$ with probability at least $1 - 1/N^k$. □

Packet routing without routing tables

In the previous algorithm, each good input node m was required to store a table containing information about $REACH(m) \cap REACH(m')$ for every other node m'. In this section, we will show that is possible to route packets in a faulty butterfly without using such routing tables. The information about the placement of the faults will be used only during the reconfiguration when the good and bad nodes are identified. This information will not be needed for the routing itself. We will assume that any packet that attempts to go through a fault is simply lost. We will further assume that a node that receives a packet sends back an acknowledgement message (ACK) to the sender. The ACK messages will follow the path of the packet in reverse. The algorithm for routing proceeds in rounds. There will be a total of $A \log\log N$ rounds (for some constant A). Each round consists of the following steps (R takes values from 0 to $A \log\log N - 1$ and denotes the round number).

SEND-PACKET: In Stage 0, if packet p has not yet been delivered to its destination, send 2^R identical copies of p to the input node m in its row. In Stage 1, send each copy of the packet independently to a random middle node. In Stage 2, send each copy to the appropriate output node m'. In Stage 3, send each copy to the appropriate destination node in that row.

RECEIVE-PACKET: If a packet is received send an ACK along the same path that the packet came through in reverse.

WAIT: Wait $B \log N$ steps before starting the next round.

Theorem 3.3.3 *For appropriate choices of the constants A and B, the above algorithm routes any permutation on the $(N - O(f)) \log N$ good nodes of the butterfly in $O(\log N \log\log N)$ steps with probability at least $1 - \frac{1}{N^k}$, for any fixed constant $k > 0$.*

Proof: First we show that there is very little probability that a packet survives $A \log\log N$ rounds without reaching its destination, where A is an appropriately chosen constant. A packet can never encounter a fault in Stages 0 and 3, since its source and destination rows are fault-free. Let the input node that this packet passes through be m and the output node m'. In Stage 1, if the packet chooses any middle node in $REACH(m) \cap REACH(m')$ it will succeed in reaching its destination. Since the cardinality of $REACH(m) \cap REACH(m')$ is at least $3N/5$ the probability of this happening is at least $3/5$. Suppose the packet did not get through after $A \log\log N - 1$ rounds. Then $2^{A \log\log N} = \log^A N$ copies of the packet will be transmitted in the last round. Note that the probability of each copy surviving is independent of the others. Hence the probability that none of these copies reach their destination is at most $(1 - 3/5)^{\log^A N}$ which is at most $1/N^{k+2}$ for an appropriate choice for the constant A. Thus, the probability that some packet does not reach its destination is at most $N \log N / N^{k+2}$ which is $o(1/N^k)$.

Next we show that each round takes $O(\log N)$ time with high probability. We assume inductively that at the beginning of round i the total number of packets (counting each copy once) to be transmitted from any row in Stage 0 of the algorithm or received by any row in Stage 3 of the algorithm is at most $q \log N$ for some constant $q > 1$. Clearly the basis of the induction is true since at the beginning of the first round there are exactly $\log N$ packets sent by each row in Stage 0 and received by each row in Stage 3. The average number of copies that were sent from a row in Stage 0 or that were destined for a row in Stage 4 that did not get through is at most $2q \log N/5$. The value of q is chosen such that the probability that more than $q \log N/2$ copies do not get through in any row can be shown to be small, i.e., $o(1/N^{k+1})$, using Chernoff bounds. At the beginning of the next stage, each unsent copy will be duplicated and hence with high probability, the number of packets in any row in the next round will be at most $q \log N$. Since there are $\log\log N$ rounds the probability that the inductive hypothesis will not hold in the beginning of any one round is at most $A \cdot \log\log N \cdot o(1/N^{k+1})$ which is $o(1/N^k)$.

Now we assume that the inductive hypothesis is true and show that each round takes only $O(\log N)$ steps with high probability. Consider any round i. From the inductive hypothesis, the congestion of any node in Stage 0 or 3 is at most $q \log N$ which is $O(\log N)$. In Stage 1, a node at level l can receive packets from any one of the 2^l input nodes, each packet with a probability of 2^{-l}. The total number of packets which pass through an input node is at most $q \log N$ by our inductive hypothesis. Therefore the average number of packets passing through the node is $q \log N 2^l 2^{-l}$ which is $q \log N$. The value of q is chosen so that the probability that any node gets more than $2q \log N$ packets can be shown to be $o(1/N^{k+1})$, using Chernoff bounds. The analysis for Stage 2 is similar. Thus we have shown that if the inductive hypothesis is true, the congestion of any node is $O(\log N)$ with high probability. Therefore, using Ranade's scheme to schedule the packets, the routing will complete in $C \log N$ steps with probability at least $o(1/N^{k+1})$, for an appropriate constant C. The ACKs follow the path of packet in the reverse direction. Therefore the congestion in any node due to ACKs can be no more than the congestion due to packets and is also therefore $O(\log N)$. Since we are using Ranade's algorithm to schedule the packets, the probability that the ACKs will not reach their destinations in $D \log N$ time is at most $o(1/N^{k+1})$, for some suitably large constant D. We choose the constant B in the algorithm to be at least $C + D$ so that the algorithm waits long enough for both the packet routing and the routing of ACKs to finish before starting the next round of routing. The probability that either the packet routing or the ACK routing fails to complete in some round is at most $2A \log\log N o(1/N^{k+1})$ which is $o(1/N^k)$.

The probability that either some packet remains untransmitted after the last round or that the inductive hypothesis does not hold for some round or that some round fails to complete in $B \log N$ steps is simply $3o(1/N^k)$ which is $O(1/N^k)$. Thus the algorithm successfully routes every packet to its destination in $O(\log N \log\log N)$ steps with probability at least $1 - O(1/N^k)$. □

If the number of worst-case faults is smaller there is a simpler way of routing without any routing tables.

Theorem 3.3.4 *Given a butterfly with $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$), it is possible to identify $N - o(N)$ good nodes in the butterfly such that any permutation routing problem on the good nodes can be routed in $O(\log N)$ steps with probability greater than $1 - 1/N^k$, for any fixed constant k, without any routing tables.*

Proof: We choose the good nodes in the butterfly much in the same way as before except that our threshold is much smaller. Any good node must have fault-free row and the input node in the row of a good node must reach at least $N - t$ middle nodes using fault-free paths of length $\log N$, for $t = N^{1-\epsilon/2}$. Using Lemma 3.3.1, we can show that the number of good nodes is at least $N - \Theta(N^{1-\epsilon/2})$. The algorithm is the same as the previous algorithm except that we now need only a constant number of routing rounds with high probability. This is because each unsent packet at each round has only $\Theta(1/N^{\epsilon/2})$ probability of hitting a fault. Therefore it is sufficient to have $\Theta(k/\epsilon)$, i.e., some constant, number of rounds before every packet is delivered with probability at least $1 - 1/N^k$. □

3.3.2 Fault-tolerant circuit-switching

In this section, we examine the ability of the Beneš network to establish disjoint paths between its inputs and outputs when some of its switches fail. We assume that no path can pass through a faulty switch. The primary result of this section is that for arbitrarily small positive constants ϵ and δ, there is a constant b such that given a b-dilated $\log N$-dimensional Beneš network with $f = N^{1-\epsilon}$ worst-case switch failures, we can identify a set of $N - 2N^{1-\delta}$ input and output nodes such that it is possible to route edge-disjoint paths in any permutation between the corresponding input and output edges. (A *b-dilated* Beneš network is one in which each edge is replaced by b parallel edges, and each 2×2 switch is replaced by a $2b \times 2b$ switch.) As an example, a 2-dilated Beneš network can tolerate up to $\sqrt{N}/4$ worst-case switch failures and still route arbitrary permutations between 75 percent of its inputs and outputs.

In a $\log N$-dimensional Beneš network, levels 1 through $2\log N - 1$ can be decomposed into two disjoint sub-Beneš networks of dimension $\log N - 1$, a top sub-Beneš network and a bottom sub-Beneš network. Note that the two paths that originate from input edges that share an input node cannot use the same sub-Beneš network. The same is true for paths that end on output edges that share the same output node. For a full permutation, there are $2N$ input-output pairs which require paths to be routed between them. The standard algorithm presented below for setting the switches in a Beneš network, due to Waksman [Wak68], uses bipartite graph matching to split the set of $2N$ pairs into two sets of N pairs which are each then routed recursively in one of the smaller sub-Beneš networks.

We present Waksman's algorithm with a twist. We will call this algorithm RANDSET (for RANDom switch SETting). The way RANDSET differs from Waksman's algorithm is that it randomly chooses which of the two sets of N pairs to route through the top (and bottom) sub-Beneš network. The input to RANDSET is a permutation ϕ represented as a $2N \times 2N$ bipartite graph. The nodes of the graph represent the $2N$ input edges and the $2N$ output edges of the network. An edge in the bipartite graph from input i to output $\phi(i)$ indicates that a path must be routed from i to $\phi(i)$ in the network. The first step is to merge pairs of nodes in the bipartite graph that correspond to input edges (or output edges) that share the same input (or output). The result is a 2-regular $N \times N$ bipartite graph. The second step is to split the edges of this graph into two perfect matchings, M_0 and M_1. Next, we pick a binary value for random variable X at random. If $X = 0$ then we recursively route the paths in matching M_0 through the top sub-Beneš network and those in M_1 through the bottom sub-Beneš network. If $X = 1$ we do the reverse. The following lemma shows that RANDSET chooses the path from i to $\phi(i)$ uniformly from among all possible paths.

Lemma 3.3.5 *For any i, the path chosen by algorithm RANDSET between input i and output $\phi(i)$ in a $2N$-input Beneš network passes through any of the N middle nodes (nodes in level $\log N$) with equal probability ($1/N$).*

Proof: At the first stage, the path from i to $\phi(i)$ goes to the top or the bottom sub-Beneš network with probability $1/2$ depending on whether the matching that contains the edge corresponding to this input-output pair is chosen to be routed through the top or the bottom. The decisions made at the succeeding levels of the recursion are similar and independent of all other decisions. $\square$

It is important to remember that given a permutation, the *paths themselves are highly correlated* and determining one path gives some information about the others.

We will classify the input and output nodes of the Beneš network as either good or bad depending on whether they can reach a sufficiently large number of middle nodes. Let F

be the set of faulty switches in the Beneš network. In a fault-free Beneš network, there is a path from each input (and output) node to each of the N middle nodes. The middle nodes in fact form the leaves of a complete binary tree with the input (or output) node as the root. The faults could make some of these paths unusable. We will declare an input (or output) node *bad* if the number of middle nodes that it cannot reach exceeds a certain threshold. The threshold will be chosen so that it is possible to establish edge-disjoint paths between the *good* (i.e., not bad) inputs and outputs in any permutation. (The 2 inputs coming into an input (or output) node are good or bad depending on whether the corresponding input (or output) node is good or bad).

Let $BAD(t)$ be the set consisting of input nodes as well as output nodes for which more than t middle nodes are unreachable. The first and last $\log N + 1$ levels of the Beneš network each form a $\log N$-dimensional butterfly. By using Lemma 3.3.1 for each of these butterflies, we know that $|BAD(t)| \leq fN/t$.

Theorem 3.3.6 *For any constants $0 < \epsilon < \delta \leq 1$, there exists a constant $b = \lceil 1 + (2-\epsilon)/(\epsilon - \delta) \rceil$ such that a b-dilated Beneš network with $N^{1-\epsilon}$ worst-case switch faults has a set of $N - 2N^{1-\delta}$ input nodes and output nodes between whose input and output edges it is possible to route any permutation using edge-disjoint paths.*

Proof: We will declare any input or output node in $BAD(N^{1+\delta-\epsilon}/2)$ to be bad. Since we need the number of good input nodes and good output nodes to be equal we may have to declare some extra input nodes or output nodes to be bad. From Lemma 3.3.1, we know that $BAD(N^{1+\delta-\epsilon}/2)$ is at most $2N^{1-\delta}$. Thus, the number of good input nodes (or output nodes) is at least $N - 2N^{1-\delta}$.

We now prove that we can route any permutation between the good inputs and good outputs using edge-disjoint paths. In this proof, we will simply show that for every permutation, such a set of paths *exists*, without showing how to compute these paths efficiently. Later, we give an efficient procedure for computing these paths.

Given a permutation ϕ on the good inputs and outputs, we will select paths using RANDSET in b rounds. In the first round, we route all the paths using RANDSET. Some of these paths pass through faults in the network. The number of paths that pass through faults is at most $2N^{1-\epsilon}$, since each fault can kill at most 2 paths. These paths are not permissible and have to be rerouted in the second round using RANDSET. Note that every good input node (or output node) has at most $N^{1+\delta-\epsilon}/2$ unreachable middle nodes. Thus, from Lemma 3.3.5, the probability that any one of the paths hits a fault in the first $\log N + 1$ levels is at most $N^{-(\epsilon-\delta)}/2$. The probability that it hits a fault in the second $\log N + 1$ levels is also at most $N^{-(\epsilon-\delta)}/2$. The net probability that the path will hit a fault is at most $N^{-(\epsilon-\delta)}$. Even though the probabilities that any two paths hit a fault are correlated, the expected number of paths that hit faults is at most $2N^{1-\epsilon}N^{-(\epsilon-\delta)}$. This implies that there is a non-zero probability that RANDSET will find paths such that at most $2N^{1-\epsilon-(\epsilon-\delta)}$ paths hit faults. Note this also means that there *exists* a way of selecting the paths so that at most $2N^{1-\epsilon-(\epsilon-\delta)}$ paths hit faults. We select paths such that this criterion is satisfied and route the paths that hit faults again using RANDSET. We continue to do the rerouting until the expected number of paths that hit faults drops below 1. At this point with non-zero probability RANDSET routes all of the paths without hitting any faults. In particular, such a set of paths exist. The expected number of paths that hit faults in the i^{th} round is $2N^{1-\epsilon-(i-1)(\epsilon-\delta)}$. Thus, for $b = \lceil 1 + (2-\epsilon)/(\epsilon-\delta) \rceil$, the number of paths that hit faults at the end of the b^{th} round is less than 1. Therefore all paths will be routed by the end of the b^{th} round. Since we use at most b rounds of routing and since each edge of the Beneš network has been replaced by b edges we will obtain edge-disjoint paths for the permutation. □

Derandomizing RANDSET

In the proof of Theorem 3.3.6, we show the existence of edge-disjoint paths by using the fact that algorithm RANDSET will find them with non-zero probability. In this section we construct a deterministic algorithm that always finds these paths using the technique due to Raghavan [Rag88] and Spencer [Spe87] to remove the randomness. Further, like Waksman's algorithm for finding the switch settings in a fault-free Beneš network with N input nodes, the algorithm runs in $O(N \log N)$ time.

Let P be the random variable which denotes the number of paths that pass through faults at some stage of rerouting. Let X be the binary random variable used by RANDSET to make its random decision to select which matching is to be routed through which sub-Beneš network. Let us further define two random variables, P_l and P_r to denote the number of paths that RANDSET routes through faults in the left butterfly and the right butterfly respectively. Let $U(P)$ be an upperbound on $E(P)$ that is defined as $E(P_l) + E(P_r)$. In the proof of Theorem 3.3.6, we used the fact that there is a non-zero probability that RANDSET will find a set of paths with at most $U(P)$ paths hitting faults. We will define a procedure DSET (for Deterministic switch SETting) that will deterministically find such a set of paths. Algorithm DSET is the same as RANDSET except that instead of selecting a random value for X, we select the "better" choice for X as follows. We compute $U(P|(X = i)) = E(P_l|(X = i)) + E(P_r|(X = i))$, for $i = \{0, 1\}$. We then choose X to be the value of i that yields the minimum of the two values computed above.

Theorem 3.3.7 *Given a (partial) permutation ϕ to be routed, Algorithm DSET deterministically computes paths such that not more than $U(P)$ paths hit faults in each round, and has the same asymptotic running time as RANDSET.*

Proof: We will prove the theorem by induction on the size of the Beneš network. The base case is trivial. Consider an N-input Beneš network with a (partial) permutation ϕ to route. From the definition of $U(P)$ we have the following.

$$\begin{aligned} U(P) &= E(P_l) + E(P_r) \\ &= \frac{1}{2}(E(P_l|(X=0)) + E(P_l|(X=1))) \\ &\quad + \frac{1}{2}(E(P_r|(X=0)) + E(P_r|(X=1))) \\ &= \frac{1}{2}(U(P|(X=0)) + U(P|(X=1))) \qquad (7) \end{aligned}$$

Let i be the value chosen for X in Step 2 of DSET. From Equation 7 it is clear that $U(P|(X = i)) \leq U(P)$. For $X = i$, let random variables P_t and P_b be the number of paths passing through faults in the top and bottom sub-Beneš networks respectively but not in the first or the last stage of the Beneš network. Let C be the number of paths that pass through faults in the first or the last stage of the Beneš network. Algorithm DSET eliminates paths that pass through faults in the first or last levels before recursively routing through the smaller sub-Beneš networks. The number of paths eliminated in this manner is C. By the induction hypothesis, we can assume that DSET routes no more than $U(P_t)+U(P_b)$ paths through faults in the smaller sub-Beneš networks. Hence DSET routes at most $C + U(P_t) + U(P_b)$ paths through faults. This equals $U(P|(X = i))$ which is at most $U(P)$. Thus DSET routes at most $U(P)$ paths through faults in each round.

Now we deal with the question of how efficiently $U(P|(X = i))$, for $i \in \{0, 1\}$, can be calculated in step 2 of DSET. For every node m in the Beneš network, let $REACH(m)$ be the set of middle nodes reachable from node m using fault-free paths. We can precompute the cardinality of $REACH(m)$ as follows. The values for the middle nodes are trivially known. We then compute the values for levels on both sides adjacent to levels where the values are known and continue in this manner. This takes only $O(N \log N)$ steps of precomputation and does not affect the asymptotic time complexity of the algorithm. Given the values of $|REACH(m)|$, the values of $U(P|(X = i))$ can be easily calculated by summing up the appropriate values of $|REACH(m)|/N$. This is an $O(N)$ time computation. Since Step 1 of the algorithm takes N time just to set N switches in the first level, this will not affect the asymptotic time complexity . Hence using DSET yields the same asymptotic time complexity as RANDSET and takes time linear in the size of the Beneš network. $\square$

3.4 Emulations on faulty butterflies

In this section, we show that for any constant $\epsilon > 0$, a $\log N$-dimensional butterfly with $N^{1-\epsilon}$ worst-case faults (the host H), can emulate any computation of a fault-free version of itself (the guest G) with only constant slowdown. We assume that a faulty node cannot perform computations and that packets cannot route through faulty nodes. For simplicity we assume that the butterflies wrap around, i.e., the nodes of level 0 are identified with the nodes of level $\log N$.

We model the emulation of G by H 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 represents the entire state of the computation performed at node v at time t. The s-pebble contains local memory values, registers, stacks, and anything else that is required to continue the computation at v. We will view G as a directed graph by replacing each undirected edge between nodes u and v by two directed edges: one from u to v and the other from v to u. With each directed edge e and every time step t, we associate a *communication pebble* (c-pebble), $[e, t]$, which represents the message transmitted along edge e at time step t.

The host H will emulate each step t of G by creating an 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 has previously created s-pebble $\langle v,t-1\rangle$ and has received all of the c-pebbles $[e,t-1]$, where e is an edge into v. It takes unit time to create an s-pebble. It can create a c-pebble $[g,t]$ for an edge g out of v only if it contains an s-pebble $\langle v,t\rangle$. A node of H can also 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. Note that H can create more than one copy of an s-pebble or c-pebble. The ability of H to create redundant pebbles is crucial to our emulation schemes. In our emulations, each node of H is assigned a fixed set of nodes of G to emulate, and creates s-pebbles for them for each time step.

3.4.1 Assignment of nodes of G to nodes of H

We now show how to map the computation of G to the faulty butterfly H. The host H has $N^{1-\epsilon}$ arbitrarily distributed faults. We will first divide H into sub-butterflies of dimension $\epsilon \log N/2$, at levels $i\epsilon \log N/2$, for integer $i = 0$ to $2/\epsilon - 1$. (Without loss of generality, we will assume that $2/\epsilon$ and $\epsilon \log N/4$ are integers.) The butterfly contains as subgraphs $N^{1-\epsilon/2}$ sub-butterflies between each of these levels of division. There are a total of $(2/\epsilon)N^{1-\epsilon/2}$ sub-butterflies in all. The faults in the network may make some of these sub-butterflies unusable. We will identify "good" and "bad" sub-butterflies according to the following rules.

RULE 1: A sub-butterfly that contains a node that lies in a butterfly row in which there is a fault is a *bad* sub-butterfly (even if the fault lies outside of the sub-butterfly).

RULE 2: In order to apply Rule 2, we embed a Beneš network in the butterfly. The edges of the first stage of the Beneš network traverse the butterfly in increasing order of dimension and the edges of the second stage in decreasing order of dimension. The input nodes, the middle nodes, and the output nodes of the Beneš network are all embedded in level 0 of the butterfly (which is the same as level $\log N$). For $\delta = 2\epsilon/3$, identify the set of bad inputs/outputs (they are the same set here) according to the procedure outlined in the proof of Theorem 3.3.6 in Section 3.3.2. Any sub-butterfly that contains a node that has a bad input/output at the end of its butterfly row is a bad sub-butterfly. In the following two lemmas, we will bound the number of bad sub-butterflies.

Lemma 3.4.1 *For any $\epsilon > 0$, the number of rows in which there is either a fault or a bad input or output is at most $N^{1-\epsilon} + 2N^{1-2\epsilon/3}$.*

Proof: The number of rows containing a fault is at most $N^{1-\epsilon}$, since there are at most $N^{1-\epsilon}$ faults. By Theorem 3.3.6, for $\delta = 2\epsilon/3$, the number of bad inputs and outputs is at most $2N^{1-\delta} = 2N^{1-2\epsilon/3}$. □

Lemma 3.4.2 *For sufficiently large N and any fixed $\epsilon > 0$, at least half the sub-butterflies of H are good.*

Proof: The total number of sub-butterflies is $(2/\epsilon)N^{1-\epsilon/2}$. By Lemma 3.4.1, the number of rows containing either a fault or bad input or output is at most $N^{1-\epsilon} + 2N^{1-2\epsilon/3}$.

Since each bad row passes through $2/\epsilon$ different sub-butterflies, the total number of sub-butterflies identified as bad by Rules 1 and 2 cannot exceed $2(N^{1-\epsilon}+2N^{1-2\epsilon/3})/\epsilon$. Observe that $2(N^{1-\epsilon}+2N^{1-2\epsilon/3})/\epsilon$ is at most $N^{1-\epsilon/2}/\epsilon$, for sufficiently large N. □

Now we will divide the guest G into overlapping sub-butterflies of dimension $\epsilon \log N/2$ and map them to the good sub-butterflies of H. For any node v of G, two nodes of H will receive the initial state of the computation of node v, i.e., s-pebble $\langle v, 0\rangle$. These two nodes in H will create the s-pebbles for v. The mapping proceeds as follows. Take the guest G and cut it into sub-butterflies at levels $i\epsilon \log N/2$, for integers $i = 0$ to $2/\epsilon - 1$. Map each sub-butterfly to a good sub-butterfly of H in such a way that at most two sub-butterflies of G are mapped to each good sub-butterfly of H. Now cut G again to produce a second set of sub-butterflies, this time at levels $(\epsilon/4 + i\epsilon/2)\log N$, for integers $i = 0$ to $2/\epsilon - 1$. Map the sub-butterflies evenly to good sub-butterflies of H as before. The mapping of each sub-butterfly of G to a sub-butterfly of H is one-to-one. Thus at most four nodes of G are mapped to each node of H. Further, note that each sub-butterfly belonging to the first set intersects $N^{\epsilon/4}$ sub-butterflies belonging to the second set and vice-versa.

3.4.2 Building constant-congestion paths

We define the boundary nodes of G and H as follows. We call the nodes v of G belonging to levels $i\epsilon \log N/4 - 1$ and $i\epsilon \log N/4$, for $i = 0$ to $4/\epsilon - 1$, *boundary nodes* since each lies on the boundary of some sub-butterfly that was cut out of G. Let the set of boundary nodes of G be denoted by $\mathcal{B}_{\mathcal{G}}$. Similarly we define the nodes in the levels where H was cut to form sub-butterflies, i.e., levels $i\epsilon \log N/2$, and $i\epsilon \log N/2 - 1$, for $i = 0$ to $2/\epsilon - 1$, the boundary nodes of H. Let us denote this set $\mathcal{B}_{\mathcal{H}}$.

Let ϕ be the function that maps an s-pebble, $\langle v, t\rangle$ to the node in H that creates it. The creation of $\langle v, t\rangle$ requires that node $\phi(\langle v, t\rangle)$ of H gets all the c-pebbles $[e, t]$ from some other node of H, for every edge e into v. Suppose that $\langle v, t\rangle$ is mapped to some node $m = \phi(\langle v, t\rangle)$ in the interior of a good sub-butterfly of H. Then the neighbors of m in H also contain the s-pebbles of the neighbors of v in G. In this case m will receive the required c-pebbles from all its neighbors in H.

On the other hand, if the s-pebble for v is mapped to some node $m \in \mathcal{B}_{\mathcal{H}}$, the neighbors of m in H may not create the s-pebbles of the neighbors of v in G. However, since every node v of G is mapped to two nodes of H, there is another node m' of H which also creates an s-pebble for v. It is important to note that by the property of our mapping *m' is necessarily a node in the center of a sub-butterfly of H*, i.e., in level $\epsilon \log N/4$ or $\epsilon \log N/4 - 1$ of the sub-butterfly. Node m' of H will send to node m c-pebbles for *all* of the edges e into v.

To facilitate the transmission of c-pebbles we will establish constant-congestion fault-free paths in H, using the results of Section 3.3.2, between all pairs of nodes m and m' of H that create the s-pebbles for the same node v in $\mathcal{B}_{\mathcal{G}}$. The number of paths originating in a row of H is at most the number of nodes mapped to sub-butterfly boundaries in that row, which is at most $4 \cdot 2/\epsilon$, i.e., a constant. Similarly the number of paths ending in any row is $8/\epsilon$. We can divide the paths into $8/\epsilon$ sets such that each set has at most one path originating in a row and one path ending in a row. Note that all paths start and end in rows which have good inputs and outputs for doing Beneš-type routing. Therefore, each set can be routed with constant congestion using the results of Section 3.3.2. Since there are only a constant number of such sets the total congestion is also a constant.

3.4.3 The emulation

We will now formally describe the emulation and prove its properties. Initially, nodes of H contain s-pebbles $\langle v, 0\rangle$ for nodes v of G. We say that H has emulated T steps of the computation of G iff for every node v, an s-pebble $\langle v, T\rangle$ has been created. The emulation algorithm is executed by every node m of H and 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 already contains s-pebble $\langle v, t-1\rangle$ and c-pebbles $[e, t-1]$ for every edge e into v.

2. Communication step. For every node v whose s-pebble was updated from $\langle v, t-1\rangle$ to $\langle v, t\rangle$ in the computation step, node m sends a c-pebble to each of its neighbors in the sub-butterfly of H that contains m. In addition, if m is a node in one of the center levels (levels $\epsilon \log N/4$ or $\epsilon \log N/4 - 1$ in the sub-butterfly), it starts four c-pebbles, one for each edge out of v, along on their way to the node m' that also creates s-pebbles for v.

3. Routing step. Node m moves every c-pebble $[e, t]$ that is passing through it en route to its destination, one step closer to that destination.

Lemma 3.4.3 *Each macro-step takes only a constant number of time steps to execute.*

Proof: There are at most 4 s-pebbles mapped to each node. Therefore the computation step takes constant time. Every s-pebble that is updated can cause at most 8 c-pebbles to be sent. Therefore the communication step takes only constant time. There can be only a constant number of c-pebbles in transit residing at any node at any time step. The reason is that there are only a constant number of paths passing through every node. Furthermore, 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. □

Theorem 3.4.4 *Any computation on a fault-free butterfly G that takes time T can be emulated in $O(T + \log N)$ time by H.*

Proof: We will show that only $O(T + \log N)$ macro-steps are required for a T-step computation of G. The final result will then follow from Lemma 3.4.3.

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. Let ϕ be the function that maps an s-pebble, $\langle v,t\rangle$ to the node in H that creates it. 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 a c-pebble $[e,t-1]$ for each edge e into node v in G. These c-pebbles are sent by other s-pebbles. There are two cases. If $\phi(\langle v,t\rangle)$ is a node in $\mathcal{B}_{\mathcal{H}}$, then s-pebble $\langle v,t\rangle$ gets all of its c-pebbles from the other s-pebble for node v which we will denote by $\langle v,t-1\rangle'$. If $\phi(\langle v,t\rangle)$ is not a node in $\mathcal{B}_{\mathcal{H}}$, then it gets its c-pebbles from s-pebbles $\langle u,t-1\rangle$ such that u is a neighbor of v in G and $\phi(\langle u,t-1\rangle)$ is a neighbor of $\phi(\langle v,t\rangle)$ in H. 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 $\langle v,t-1\rangle'$ in the first case and the dependency trees of $\langle v,t-1\rangle$ and all s-pebbles $\langle u,t-1\rangle$ in the second case.

We will bound the number of macro-steps, T', that is required by H to emulate a T-step computation of G. Let $\langle v,T\rangle$ be an s-pebble that was updated in the last macro-step. We will now look at the dependency tree of $\langle v,T\rangle$. 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*, $s_T, s_{T-1},\ldots,s_0$, of tree nodes from the root to the leaves of the tree as follows. The root of the tree, s_T, is $\langle v,T\rangle$. The creation of s_T requires the s-pebble $\langle v,T-1\rangle$ and c-pebbles $[e,T-1]$. 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 which sent the c-pebble that arrived last at node $\phi(\langle v,T\rangle)$. After choosing s_{T-1}, we choose the rest of the sequence recursively in the subtree with s_{T-1} as the root. We will 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 and because a c-pebble moves once in every macro-step, $\tau(s_i)-\tau(s_{i-1})=l_i$. Thus,

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

Now suppose that some l_i is greater than one. This corresponds to some long path taken by some c-pebble to go from $\phi(s_{i-1})$ in the center level of a sub-butterfly of H to $\phi(s_i)$ in $\mathcal{B}_{\mathcal{H}}$. Thus l_i is the length of the path taken in H, which is at most $4\log N$. The key observation is that since $\phi(s_{i-1})$ is a node in the center level, working down the tree from s_{i-1} there can be no more long paths until we reach an s-pebble mapped to the boundary $\mathcal{B}_{\mathcal{H}}$, i.e., $l_{i-j}=1$ for $1\le j\le \epsilon\log N/4-1$. Thus $T'=\sum_{0<i\le T} l_i$ can be no more than $(16/\epsilon+1)T+4\log N$ which is $O(T+\log N)$. □

We can extend these results to the shuffle-exchange network using Schwabe's proof [Sch90] that an N-node butterfly can emulate an N-node shuffle-exchange network with constant slowdown, and vice versa.

Theorem 3.4.5 *Any computation on a fault-free N-node shuffle-exchange network G which takes time T can be emulated in $O(T+\log N)$ time by a an N-node shuffle-exchange network H with $N^{1-\epsilon}$ worst-case faults, for any constant $\epsilon>0$.*

Proof: Schwabe [Sch90] shows how to emulate the computation of a fault-free butterfly on a fault-free shuffle-exchange network and vice versa. First we use this result to map the computation of a butterfly B to the faulty shuffle-exchange network H. Any node of B that is mapped to a faulty node of H is declared faulty. If there is any routing path required for the emulation that passes through this faulty node of H, then we declare the nodes of B that use this path to be faulty. The number of faults in B will only be a constant factor more than $N^{1-\epsilon}$, since both the load and the congestion of the paths used in the emulation are constant. Now we use Theorem 3.4.4, to emulate a fault-free butterfly, B', on B. We again use Schwabe's result to emulate the fault-free shuffle-exchange network, G, on the fault-free butterfly B'. Each of these emulations has constant slowdown. Therefore the entire emulation of G on H has constant slowdown. □

3.4.4 Emulating normal algorithms on the hypercube

Many practical computations on the hypercube are structured. (Please refer to Section 3.1 for the definition of a hypercube.) The class of algorithms in which every node of the hypercube uses exactly one edge for communication at every time step and further all of the edges used in a time step belong to the same dimension of the hypercube are called *leveled algorithms* (also known as *regular algorithms* [BCS92]). A useful subclass of leveled algorithms are *normal algorithms*. A normal algorithm has the additional restriction that the dimensions used in consecutive time steps are consecutive. Many algorithms such as bitonic sort, FFT, and tree-based algorithms like branch-and-bound can be implemented on the hypercube as normal algorithms [Lei92]. An additional property of normal algorithms is that they can be emulated efficiently by bounded-degree networks such as the shuffle-exchange network and the butterfly. We state a result due to Schwabe [Sch91] to this effect.

Lemma 3.4.6 *An N-node butterfly can emulate any normal algorithm of an N-node hypercube with constant slowdown.*

We require the following well known result on embedding a butterfly in a hypercube (For a stronger result that the butterfly is a subgraph of the hypercube, please refer to [GHR90]).

Lemma 3.4.7 *An N-node butterfly can be embedded in an N-node hypercube with constant load, congestion, and dilation.*

Theorem 3.4.8 *An N-node hypercube with $N^{1-\epsilon}$ worst-case faults (for any fixed $\epsilon > 0$) can emulate any normal algorithm on an N-node fault-free hypercube with only constant slowdown.*

Proof: Let the faulty N-node hypercube be H and the fault-free N-node hypercube be G. H has some set of $N^{1-\epsilon}$ faulty nodes. H emulates any normal algorithm of G using a sequence of constant-slowdown simulations. Let an N-node butterfly, B, be embedded in H in the manner of Lemma 3.4.7. Any node of B which is mapped onto a faulty node of H will be considered faulty. Since this is a constant load embedding the number of faulty nodes in B is $O(N^{1-\epsilon})$. Clearly, H can emulate any computation of B (the faulty nodes of B do no computation) with constant slowdown using the constant load, dilation and congestion embedding of B in H. Let B' be a fault-free N-node butterfly. From Theorem 3.4.4, B can emulate B' with a constant slowdown. Now, from Lemma 3.4.6, B' can emulate any normal algorithm of G with a constant slowdown. Putting all these simulations together, we can obtain a constant-slowdown simulation of any normal algorithm on G on the faulty hypercube H. $\square$

3.5 Random faults

In this section we show that an N-input host butterfly H can sustain many random faults and still emulate a fault-free N-input guest butterfly G with little slowdown. In particular, we show that if each node in H fails independently with probability $p = 1/\log^{(k)} N$, where $\log^{(k)}$ denotes the logarithm function iterated k times, the slowdown of the emulation is $2^{O(k)}$, with high probability. For any constant k this slowdown is constant. Furthermore, for some $k = O(\log^* N)$ the node failure probability, p, is constant, and the slowdown is $2^{O(\log^* N)}$. Previously, the most efficient self-emulation scheme known for an N-input butterfly required $\omega(\log\log N)$ slowdown [Tam92a].

The proof has the following outline. We begin by showing that the host, H, can emulate another N-input butterfly network B_k with constant slowdown. As in H, some of the nodes in B_k may fail at random (in which case it is not necessary for H to emulate them), but B_k is likely to contain fewer faults than H. In turn, B_k can emulate another butterfly B_{k-1} with even fewer faults. Continuing in this fashion, we arrive at B_1, which with high probability, contains so few faults that it can emulate the guest, G, with constant slowdown. There are $k+1$ emulations, and each incurs a constant factor slowdown, so the total slowdown is $2^{O(k)}$.

3.5.1 Emulating a butterfly with fewer faults

We begin by explaining how H emulates B_k. The first step is to cover the N-input butterfly B_k with overlapping $(\log^{(k)} N)^2$-input sub-butterflies. For ease of notation, let $M_k = (\log^{(k)} N)^2$ (For simplicity, we assume that $\log M_k$ is an integral multiple of 4). For each i from 0 to $4\log N/\log M_k$, there is a band of disjoint M_k-input sub-butterflies in B_k spanning levels $(i\log M_k)/4$ through $((i+4)\log M_k)/4 - 1$. We call these sub-butterflies the *band* i sub-butterflies. Note that each band i sub-butterfly shares $M_k^{3/4}$ rows with $M_k^{1/4}$ different band $i-1$ sub-butterflies, and $M_k^{3/4}$ rows with $M_k^{1/4}$ different band $i+1$ sub-butterflies.

Each M_k-input sub-butterfly in B_k will be emulated by the corresponding sub-butterfly in H. We say that an M_k-input sub-butterfly in B_k *fails* if more than $\alpha\sqrt{M_k}\log M_k$ nodes inside the corresponding M_k-input sub-butterfly in H fail, where α is a constant that will be determined later. If a sub-butterfly in B_k fails, then H is not required to emulate any of the nodes that lie in that sub-butterfly. As we shall see, if it does not fail, then the corresponding sub-butterfly in H contains few enough faults that we can treat them as worst-case faults, and apply the technique from Section 3.4 to reconfigure around them.

The following lemma bounds the probability that a sub-butterfly in B_k fails.

Lemma 3.5.1 *For $\alpha > 2e$, an M_k-input sub-butterfly in B_k fails with probability at most $1/\log^{(k-1)} N$.*

Proof: An M_k-input sub-butterfly fails if the corresponding M_k-input sub-butterfly in H contains more than $\alpha M_k \log M_k$ faults. An M_k-input sub-butterfly in H contains a total of $M_k \log M_k = 2(\log^{(k)} N)^2(\log^{(k+1)} N)$ nodes, each of which fails with probability $1/\log^{(k)} N = 1/\sqrt{M_k}$. Thus, the expected number of nodes that fail is $\sqrt{M_k}\log M_k = 2(\log^{(k)} N)(\log^{(k+1)} N)$. Since each node fails independently, we can bound the probability that more than $\alpha\sqrt{M_k}\log M_k$ nodes fail using a Chernoff-type bound. For $\alpha > 2e$, the probability that more than $\alpha\sqrt{M_k}\log M_k$ nodes fail is at most $2^{-\alpha\sqrt{M_k}\log M_k}$ (for a proof, see [Rag90]). Since $\alpha\sqrt{M_k}\log M_k > \log^{(k)} N$, this probability is at most $2^{-\log^{(k)} N} = 1/\log^{(k-1)} N$. □

The next lemma shows that if a sub-butterfly in B_k does not fail, then the corresponding sub-butterfly in H can emulate it with constant slowdown.

Lemma 3.5.2 *If an M_k-input sub-butterfly in B_k does not fail, then the corresponding sub-butterfly in H can emulate it with constant slowdown.*

Proof: Since the number of faults in an M_k-input sub-butterfly that does not fail is most $\sqrt{M_k}\log M_k$, we can treat them as worst-case faullts and apply Theorem 3.4.4 with $\epsilon \approx 1/2$. □

The next lemma shows that the host H can emulate any computation performed by an N-input butterfly network B_k with constant slowdown. Recall that H is not required to emulate nodes in B_k that lie in sub-butterflies in B_k that have have failed.

Lemma 3.5.3 *The host H can emulate B_k with constant slowdown.*

Proof: By Lemma 3.5.2, each M_k-input sub-butterfly in B_k that has not failed can be emulated by the corresponding sub-butterfly in H with constant slowdown using the technique of Section 3.4. (Note that each node in B_k may be emulated by as many as four different sub-butterflies in H.) In order to emulate the entire network B_k, it is also necessary to emulate the connections between the sub-butterflies. As in Section 3.4, let $M_k^{1-\epsilon}$ denote the number of faults in an M_k-input sub-butterfly of H. For a sub-butterfly that has not failed, $\epsilon \approx 1/2$. By Lemma 3.4.1, the number of rows containing either a fault or an input or output that is bad for Beneš routing is at most $M_k^{1-\epsilon} + 2M_k^{1-2\epsilon/3}$, which is approximately $2M_k^{2/3}$. Each band i sub-butterfly that does not fail shares $M_k^{3/4}$ rows with each of the band $i-1$ sub-butterflies (and band $i+1$ sub-butterflies) with which it overlaps. Thus, for each pair of overlapping butterflies, most of the shared rows are both fault-free and good for routing in both sub-butterflies. The emulation strategy of Section 3.4 covers each M_k-input sub-butterfly in H with smaller sub-butterflies, each having $M_k^{\epsilon/2}$ inputs. If a smaller sub-butterfly is used in the emulation, then none of the rows that pass through it contain either a fault or a bad input or output. Thus, the $M_k^{3/4}$ connections between the two sub-butterflies in bands i and $i-1$ (and $i+1$) can be emulated by routing constant-congestion paths of length $O(\log M_k)$ through the shared rows. The rest of the proof is similar to that of Theorem 3.4.4. $\square$

3.5.2 Emulating a series of butterflies

So far we have shown that the host H can emulate an N-input butterfly B_k that contains some faulty nodes. Although our ultimate goal is to show that H can emulate the guest network G, which contains no faulty nodes, we have made some progress. In the host network, H, each node fails independently with probability $1/\log^{(k)} N$. In B_k, each $(\log^{(k)} N)^2$-input sub-butterfly fails with probability $1/\log^{(k-1)} N$. A node in B_k fails if it lies in a sub-butterfly that fails. Since each node in B_k lies in at most four $(\log^{(k)} N)^2$-input sub-butterflies we have reduced the expected number of faults from $(N \log N)/\log^{(k)} N$ in H to fewer than $4N \log N/\log^{(k-1)} N$ in B_k.

The next step is to show that butterfly B_k can emulate a butterfly B_{k-1} with even fewer faults. In general, we will cover butterfly B_j with $(\log^{(j)} N)^2$-input sub-butterflies. For ease of notation, let $M_j = (\log^{(j)} N)^2$. We say that an M_j-input sub-butterfly in B_j fails if the corresponding M_j-input sub-butterfly in B_{j+1} contains more than $\alpha\sqrt{M_j}$ M_{j+1}-input sub-butterflies that have failed. The following three lemmas are analogous to Lemmas 3.5.1 through 3.5.3.

Lemma 3.5.4 *For $\alpha > 8e$, the probability that an M_j-input sub-butterfly in B_j fails is at most* $1/(\log^{(j-1)} N)$.

Proof: The proof is by induction on j, starting with $j = k$ and working backwards to $j = 0$. The base case is given by Lemma 3.5.1. For each value of i from 0 to $4 \log M_j/\log M_{j+1}$, there is a band of disjoint M_{j+1}-input sub-butterflies in B_{j+1} that span levels $(i \log M_{j+1})/4$ through $((i+4)\log M_{j+1})/4$. These M_{j+1}-input sub-butterflies can be partitioned into four disjoint *classes* according to their band numbers. Two bands of sub-butterflies belong to the same class if their band numbers differ by a multiple of four. There are at most

$$M_j \log M_j / M_{j+1} \log M_{j+1} = (\log^{(j)} N)^2/(\log^{(j+1)} N)(\log^{(j+2)} N)$$

sub-butterflies in each of these classes, and within a class, the sub-butterflies are disjoint. By induction, each sub-butterfly fails independently with probability at most $1/\log^{(j)} N$. Thus, in any particular class, the expected number of sub-butterflies that fail is at most $(\log^{(j)} N)/(\log^{(j+1)} N)(\log^{(j+2)} N)$, which is less than $\log^{(j)} N$. Using Chernoff-type bounds as in Lemma 3.5.1, for $\alpha > 8e$, the probability that more than $(\alpha/4)\log^{(j)} N = (\alpha/4)\sqrt{M_j}$ of these sub-butterflies fail is at most $2^{-(\alpha/4)\log^{(j)} N}$, which is less than $1/4 \log^{(j-1)} N$. Thus, the probability that a total of $\alpha \log^{(j)} N$ sub-butterflies fail in the four classes is at most $1/\log^{(j-1)} N$. $\square$

Lemma 3.5.5 *If an M_j-input sub-butterfly in B_j does not fail, then the corresponding M_j-input sub-butterfly in B_{j+1} can emulate it with constant slowdown.*

Proof: If an M_j-input sub-butterfly in B_j does not fail, then at most $\alpha\sqrt{M_j} = \alpha \log^{(j)} N$ of the overlapping M_{j+1}-input sub-butterflies in the corresponding M_j-input sub-butterfly in B_{j+1} fail. Each of these sub-butterflies contains $M_{j+1} \log M_{j+1} = 2(\log^{(j+1)} N)^2 \log^{(j+2)} N$ nodes. Since the total number of nodes in all of these sub-butterflies is at most $2\alpha \log^{(j)} N (\log^{(j+1)} N)^2 \log^{(j+2)}$ i.e., approximately $\sqrt{M_j}$, we can treat them all as if they were worst-case faults and apply Theorem 3.4.4 with $\epsilon \approx 1/2$. □

Lemma 3.5.6 *For $1 \leq j < k$, butterfly B_{j+1} can emulate B_j with constant slowdown.*

Proof: The proof is similar to the proof of Lemma 3.5.3. □

Theorem 3.5.7 *For any fixed $\gamma > 0$, with probability at least $1 - 1/2^{N^{1-\gamma}}$, an N-input butterfly in which each node fails with probability $1/\log^{(k)} N$ can emulate a fault-free N-input butterfly with slowdown $2^{O(k)}$.*

Proof: The host network, $H = B_{k+1}$, can emulate network B_1 with a net slowdown equal to the product of the slowdowns of the emulations of B_j by B_{j+1}, $1 \leq j \leq k$. Since we have shown that each of these emulations have constant slowdown, the slowdown of the emulation of B_1 by H is $2^{O(k)}$. In B_1, each sub-butterfly with $(\log N)^2$ inputs fails with probability $1/\log^{(0)} N = 1/N$. Using Chernoff-type bounds as in Lemma 3.5.1, the probability that more than $N^{1-\gamma}$ of these sub-butterflies fail is at most $1/2^{N^{1-\gamma}}$. If fewer than $N^{1-\gamma}$ of them fail, then we can treat the nodes contained in these sub-butterflies as if they were worst-case faults. In this case, the total number of worst-case faults is at most $2N^{1-\gamma} \log^2 N \log\log N$. Hence, by applying Theorem 3.4.4 with $\epsilon \approx \gamma$, B_1 can emulate the guest network G with constant slowdown. □

3.6 Open problems

Some of the interesting problems left open by this chapter are listed below.

1. Can the butterfly (or any other bounded-degree network) tolerate random faults with constant failure probability and still simulate itself with constant slowdown? (Please refer to Conjecture 4.3.9.)

2. Can an N-node butterfly (or any other N-node bounded-degree network) tolerate *more* than $N^{1-\epsilon}$ worst-case faults (e.g., $N/\log N$) and still simulate itself with constant slowdown?

3. Can an N-input Beneš network tolerate $\Theta(N)$ worst-case switch failures and still route disjoint paths or constant congestion paths in any permutation between some set of $\Theta(N)$ inputs and outputs?

4. Can an N-input butterfly be embedded with constant load, congestion, and dilation in an N-input butterfly with more than $\log^{O(1)} N$ (e.g., $N^{1-\epsilon}$) worst-case faults?

Chapter 4

Fault Tolerance of Processor Arrays

4.1 Introduction

One of the most popular ways to construct a parallel computer is to arrange the processors as a two-dimensional array. Commercial machines including the Intel Touchstone, and MasPar MP-1 have this topology, as do experimental machines such as the J-Machine and Mosaic. In this chapter 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 some fixed $\epsilon < 1$, and still emulate a fault-free $N \times N$ array with constant slowdown.

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-lengh 8, nodes $(3, 2, 4, 8)$ and $(3, 2, 3, 8)$ are neighbors, while $(3, 2, 4, 8)$ and $(3, 2, 3, 7)$ are not. A two-dimensional array is also called a *mesh* and is shown in Figure 5. The nodes labeled (j, i), $1 \le i \le N$, are said to belong to the j^{th} row of the mesh. The nodes labeled (i, j), $1 \le i \le N$ are said to belong to the j^{th} column of the mesh. The nodes in an array represent processors and the edges represent communication links. An array is assumed to be synchronous. At each time step each edge can transmit a single message. 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 chapter hold whether or not the array has wrap-around edges.

We use the same fault model as Chapter 3. We assume that faults are static and that their locations are known. 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. In this chapter, we deal primarily with the worst-case fault model. All

This chapter describes joint research with Bruce Maggs and Richard Cole.

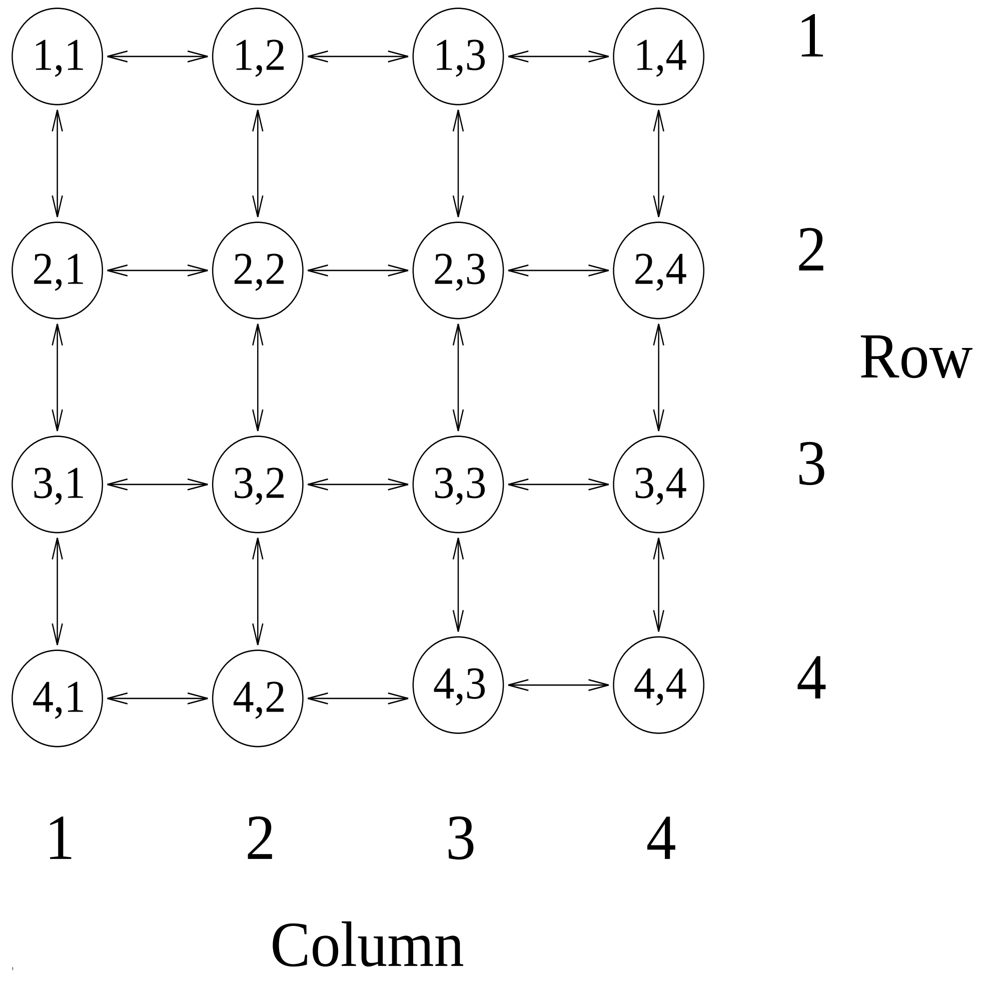

Figure 5: A 4×4 Mesh.

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 4.4, we use a weaker fault model for one-dimensional arrays by allowing faulty nodes to communicate, though they cannot compute. We show that even in this weaker fault model the linear array cannot tolerate as many worst-case faults as two- or higher- dimensional arrays.

4.1.1 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. As in Chapter 3, we call H the *host* graph and G the *guest* graph. An embedding maps nodes of G to non-faulty nodes of H, and edges of G to non-faulty paths in H. Recall that the three important measures of an embedding are its load (l), congestion (c), and dilation (d). Leighton, Maggs, and Rao [LMR88] 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, any embedding of an N-node (2- or 3-dimensional) array into an array of the same size containing more than a constant number of worst-case faults, or $\Theta(N)$ random faults must have more than constant load or dilation [GE84, KKL+90]. 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.

4.1.2 Redundant computation

All of the emulations in this chapter use a technique called *redundant computation.* Recollect that we used this technique in Chapter 3 to tolerate faults in hypercubic networks. 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. In this section, we describe informally the basic ideas involved in using this technique to tolerate faults in the mesh. A complete and formal description will be provided in the succeeding sections of this chapter.

The ideal way for a host mesh to emulate a guest mesh is to map each guest node to the corresponding host node. However, this strategy will only work if there are no faults in the host graph. If the host graph contains faults, then some regions in the host may contain too many faults to emulate the corresponding regions in the guest. We call each such region in the host a core. The cores will not be used in the emulation. Instead, we use the technique of redundant computation to emulate the regions of the guest that correspond to cores in the host.

The technique can be most easily understood by examining the following case. Suppose that the host is $3k \times 3k$ mesh and contains a single core which is a $k \times k$ square submesh located at the host's center, and all of the faults in the host lie in this core. In this case, we divide the guest mesh into two regions that overlap called the outerskirt and the patch. The patch is a $2k \times 2k$ square and the outerskirt is a "square annulus" of width $2k$ (see Figure 6). The width of the region of overlap of the patch and the outerskirt is k. The outerskirt can be embedded in a one-to-one fashion into the corresponding region of the host. The patch can also be embedded with constant load, dilation, and congestion into a region of the host that contain no faults, i.e., a region of the host not intersecting the core. Note that some nodes of the guest are in both the patch and the outerskirt. The computation performed by these nodes will be emulated in two different nodes of H.

The host H can now emulate the patch and the outerskirt *independently* with constant slowdown. The only problem in this emulation arises due to the fact that the patch does not contain all the neighbors (in G) of the nodes on its border. Similarly, the outerskirt does not contain all the neighbors (in G) of the nodes in its (inner) border. Therefore, these border nodes will not receive inputs from some of their neighbors in this emulation and

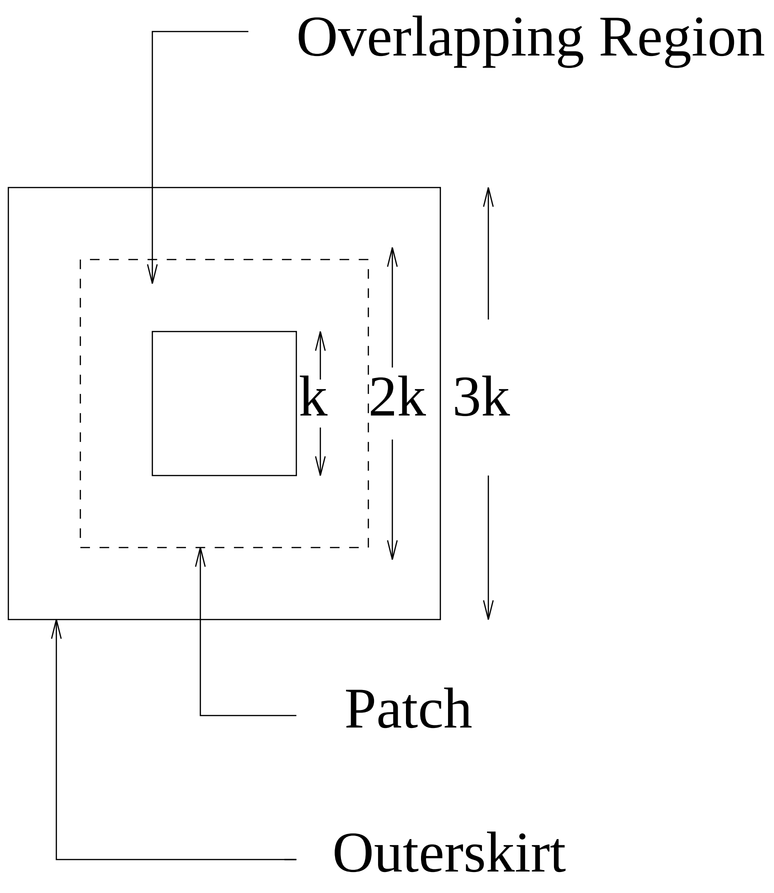

Figure 6: The patch and the outerskirt

cannot be emulated even for one step. In fact, it can be seen that nodes at a distance $i-1$ from the border of the patch or the (inner) border of the outerskirt will stop computing in the i^{th} step for the lack of inputs from some of their neighbors. But observe that if we emulate for less than k steps, i.e., for less than the width of the overlapping region, at least one copy of the computation of *every node* of G computes for *all* k steps. This is crucial since after k steps of the emulation, the copy of the computation that computed all k steps can *update* the copy of the computation that stopped computing. This update is performed by routing update messages and takes time proportional to the size of the host, i.e., $\Theta(k)$ steps. Thus, after the update is complete, *every* copy of the computation of G has computed for k steps and the total time taken is $\Theta(k)$. Repeating this over and over again yields a constant-slowdown emulation of G on H.

The actual emulation algorithm described later in this chapter is in some ways simpler than the one presented here. However the emulation scheme presented here best illustrates the fundamental principles involved.

4.1.3 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 [Rag89] 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 $p \leq .29$, any packet that can reach its destination does so in $O(N \log N)$ steps, with high probability. In [Mat92], Mathies improves the $p \leq .29$ bound to $p \approx .4$.

Kaklamanis et al. [KKL+90] improved upon Raghavan's result by devising an $O(N)$-step deterministic algorithm. Their 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 $N \times N$ array with constant-probability failures or up to $O(N)$ worst-case faults can sort or route some set of N^2 items in $O(N)$ time. They also showed that, with high probability, an $N \times N$ array with constant-probability failures can emulate a fault-free $N\sqrt{\log N} \times N\sqrt{\log N}$ array with $O(\log N)$ slowdown.

Aumann and Ben-Or [AB92] use Rabin's information dispersal technique [Rab89] 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 assume 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.

In [BCH91], [BCH92a], and [BCH92b], Bruck, Cypher, and Ho show that by adding k spare processors to an array, it is possible to tolerate up to k worst-case node or edge failures and still find a working fault-free array as a subgraph of the faulty-array. The degree of each node, however, is proportional to k. Ajtai et al. analyze the technique of adding spare processors to larger classes of graphs which include arrays in [AAB+92].

4.1.4 Our results

In Section 4.2 we show that an $N \times N$ array can tolerate $\log^{O(1)} N$ worst-case faults and still emulate a fault-free array with constant slowdown. Previously it was only known that a constant number of worst-case faults could be tolerated with constant slowdown. Section 4.2 introduces most of the terminology that is used throughout this chapter. If a faulty node is allowed to communicate, but not compute, then the reconfiguration scheme presented in this section can be used to tolerate $\log^{O(1)} N$ worst-case faults on an N-node one-dimensional array with constant slowdown.

Two- or higher-dimensional arrays can tolerate even more worst-case faults with constant slowdown. In Section 4.3 we present a method called *multi-scale emulation* for tolerating $N^{1-\epsilon}$ worst-case faults on an $N \times N$ array with constant slowdown, for some fixed $\epsilon < 1$. This result nearly matches the $O(N)$ upper bound on the number of worst-case faults that can be tolerated with constant slowdown. Using the technique in Section 3.5, it is possible to use the scheme for tolerating $N^{1-\epsilon}$ worst-case faults to construct a scheme for tolerating constant-probability node failures with slowdown $2^{O(\log^* N)}$. Previously, the smallest slowdown known for constant-probability node failures was $O(\log N)$ [KKL+90].

In Section 4.4 we show that an N-node linear array cannot tolerate more than $\log^{O(1)} N$ worst-case faults without suffering more than constant slowdown, even if faulty nodes are allowed to communicate, provided 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 a and b. In this case we say that there are multiple *instances* of the guest node u. For each guest time step, 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 v' sends a packet for the edge e to u'. All known emulations are static (e.g., [Fel85, KLM+89, Sch90]).

4.2 A simple method for tolerating worst-case faults on the mesh

In this section, we show that an $N \times N$ mesh with $\log^{O(1)} N$ 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 will call this step the *growth process*. We will describe this process in Section 4.2.1. The next is an emulation technique that will map the computation of the fault-free mesh (the guest) to nodes in the faulty mesh (the host). The boxes grown in the first step will determine how the mapping of the computation is done. This is described Section 4.2.2. For simplicity, we assume that the mesh has wrap-around edges which connect nodes in the first row to the corresponding nodes in the last row and nodes in the first column to the corresponding nodes in the last column. This assumption can be easily done away with at the cost of considering some special cases for faults near the boundary of the mesh.

4.2.1 The growth process

The growth process grows boxes according to a set of rules 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. A finished box of side-length $3k$ consists of a core of side-length k surrounded by a *skirt* of width k as shown in Figure 7.

At every stage of the growth process, there is 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 take the smallest square box that contains both the

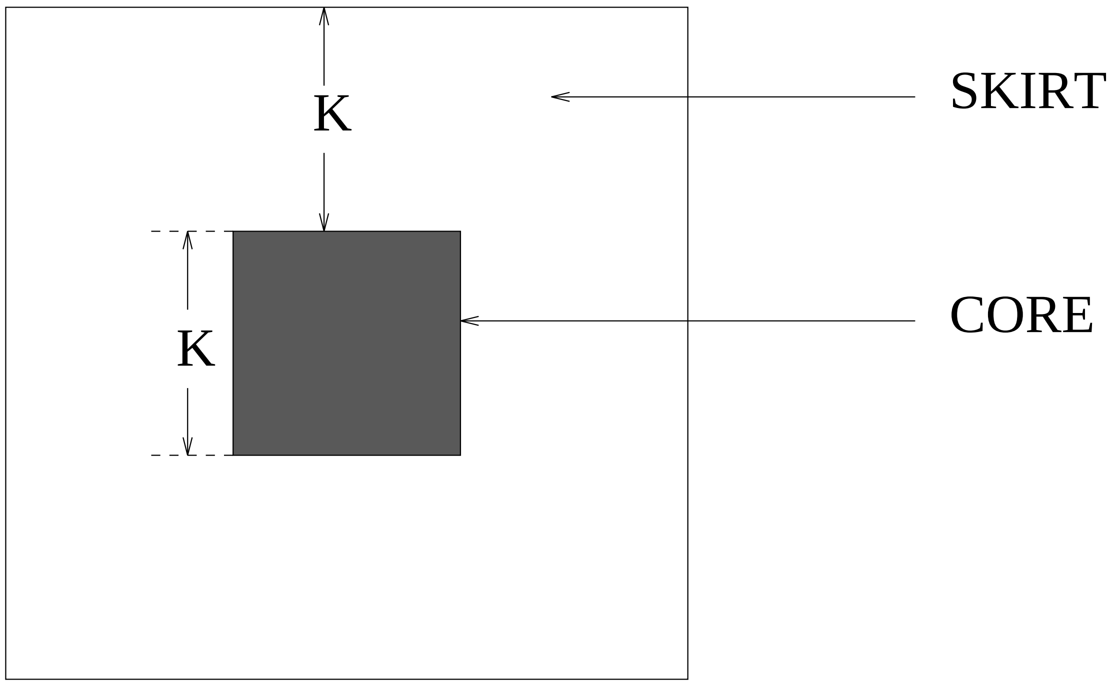

Figure 7: A finished box

cores and turn this bounding box into a core. Any old cores enclosed completely inside this newly formed core are removed from the set of cores. If the core or the skirt intersects a finished box, we find the 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 as 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 $N/3$. We now show that this cannot happen if there are fewer than $(\log N)/2$ faults.

Note that 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 a fault may be geometrically located in the region of a core and yet not be contained in that core.

Lemma 4.2.1 *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 contains. We will 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 the side-lengths of these two cores be denoted by x and y. 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

$$\begin{aligned} F(k) &\geq (\log \lfloor k/2 \rfloor)/2 + 2 \\ &\geq (\log k)/2 + 1 \end{aligned} \tag{8}$$

This proves our inductive hypothesis.

Now suppose that there is a core of side-length greater than $N/3$. Then there must be at least $F(\lfloor N/3 \rfloor + 1)$ faults, which is more than $\log N/2$. This is a contradiction. Thus there can never be a core of side-length more than $N/3$ in side-length. Therefore, the growth process must terminate with non-overlapping finished boxes. $\square$

4.2.2 The Emulation

In this section, we will 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 slow-down.

As in Section 3.4, 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. We will view G as a directed graph by replacing each undirected edge between nodes u and v by two directed edges: one from u to v and the other from v to u. With each directed edge e 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.

As in Section 3.4, the host H will emulate each step t of G by creating an 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. It can create a c-pebble $[g, t]$ for an edge g out of v only if it contains an s-pebble $\langle v, t \rangle$. A node of H can also 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. In our emulations, each node of H is assigned a fixed set of nodes of G to emulate, and creates s-pebbles for them for each time step.

Using the growth process of the previous section, we grow a collection of non-overlapping finished boxes on the faulty mesh H. If the faulty mesh H has fewer than $(\log N)/2$ faults placed arbitrarily by an adversary, then the growth process will terminate with non-intersecting finished boxes. Every node of H that does not belong to any of the finished boxes will simulate the computation of the corresponding node of G. Every finished box

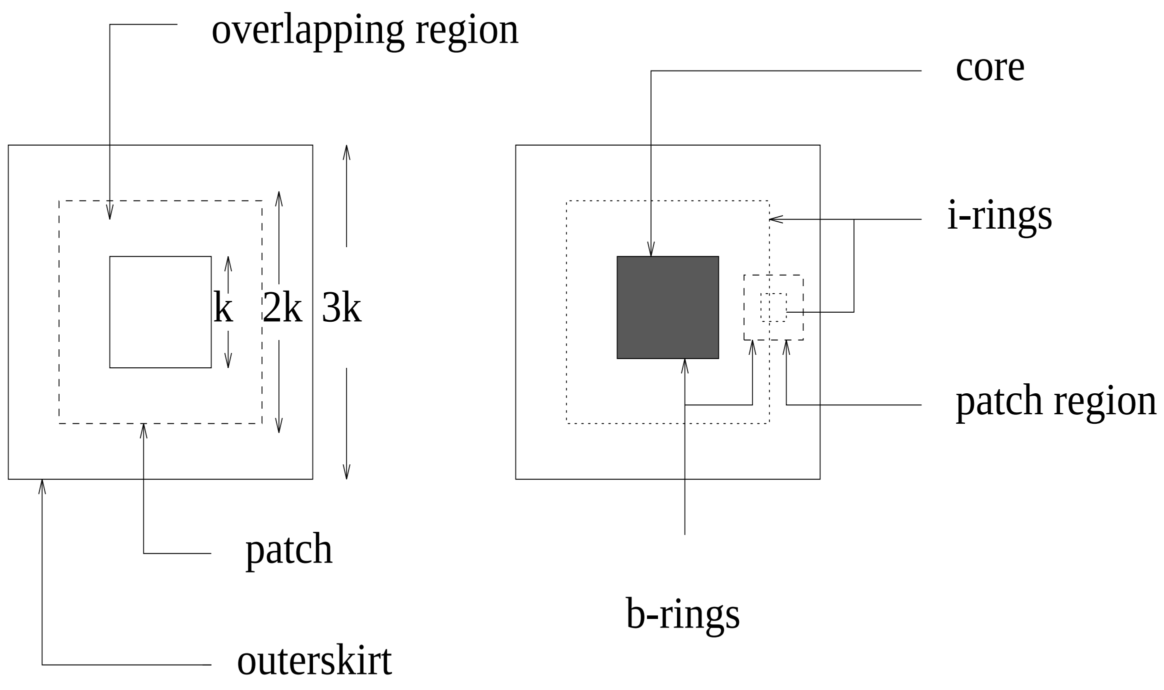

Figure 8: Mapping the computation inside a finished box

of H will be responsible for simulating the corresponding submesh of G. However since some of the nodes inside the core of the 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 the 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. To do this we use the technique of *redundant computation* for hiding communication latency. Therefore some nodes of G will be simulated by two different nodes 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 8). First, the submesh of G corresponding to a finished box is divided into two overlapping regions: the patch and the outerskirt. The patch is a square piece of the computation of side-length $2k$ as shown in Figure 8. The outerskirt consists of the entire submesh of G with a square of side-length k removed from its center. The patch and the outerskirt are mapped to the finished box as follows. The outerskirt is of the same size and shape as the skirt of the finished box. Therefore every node in the outerskirt is mapped to its corresponding node in the skirt. Let the finished box be of side-length $3k$. 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 8. 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. The nodes on the border of the patch have duplicates in the interior of the outerskirt which perform the same computation. The nodes of the finished box to which these duplicates are mapped form a ring. Similarly, the nodes in the border of the outerskirt have duplicates in the interior of the patch which are mapped to a ring in the finished box (See Figure 8). (Formally, a *ring* in a square region is a set of nodes that are equidistant from the border of the region.) We will call both these rings *interior rings*, or i-rings for short. The two rings in the finished box to which the border of the patch and the border of the outerskirt are mapped are called *border rings*, or b-rings for short.

A node H containing an s-pebble for a node v of G needs to receive c-pebbles corresponding to each of the edges into v in G in order for the computation to proceed. If this s-pebble is in the interior of the patch or the outerskirt, the s-pebbles of its neighbors are also in the patch or outerskirt respectively. Therefore it can obtain the required c-pebbles from these nodes in one time step. However, the s-pebbles in the border of the patch or the outerskirt have may not have neighboring s-pebbles close by in H. Since the patch and the outerskirt overlap, every s-pebble in the border of the patch or the outerskirt has an s-pebble for the same node computation in the interior of the outerskirt or the patch respectively. In our emulation, every s-pebble in the border will receive the c-pebbles for *all* of its incoming edges from its duplicate copy that is mapped to one of the i-rings in the finished box.

We now lay down constant-congestion paths from each s-pebble on an i-ring to its duplicate on a b-ring. The c-pebbles will be routed along these paths during the emulation. Note that these paths are determined off-line before the start of the emulation. Each b-ring and i-ring is a square with 4 sides. The nodes on each side of an i-ring need to be connected to the appropriate nodes on the appropriate side of the corresponding b-ring. Given two sides whose nodes need to be connected, draw two paths connecting the endpoints of one side to the endpoints of the other side. The pairs of endpoints that are chosen to be connected by a path are such that the paths and the two sides enclose a region. Further, the paths are chosen such that that this enclosed region does not intersect the core and this region can be partitioned into squares of side-length $k/4$, where $3k$ is the side-length of the finished box. From Figure 8, it can be seen that it is possible to choose such paths for every pair of sides. Now route all the required paths inside the enclosed region by using the squares as simple crossbars. Each square is used to either pass paths right across or to turn all or a subset of these paths by 90 degrees. It is clear that paths have length $\Theta(k)$. The congestion created by these paths is a constant. Since there are only eight such pairs of sides to be connected, the total congestion of these paths is also a constant.

We now describe the actual emulation. The algorithm for emulation is similar to the algorithm in Section 3.4.3. 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 in the computation step of the current macro-step and for every edge e out of v, node m starts c-pebbles $[e,t]$ along on their way to nodes of H that require $[e,t]$ from m.

3. ROUTING STEP: Move every c-pebble $[e,t]$, which has not reached its destination and is currently at node m one step closer to its final destination.

Note that in the communication step, if the s-pebble, $\langle v,t\rangle$, is in the interior of the patch or the outerskirt, the c-pebbles are sent only to neighboring nodes of H. However, if it is on an i-ring it sends a copy of all its c-pebbles to its duplicate on the respective b-ring *in addition to* sending the appropriate c-pebbles to its neighboring nodes.

Lemma 4.2.2 *A macro-step takes a constant number of time steps.*

Proof: At each node of H, there are at most two 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 graph) to be sent. If the s-pebble is on the i-ring, it must send four 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. $\square$

Theorem 4.2.3 *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 4.2.1, we know that since the number of worst-case faults in H is less than $(\log N)/2$, the growth process terminates with non-overlapping finished boxes. The computation of G is mapped inside each of these finished boxes and each node of H performs the emulation algorithm as described earlier in this section. 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 4.2.2. The proof is similar to that of Theorem 3.4.4. Let ϕ be the function that maps an s-pebble, $\langle v,t\rangle$ to the node in H that contains it. 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 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. There are two cases. If $\langle v,t\rangle$ is an s-pebble in the border of the patch or the outerskirt, then it gets *all* its c-pebbles from the other s-pebble for node v which we will denote by $\langle v,t-1\rangle'$. If $\langle v,t\rangle$ is not on the border of the patch or the outerskirt then its gets its c-pebbles from $\langle u,t-1\rangle$ such that 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 $\langle v,t-1\rangle'$ in the first case and the dependency trees of $\langle v,t-1\rangle$ and $\langle u,t-1\rangle$, for all neighbors u of v in G in the second case.

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. We will now look at the dependency tree of $\langle v, T\rangle$. 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 $\langle e, T-1\rangle$. 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 which 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 will 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 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 and has a value of k. This corresponds to some long path taken by some c-pebble to go from $\phi(s_{i-1})$ in the i-ring to $\phi(s_i)$ in the b-ring of some finished box. The key observation is that since $\phi(s_{i-1})$ is a node in an i-ring, we can encounter no more long paths on the critical path until we reach an s-pebble embedded in the b-ring, i.e., the values of l_j, $i-1 \leq j \leq i-\Theta(k)$ are necessarily equal to 1. Thus $\sum_i l_i$ is at most $O(T+N)$. Thus the total number of macro-steps, T', is at most $O(T+N)$. □

It is possible to apply the construction described in this section recursively to show that we can tolerate $\log^{O(1)} N$ worst-case faults in an $N \times N$ mesh with constant slowdown.

Theorem 4.2.4 *For any constant c, an $N \times N$ mesh with $\log^c N$ worst-case faults can simulate the computation of a fault-free $N \times N$ mesh with constant slowdown.*

Proof: In Theorem 4.2.3, we showed that an $N \times N$ mesh can tolerate any number of faults less than $\log N/2$ with constant slowdown. We will now show how to tolerate any number faults less than $\log^2 N/16$ with a greater but still constant slowdown. Iterating this argument several times, we can show that the mesh can tolerate $\log^c N$ faults, for any constant c, with constant slowdown.

We will describe the emulation of G on H as consisting of two parts: a constant-slowdown emulation of G on a new network B and a constant-slowdown emulation of B on H. First, the mesh H is subdivided into $\sqrt{N} \times \sqrt{N}$ submeshes. Any submesh that has less than $(\log\sqrt{N})/2 = (\log N)/4$ faults is declared to be good. The rest of the submeshes are declared to be bad. Network B is an $N \times N$ mesh and has faulty nodes as defined below. Every node of B in a submesh that corresponds to a good submesh of H is fault-free. Every node of B in a submesh that corresponds to a bad submesh of H is faulty.

The emulation of B on H is described below. H needs to emulate only the non-faulty nodes of B. H will not use any node in a bad submesh. Since a good submesh has less than $\log N/4$ faults, we could use it to perform the computations of a $\Theta(\sqrt{N}) \times \Theta(\sqrt{N})$ mesh with constant slowdown, using the reconfiguration scheme described earlier in this section. Each such good submesh of H will emulate the corresponding submesh of B and some extra computation surrounding this submesh of width $\Theta(\sqrt{N})$. This extra computation covers the latency of the communication between adjacent submeshes in B. This leads to a constant-slowdown emulation.

The emulation of G on B is done as follows. We embed B in a $\sqrt{N} \times \sqrt{N}$ mesh B' by embedding each submesh of B into a single node of B'. A node in B' is defined to be faulty iff the corresponding submesh in B has faults. The number of submeshes of B with faults is less than $\log N/4$, since the total number of faults in H is less than $\log^2 N/16$. Therefore, B' has less than $\log N/4$ faults and hence we can use the growth process to grow a set of non-overlapping finished boxes in B'. This yields a set of non-overlapping finished boxes in B whose cores are regions embedded into the cores of the finished boxes of B'. Using these finished boxes we can perform an emulation of G on B with constant slowdown. □

4.3 Multi-scale emulation

In this section, we will show that an $N \times N$ mesh, H, with any set of $N^{1-\epsilon}$ faults (for some constant $0 \leq \epsilon < 1$) can simulate any computation of a fault-free $N \times N$ mesh, G, with constant slowdown. (Note that unlike the result proved in Section 3.4, we prove this result for a fixed $\epsilon < 1$.) The broad outline of our proof is similar to that in Section 4.2. First we grow a collection of finished boxes on the faulty mesh H (Section 4.3.1). Then we map the computation of G to H using the finished boxes created by the growth process (Section 4.3.2). Finally, we perform the emulation as a series of macro-steps (Section 4.3.3). The major difference between the emulation scheme in this section and that in Section 4.2 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*.

4.3.1 The growth process

In this section, we show how to grow boxes on the faulty mesh H. We have two types of boxes: a core, which is not capable of performing any portion of the emulation, and a finished box, which consists of a core surrounded by a skirt. An (α-β)-ensemble is a collection of finished boxes, each with a skirt of width α times the side-length of its core. Every finished box B in the ensemble has a unique level number. Unlike Section 4.2, the two finished boxes in an (α-β)-ensemble may intersect. The *intersecting region* of a finished box B in the ensemble is defined to be the region formed by nodes that lie both in B and in some other finished box with a smaller level number than B. The boxes in the 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. The perimeter of the intersecting region of every finished box B is at most β times the side-length of B.

Initially, every fault is enclosed in a unit-sized core. The growth process produces an (α-β)-ensemble of boxes. It proceeds in rounds until there are no more cores left. Each round produces either a new core or a new finished box. The round number in which a finished box is produced is also its level number. At the beginning of each round, a core of the smallest side-length (say k) is selected and a skirt of width αk is grown around it to form a box (call this box B). If $k > N/(2\alpha + 1)$ then the side-length of B will have to be bigger 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 perimeter of the intersecting region of B is more than β times the side-length of B, then 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 will cease to exist.

Create Step Otherwise the perimeter of the intersecting region of B is at most β times the side-length of B. 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 4.3.1 *The growth process produces an (α-β)-ensemble of finished boxes, provided that it does not fail.*

Proof: We must show that both 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. When a finished box B is created in the Create step, the perimeter of the intersecting region of B is at most β times its side-length. New finished boxes created in later rounds do not affect this intersecting region since they all have greater level numbers than B. Some of the finished boxes of level number 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 perimeter of intersecting region of B. Thus Property 2 will be true for all the finished boxes when the growth process terminates. $\square$

Theorem 4.3.2 *For any constants α and β, there exists a constant $\epsilon < 1$ such that for any set of $O(N^{1-\epsilon})$ faults in H the growth process grows an (α-β)-ensemble of finished boxes.*

Proof: We must show that for any value of α and β there is some ϵ 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 4.3.1, we can then infer the theorem.

Let $F(k)$ denote the minimum number of faults that a core of side-length k contains. We show by induction that $F(k) \geq Ak^{1-\epsilon}$, for some choice of constants A and $\epsilon < 1$. In order to satisfy the basis of the induction, we will choose A to be small enough such that $F(k) \geq Ak^{1-\epsilon}$, for all $k \leq \left\lceil \frac{1}{\alpha} \right\rceil$. 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 this new core. Using the fact that the minimum number of faults in the new core is at least the sum of the minimum number of faults in the cores used to form this new core and further using the inductive hypothesis, we have

$$\begin{aligned} F(k) &\geq F(x) + F(y_1) + \cdots + F(y_m) \\ &\geq Ax^{1-\epsilon} + Ay_1^{1-\epsilon} + \cdots + Ay_m^{1-\epsilon} \\ &\geq Ax^{1-\epsilon} + Ay^{1-\epsilon} \end{aligned} \tag{9}$$

where $y = \sum_{i=1}^{m} y_i$. The last inequality follows from the fact that $f(z) = Az^{1-\epsilon}$ is a convex function.

If a new core was formed, then it must have been formed in the expand step. Since the cores y_i belong to finished boxes created in earlier rounds, $x \geq y_i$, for all i. Therefore, the maximum side-length of the new core is no more than $(2\alpha+1)x + 2(1+\alpha)(\max_i y_i)$, which is no more than $(4\alpha+3)x$. This implies that $x \geq k/(4\alpha+3)$. Further, the perimeter of the intersecting region of the box formed with the core of side-length x is greater than $\beta(2\alpha+1)x$. This implies that the sum of side-lengths of the intersecting finished boxes, $(2\alpha+1)y$, should be at least $\beta(2\alpha+1)x/4$ which is at least $(\beta(2\alpha+1)/4)\cdot(k/(4\alpha+3))$. Putting these together with Equation 9, we obtain

$$F(k) \geq A\left(\frac{k}{4\alpha+3}\right)^{1-\epsilon} + A\left(\frac{\beta k}{4(4\alpha+3)}\right)^{1-\epsilon} \geq Ak^{1-\epsilon} \tag{10}$$

provided ϵ is chosen close enough to 1 such that $1 + (\beta/4)^{1-\epsilon} \geq (4\alpha+3)^{1-\epsilon}$. This completes the inductive proof that $F(k) \geq Ak^{1-\epsilon}$ for all values of k.

A core of side-length more than $N/(2\alpha+1)$ must necessarily contain more than $F(N/(2\alpha+1))$ faults. Therefore there must be more than $CN^{1-\epsilon}$ faults in the mesh, where constant $C = A/(2\alpha+1)^{1-\epsilon}$. Thus the growth process will never fail and will always produce an $(\alpha$-$\beta)$-ensemble of finished boxes for any set of $CN^{1-\epsilon} = O(N^{1-\epsilon})$ faults. $\square$

4.3.2 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 that we are about to describe requires that an $(\alpha$-$\beta)$-ensemble of finished boxes be grown in H, for some constants α and β such that β is less than some fixed fraction of α. We choose values of α and β that obey this constraint.

Next, we choose $\epsilon < 1$ such that for any set of $\Theta(N^{1-\epsilon})$ faults an (α-β)-ensemble of boxes can be grown using the growth process outlined in Section 4.3.1. By choosing the constants α, β, and ϵ in this manner, the computation of the guest G can be mapped to host H with any set of $\Theta(N^{1-\epsilon})$ faults.

We will use the pebbling terminology introduced in Section 4.2 to describe the mapping. A mapping of the computation of G onto 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 mapping is produced by a *mapping process* that progresses iteratively in rounds. Initially, the s-pebble of every 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. This initial mapping is not a valid one since it maps s-pebbles to faulty nodes of H. The mapping process selects a finished box at the beginning of each round in the decreasing order of their level numbers. The mapping process terminates when all finished boxes in the ensemble have been selected. In each round, the mapping process locally changes the mapping inside the selected finished box such that no s-pebbles are mapped to the core of that finished box. This is done in a way similar to that of Section 4.2 where s-pebbles were removed from the core by duplicating some of the s-pebbles and setting up constant-congestion paths between duplicated s-pebbles. 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. This will be the final mapping.

A *mesh-like* computation is said to be mapped onto some finished box B if each node in B has exactly one s-pebble mapped to it. Further, each s-pebble mapped to some 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 4.3.3 *Suppose B is a level l finished box that is selected at some round. At the beginning of this round, for every finished box B' of level k, $k \leq l$, either no computation is mapped to B', or a mesh-like computation is mapped to B'. Further, 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 onto 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 iterative mapping process produces upon termination a mapping that does not map computation or communication to faulty processors.

Theorem 4.3.4 *The iterative mapping process upon termination 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 4.3.3 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 (α-β)-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 4.3.3 that there must 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. □

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 box B of level l and side-length $(2\alpha+1)k$ be selected at some round of the mapping process. We assume that Invariant 4.3.3 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. Otherwise a mesh-like computation is mapped to the nodes of B at the beginning of this round, The computation within B is partitioned into two overlapping pieces, the outerskirt and the patch. The region consisting of nodes in B not more than distance $\alpha k/5$ from the boundary of B is called the *outer region*. The outerskirt is embedded somewhere within this region. Similarly a square box of side-length $3\alpha k/5$ is demarcated as the *patch region*. (See Figure 9). The patch will be embedded somewhere inside 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 9). 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' at a smaller level 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. Further, between the i-ring and the b-ring of the outer region there must be $\Theta(k)$ free rings. A similar 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 $\Theta(k)$.

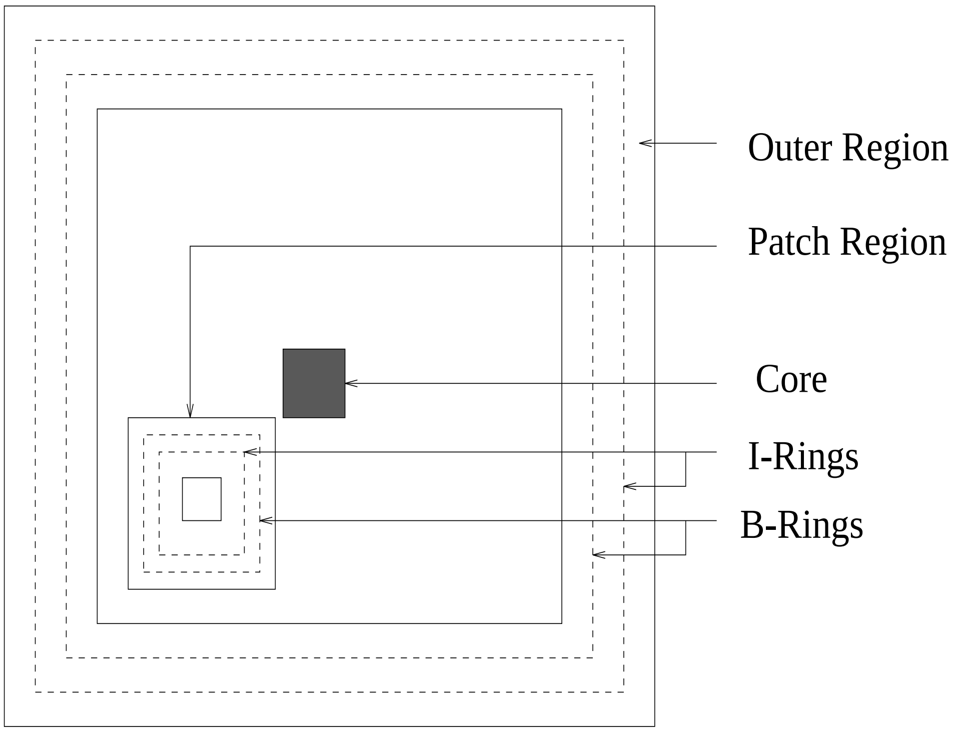

Figure 9: Layout of finished box B.

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 and must not pass through any finished boxes B' at a smaller level than B.

The procedure for finding rings with these properties is outlined in Section 4.5.

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. Unlike Section 4.2, the size of the outerskirt and patch will depend on the choice of the of the i- and b-rings. The computation mapped between the b-ring of the outer region and the boundary 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 4.3.3 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 boundary 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' at a smaller level than B which intersects this square region receives a mesh-like computation or receives no computation at all. The columns and rows of this square region that intersect such a finished box B' will be called a *bad column* or a *bad row*. The rest of the rows and columns are said to be *good rows* and *good columns* respectively. Since the total perimeter of such boxes B' is at most $\beta(2\alpha+1)k$, for a small enough value of β, the majority of the 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 bigger 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 should not pass through any finished boxes at levels lower than B. These paths can be set up since the i-ring and b-ring satisfy the second ring property mentioned earlier in this section. As in Section 4.2, the s-pebble on the b-ring receives c-pebbles from its duplicate on the i-ring using these paths.

We now show that Invariant 4.3.3 inductively holds at the end of the round in which B was selected.

Theorem 4.3.5 *Invariant 4.3.3 is true after the computation within B has been remapped.*

Proof: We must show that each finished box B' at a smaller level 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' at a smaller level 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 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 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. Further, the communication paths to the i-ring of the outer region do not pass through B'. 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, it is easy to see that none of the newly formed communication paths pass through any of the finished boxes B'. □

4.3.3 The emulation

The emulation procedure is the same as that in Section 4.2. 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 in the computation step of the current macro-step and for every edge e out of v, node m starts c-pebbles $[e, t]$ along on their way to nodes of H that require $[e, t]$ from m.

3. ROUTING STEP: Move every c-pebble $[e, t]$, which has not reached its destination and is currently at node m one step closer to its final destination.

Lemma 4.3.6 *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 using induction on the rounds of the mapping process. There is exactly one s-pebble mapped to every node of H and no communication paths passing through any node at the beginning of the first round of the mapping process. 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. 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 4.3.3, there is 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 8 c-pebbles that need to be sent, there are at most a constant number of c-pebbles created at each step. Thus the communication step takes only constant time. There can be only a constant number of c-pebbles in transit residing at any node at any time step. The reason is that there are only a constant number of paths passing through every node. Furthermore, 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. □

Theorem 4.3.7 *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 $O(N^{1-\epsilon})$ worst-case faults (for some constant $\epsilon < 1$) in time $O(T+N)$.*

Proof: We show that we require only $O(T+N)$ macro-steps for a T-step computation of G. The final result then follows from Lemma 4.3.6.

Let $B_1, B_2, \cdots, B_m$ be the finished boxes in the descending order of their level 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 B_i. The final mapping generated by the mapping process is ϕ_m. Let $s_T, s_{T-1}, \cdots, s_0$ be a sequence of s-pebbles, where $s_i = \langle v_i, t_i \rangle$. This sequence is called a T-sequence if for all i, $t_{i+1} = t_i + 1$ and v_i and v_{i+1} are either the same node of G or neighbors in G and s_i sends a c-pebble to s_{i+1}. For a given mapping ϕ_i and a T-sequence $s_T, \cdots, s_0$, $l_{i,j}$ is 1 if nodes $\phi_i(s_j)$ and $\phi_i(s_{j-1})$ are the same node or neighboring nodes in H. Otherwise, $l_{i,j}$ is the length of the path by which a c-pebble generated by s_{j-1} is sent to s_j. If $l_{i,j} > 1$ the path that corresponds to it is referred to as a *long path*.

We show that for any T-sequence $s_T, s_{T-1}, \cdots, s_0$, $\sum_{0<j\leq T} l_{m,j}$ is $O(T+N)$. We will define a series of weights $w_{i,j}$, $0 \leq i \leq m$ and $0 < j \leq T$ and c_i, d_i, $0 \leq i \leq m$. The weights are chosen such that for any value i the following two properties are satisfied.

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

The weights $w_{i,j}$ are chosen to balance all long paths (except perhaps one long path per finished box) inside the finished boxes B_h, $h \leq i$. The weights c_i and d_i are chosen to balance a long path in B_i that has not been balanced by the weights $w_{i,j}$.

Initially, we will define $w_{0,j} = l_{0,j}$, for all values of j and $c_0 = d_0 = 0$. 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. Clearly, $\sum w_{0,j} + c_0 + d_0 = \sum l_{0,j}$. Further, for all values of j, $w_{0,j}$ can be bounded from above by a fixed constant.

Inductively assume that we have determined the weights $w_{i-1,j}$, $0 < j \leq T$ and c_j, d_j, $0 \leq j \leq i-1$, such that the two properties mentioned above are satisfied. We choose $w_{i,j}$, $0 < j \leq T$, c_i, and d_i as follows. 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 4.3.3, ϕ_{i-1} maps a grid-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 $\phi_{i-1}(s_j)$ 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_j outside B_i but adjacent to its border and s_{j-1} on 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 s_{j-1}.)

We now look at s-pebbles mapped inside B_i by ϕ_{i-1}. The new mapping ϕ_i introduces long paths for s-pebbles s_j such that $\phi_{i-1}(s_j)$ lies on a b-ring of B_i. For all such s-pebbles s_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_j, $l_{i,j} = l_{i-1,j}$. We determine the weights $w_{i,j}$ for each s-pebble s_j such that $\phi_{i-1}(s_j)$ is in B_i as follows. We will consider every maximal subsequence, $s_{h+p}, s_{h+p-1}, \cdots, s_h$, of the T-sequence such that $\phi_{i-1}(s_{h+q})$ is in B_i, for $0 \le q \le p$. 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 long paths and for every $0 \le q \le p$, $l_{i,h+q} = l_{i-1,h+q} = 1$. Therefore we will define $w_{i,h+q} = w_{i-1,h+q}$ for every $0 \le q \le p$, and set $c_i = d_i = 0$. If $I \ne 0$, let L be $\sum_{q\in I}(l_{i,h+q} - l_{i-1,h+q})$, which equals the net increase in the values of $l_{i,q}$ in the subsequence.

If $|I| = 1$, there is exactly one long path. Note that this can happen only if either $\phi_i(s_0)$ or $\phi_i(s_T)$ is in box B_i. This is so because a maximal subsequence of a T-sequence that has neither $\phi_i(s_0)$ nor $\phi_i(s_T)$ in B_i must necessarily enter and leave the box B_i. Therefore such a subsequence must neccessarily use long paths an even number of times. If $\phi_i(s_0)$ is in B_i make $c_i = L$, where $L = \Theta(k_i)$. Otherwise set $c_i = 0$. Similarly if $\phi_i(s_T)$ is in B_i make $d_i = L$. Otherwise set $d_i = 0$. For every $0 \le q \le p$, set $w_{i,h+q} = w_{i-1,h+q}$.

If $|I| > 1$, let J be the set of integers q such that $\phi_i(s_{h+q})$ 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 long path in B_i is $\Theta(k_i)$ in length. This increase must be distributed evenly among the weights of the s-pebbles $s_{h+q}, q \in J$. Thus for all $q \in J$, $w_{i,h+q} = w_{i-1,h+q} + L/|J|$. For any two s-pebbles s_{h+q_1} and s_{h+q_2} such that $q_1, q_2 \in I$, the subsequence $s_{h+q_1}, \cdots, s_{h+q_2}$ contains at least $\Theta(k_i)$ s-pebbles $s_{h+q'}$, such that $q' \in J$. This is so because the 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 $\sum_j w_{i,j} + \sum_j(c_j + d_j) \ge \sum_j l_{i,j}$. Further, for all i and j, $w_{i,j}$ is at most a constant. This is so because if the weight of some s_j increases at the i^{th} iteration, i.e., $w_{i,j} > w_{i-1,j}$, then it will never increase again since $\phi_i(s_j)$ 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 at a lower level 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. Thus, we have a series of weights $w_{i,j}$, $0 \le i \le m$ and $0 < j \le T$ and c_i, d_i, $0 \le i \le m$ that satisfy the two properties mentioned previously.

We bound $\sum_j l_{m,j}$ by bounding $\sum_j w_{m,j}$ and $\sum_{i\le m}(c_i + d_i)$. 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 $O(N)$. Observe that since the core of B_i has at least $\Theta(k_i^{1-\epsilon})$ faults and since there are $\Theta(N^{1-\epsilon})$ faults in the mesh,

$$\sum_i \Theta(k_i^{1-\epsilon}) \leq \Theta(N^{1-\epsilon})$$

The maximum value of $\sum_i k_i$ with the above constraint is 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,

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

Let the emulation of T steps of G take T' macro-steps. Furthermore, let $\langle v, T \rangle$ be an s-pebble that was updated in the last macro-step. As in the proof of Theorem 4.2.3, we can define the dependency tree of $\langle v, T \rangle$ and choose the critical path, $s_T, s_{T-1}, \cdots, s_0$, of tree nodes from the root to the leaves of the tree. The critical path is a T-sequence. As in Theorem 4.2.3, we can show that $T' = \sum_{0<j\leq T} l_{m,j}$. Thus T' is $O(T+N)$. □

4.3.4 Random Faults

In Section 3.5, we used the worst-case fault result for a butterfly to derive a method for tolerating random faults on the butterfly. In a similar manner, we can use the worst-case faults result of Theorem 4.3.7 and derive the following result for the mesh.

Theorem 4.3.8 *For any fixed $\gamma > 0$, with probability at least $1 - 1/2^{N^{2-\gamma}}$, an $N \times N$ mesh in which each node fails independently with some constant probability $p > 0$ can emulate a fault-free $N \times N$ mesh with $2^{O(\log^* N)}$ slowdown.*

Proof: The proof is similar to the proof of Theorem 3.5.7 in Section 3.5. □

Though $2^{O(\log^* N)}$ is a very slowly growing function of N, it is not a constant. A stronger result is given in the following conjecture.

Conjecture 4.3.9 *An $N \times N$ mesh in which each node fails independently with some constant probability $p > 0$ can emulate a fault-free $N \times N$ mesh with constant slowdown, with high probability.*

We believe that we can prove this conjecture, using a construction along the lines of the construction for the worst-case faults. In its details, this latter construction is considerably more intricate. Resolving this conjecture will be of great interest, since then the mesh will become the first bounded-degree network known to tolerate constant-probability failures with constant slowdown.

4.4 A limit on the fault-tolerance of linear arrays

Unlike two- or higher-dimensional arrays, the one-dimensional linear array is not very tolerant of faults. It is easy to see that a linear array cannot tolerate more than a constant number of worst-case faults. This is because by placing $f(N)$ faults, an N-node linear array can be split into disjoint pieces of size $N/f(N)$, for any function $f(N)$ that can grow arbitrarily slowly with N. Emulating the entire linear array on one of these pieces entails a slowdown of at least $f(N)$. However, if we are to assume a weaker model of faults where faulty nodes cannot compute but can let communication pass through them, we can tolerate $\log^{O(1)} N$ worst-case faults with constant slowdown using the scheme of Section 4.2 in one dimension. A natural question is if there is a scheme to tolerate more than $\log^{O(1)} N$ worst-case faults in a linear array in this weaker fault model. In this section, we prove that this is not the case. 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. In this section, and this section only, we assume that faulty nodes are allowed to communicate with their neighbors but cannot compute.

In a *static* emulation, a *redundant* guest graph $G' = (V', E')$ is embedded in the host H. The redundant graph is defined as follows. For every node v in the guest graph $G = (V, E)$, 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 graph 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 graph 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 simulates T steps of the guest graph'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. Node a can create an s-pebble $\langle v', t\rangle$, only if it 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 $\langle v', t\rangle$, 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 host node a can create a single s-pebble (and the corresponding c-pebbles) and can send and receive one c-pebble on each of its edges. 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.

4.4.1 Bounding the load, congestion, and dilation

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 4.4.1 *Suppose that for any T, 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 simulate 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 4.4.2 *Suppose that for any T, 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 4.4.3 *Suppose that for any T, 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 simulates 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 simulates v'_1 to the node that simulates 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. □

4.4.2 Bounding the number of faults

Theorem 4.4.4 *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 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 graph must cross. Because the slowdown is s, and at most $N/g(s)$ host nodes lie between any pair of blocks, for $g(s) > s$, its 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 4.4.1.) 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 graph G' that crosses B. Let u be a node in the guest graph G such that every instance of u in G' on the right-hand 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 graph such that every instance of u on the right hand 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 there is always such a node u since it is not possible for the host to emulate the entire guest on the right side of B. 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 right input edge into v' back to some other node w' in G'. Node w' may be on either side of B. If it's on the left, follow the right input edge into w'. If it's on the right, follow the left input edge. Repeat this process until some node in G' is visited twice (i.e., a cycle C is formed.)

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 graph, where $l > f(s)/s$, and these nodes are simulated within distance $sl/2$ of B in the host. If the slowdown of the emulation is s, then by Lemma 4.4.3 the dilation of any cycle is at most s times the number of redundant graph nodes on the cycle. (The dilation of a cycle or path is equal to the sum of the delays of the edges on the cycle or path.) Let us define a *segment* to be a set of nodes visited by C between crossings of block B. 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 into the nodes 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 $f(s)/s$ nodes. Since a short segment has dilation at least $f(s)$, it's ratio of dilation to length (number of nodes) is more than s. Since the average ratio is at most s, there must be some long segment whose ratio of dilation to length is at most s. Thus, there must be some set of $l \geq f(s)/s$ nodes emulated within distance sl of B. 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 graph. We will call the sl 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 graph, their instances aren't 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 graph 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 inputs from outside the emulation region. However, since the embedding has congestion at most s (by Lemma 4.4.2), 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 graph 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 sl in which at least $l/4s$ nodes of the guest are emulated, for some $l \geq f(s)/s$, and some other edges of G' cause congestion 1 in the emulation region. In order to recurse on sets of of $l/4s$ guest nodes, where $l \geq f(s)/s$, we need $f(s) > 4s^2$.

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 $g(s)f(s)$ 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 $2^k < 2sl$. The blocks of faults in this pattern are spaced at a distance of $2^k/g(s)$, which is at most $2sl/g(s)$. 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 4.4.1 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 on one side of one of the blocks of P, there is an emulation region of size sl' in which at least $l'/4s$ nodes are emulated. 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 4.4.2. 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 $2^{2s+1}g(s)^{2s+1}(\log N)^{2s}f(s) = h(s)\log^{2s} N$,

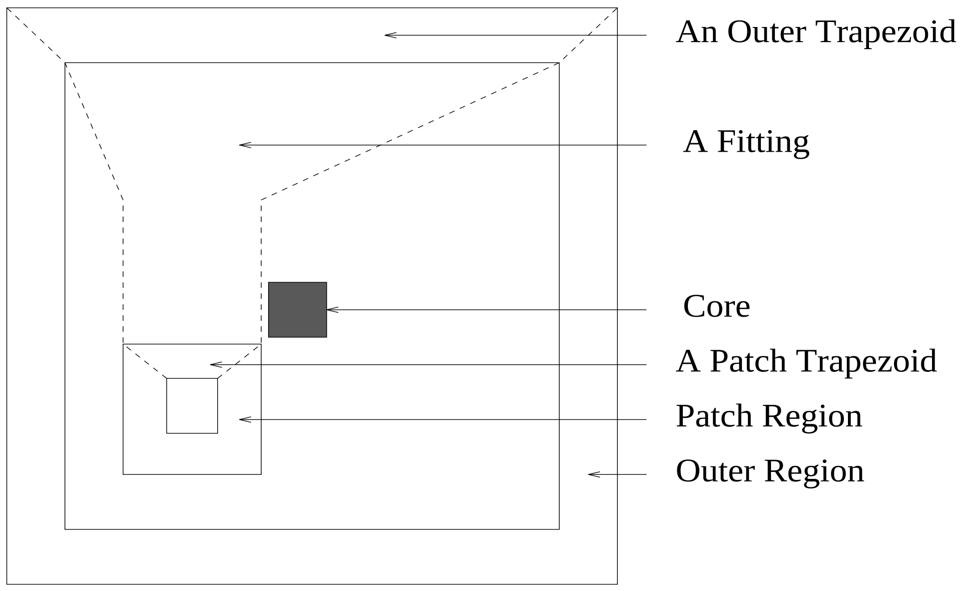

Figure 10: Finding i- and b-rings.

where $h(s) = 2^{2s+1} g(s)^{2s+1} f(s)$, $g(s) > 8s^3$ and $f(s) > 4s^2$. □

4.5 Finding the i- and b-rings

In this section, we will show how to find i- and b-rings in a finished box B of side-length $(2\alpha + 1)k$ satisfying the two ring properties outlined in Section 4.3.2. The ring properties are as follows.

1. The i-ring and the b-ring of the outer region must be free rings. Further, between the i-ring and the b-ring of the outer region there must be $\Theta(k)$ free rings. A similar 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 and must not pass through any finished boxes B' at a smaller level than B.

Recollect 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 10). The i-ring of the patch is required 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 (See Figure 10). Similarly, the outer region is also partitioned into four trapezoidal regions by joining each corner of the box B to the appropriate 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 10). 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 10). The fitting is either a trapezoidal region or a rectangular region adjoining a trapezoidal region (The patch region is positioned in such a way 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 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, 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 the patch grid respectively.

Embedding the outer trapezoid or the patch trapezoid into the 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.

We can use the above technique to embed the fitting as well. The main difference is that since the fitting may have more than $\alpha k/5$ columns, we would have to embed a constant number of columns of the fitting to one column of the fitting grid. 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 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. It must be noted 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, patch grid, and the entire rectangular grid R itself to be symmetric about the column in the center 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 the grid. 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 will define a square box in R to be *flowless* as follows.

Definition 4.5.1 *A square box F of side-length q is said to be flowless iff 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 4.5.2 *For a small enough value of β, a majority of the columns in the outer grid, fitting grid and patch grid will be 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.

We will now determine the maximum credit some node s on the perimeter of an obstacle receives. 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 the number of columns that are not live. The perimeter of the obstacles is at most some constant c (independent of β) times the perimeter of the intersecting region of B, i.e., at most $c\beta(2\alpha+1)k$. Thus the total number of credits in the nodes on the perimeters of the obstacles is equal to f and is at most $8c\beta(2\alpha+1)k$. By making β small enough, we can make this quantity less than $\alpha k/10$. It now follows that for this choice of β the majority of the columns in the outer, fitting and patch grids (each grid has $\alpha k/5$ columns) are live columns. $\square$

4.5.1 Permuting grids

We define a *permuting grid* as follows. An $l \times m$ rectangular grid with obstacles 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 4.5.3 *The 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 $l = O(m)$.*

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 four nodes. A path from node v in the left side to $\phi(v)$ 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 $\phi(v)$. In the last stage, the path goes along this horizontal path to the destination node $\phi(v)$. 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. The reason is that at most $4l/m$ nodes use a particular vertical path as its middle stage. Hence the net congestion is a constant. □

The live columns in the outer and patch grids are the set of columns from which we choose the i- and b-columns. Recollect that live columns do not pass through flowless boxes. We choose β as small as is required by Theorem 4.5.2 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 4.5.4 *The grids O, F, and P are permuting grids.*

Proof: First we show that O is a permuting grid. O is a $\alpha k/5 \times m$ grid, for some $m \leq \alpha k/5$. It has at least $\alpha k/10 \geq m/4$ live columns which can serve as the vertical paths in the grid. Now we grow constant congestion paths that do not hit obstacles from every node on the left side of O (i-column) to the corresponding node on the right side of O (b-column). 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 sizes $2^i, 0 \leq i \leq \log_2(\alpha k/5)$ (For simplicity, we assume that $\alpha k/5$ is a power of 2). The left side of every box consists of a set of adjacent nodes in the i-column. All the boxes 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 i^{th} iteration, we assume that each box of size 2^{i-1} has paths of congestion 8 and maximum length $4 \cdot 2^{i-1}$ originating from every node on its left side and ending at some node on its right side. We show how to construct paths of congestion 8 and 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 11). Let D denote the rectangular box formed by the the first half of the columns of the big box. D encloses both 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 the latter set of 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 11.) 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 i^{th} such group we assign the i^{th} 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 11). 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 upwards 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 downwards 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

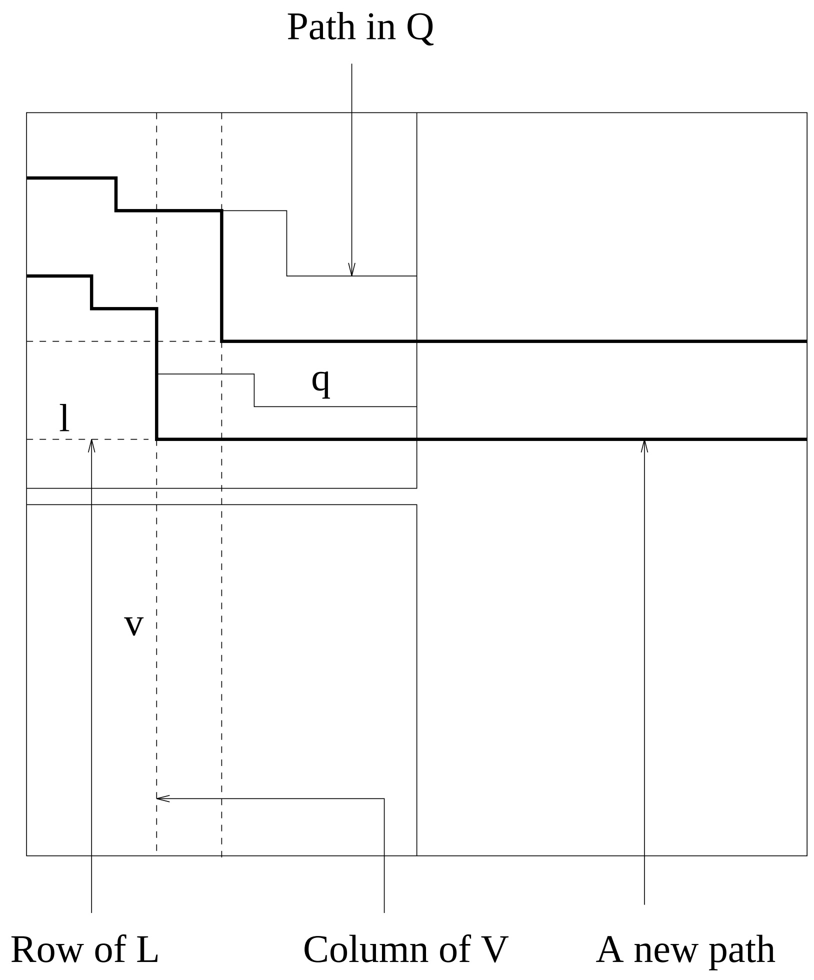

Figure 11: A new path constructed from paths q, v, and l.

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 bigger 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 4.5.5 *The b- and i-columns chosen in this manner 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 4.5.3 and Theorem 4.5.4. 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 4.5.3, 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. □

Chapter 5

Algorithm-Based Fault Tolerance

5.1 Introduction

Transient faults in a parallel computer can cause errors to appear in the data that it computes. Algorithm-Based Fault Tolerance (ABFT) was introduced as a concurrent error detection (CED) technique to detect and locate errors in matrix computations [HA84]. In ABFT, checking the correctness of the computation is carried out concurrently with the computation itself. In the case of matrix computations simple checksum computations are typically used to do the checking. There have been many applications of this technique to a variety of problems including Fast Fourier Transforms [CM88a, JA88, THC90], sorting [CM88b], and signal processing applications like matrix multiplication, matrix inversion, LU decomposition, QR decomposition, FIR filtering etc. [CA86, HA84, JA86, L85, LP88, RB90, VJ90]. It has also been applied to various architectures such as the linear array [A87, JA86], the mesh [HA84], and the hypercube [B88]. ABFT is a very attractive method for concurrent error detection and fault location due to its low hardware and time overhead. Many methods for analyzing ABFT systems also exist [BA86a, GRR90, LP86, NA88, RR88, VJ89].

In [BA86a], a simple graph-theoretic model for ABFT schemes was proposed. This model can be used for synthesizing ABFT systems as well as for analyzing the fault detectability and locatability properties of existing systems. The model is described below. An ABFT system is represented as a tripartite graph called the *PDC* graph whose vertex set is $P \cup D \cup C$ and whose edge set is $E_1 \cup E_2$, where P, D and C are the sets of processors, data, and checks, respectively, and E_1 and E_2 are the edges between P and D and between D and C, respectively. The bipartite graph with vertex set $P \cup D$ and edge set E_1 is called the *PD* graph. The bipartite graph with the vertex set $D \cup C$ and edge set E_2 is called the *DC* graph. An edge $(u, v) \in E_1$ implies that processor u affects the value of data element v in the computation, i.e., if processor u fails, v *could* have an error. An edge $(v, z) \in E_2$ implies that check z checks data element v. The set of data elements affected by a processor $u \in P$ is said to be its *data set.* As is traditionally done, we assume that a faulty processor results in an error in at least one of the data elements in its data set. The set of data

This chapter describes joint research with Niraj Jha [SJ91, SJ].

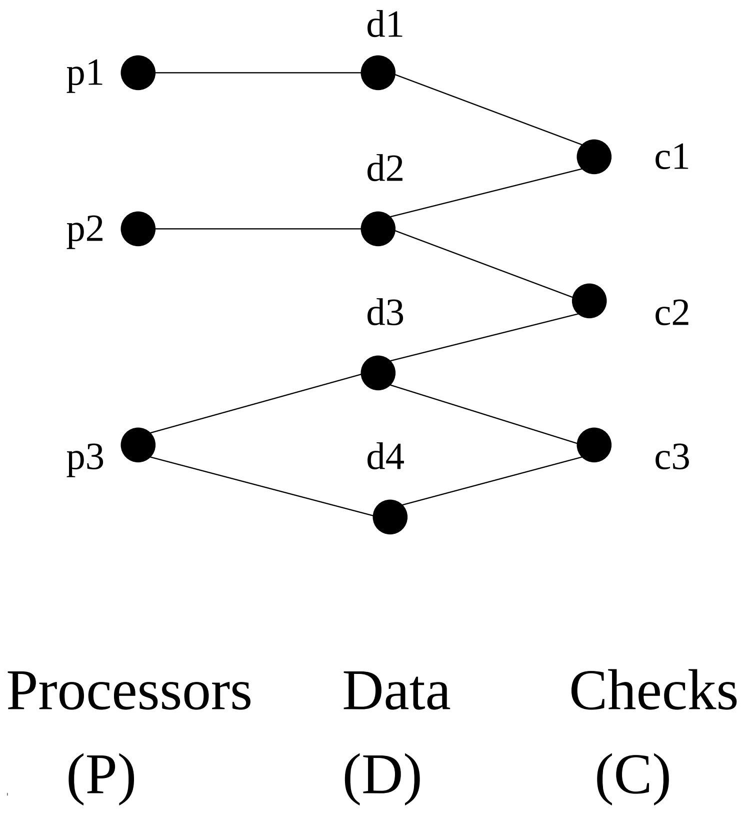

Figure 12: 2-fault detecting 1-fault locating PDC graph.

elements checked by a check $z \in C$ is said to be its *error set.* The checks are of the simplest kind. A check operates on a non-empty subset of the data and will detect exactly one error in its error set. More formally, a check z outputs a binary value of 0 or 1. The output of check z is 0 when all of the data elements in its error set are error-free. The output of check z is 1 when *exactly* one of the data elements is in error. If there is more than one error in the error set of the check, its output value is arbitrary and hence the check is undependable.

An example of a 2-fault detecting 1-fault locating PDC graph is given in Figure 5.1. If any two processors are faulty, at least one check outputs a 1. Therefore, the system is 2-fault detecting. Also if exactly one processor is faulty, the outputs at the checks uniquely identify the processor. If only $c1$ is 1 the faulty processor is $p1$. If $c1$ and $c2$ are 1 the faulty processor must be $p2$. In all other cases in which some check outputs a 1, the faulty processor is $p3$. Therefore the system is 1-fault locating.

The graph model itself makes no assumption about the implementational details of the ABFT system, for example, the architecture of the machine on which the ABFT system is run or the exact manner in which a check is implemented. It insists however that individual checks be extremely simple computations. The checks themselves are computed by some dedicated hardware *not represented in the PDC graph.* A simple implementation of a check is to use an (unweighted) checksum of all the data elements in the error set. The hardware computing a check will independently compute the value of the assigned output checksum from the system inputs and then compare it with the checksum obtained from the data elements in its error set. If these two values tally, the check outputs a 0, otherwise the check outputs a 1.

One of the main goals of research in ABFT is to design efficient systems which are t-fault locatable (or detectable), i.e., assuming that not more than t processors can fail in a computation, one would like to locate exactly which, if any, of the processors failed (or simply detect that there has been a failure). There are many different approaches to designing t-fault locatable (or detectable) ABFT systems. Two popular approaches are the synthesis for fault tolerance approach [VJ90] and the design for fault-tolerance approach [BA86a, GRR90, NA89, NA90, VJ91]. Both of these approaches require a systematic procedure to design DC graphs which can detect or locate a specified number of errors. It is also desirable that the number of checks used in the DC graph design be as small as possible. The focus of our work is to devise a method for generating such DC graphs.

We now formally define what it means for a DC graph to detect or locate s errors.

Definition 5.1.1 *A DC graph is s-error detectable if every possible non-empty set of errors in the data of cardinality at most s makes at least one of the checks output a* 1.

Definition 5.1.2 *A DC graph is s-error locatable if every possible set of errors in the data of cardinality at most s gives a different output pattern at the checks, i.e. no two distinct sets of errors of cardinalities at most s can give the same output pattern at the checks.*

5.1.1 Chapter outline

The outline for the rest of the chapter is as follows. In Section 5.2, we describe a versatile algorithm called RANDGEN that can be used to generate DC graphs with a variety of properties.

The minimum number of checks required for a DC graph with n data elements to be s-error detectable has been shown to be $\Omega(s \log n)$ [GRR90][1]. In Section 5.3, we show how to construct an s-error detectable DC graph with an asymptotically optimal number of checks.

There were no general methods known previously for designing s-error locatable graphs with a small number of checks. A lower bound for the minimum number of checks required for a DC graph to be s-error locating is $\Omega(s \log n)$ [BA86a]. In Section 5.4, we show that RANDGEN produces an s-error locatable DC graph with $O(s^2 \log n)$ checks. Since typically $s \ll n$, the number of checks is at most a small multiplicative factor from the optimal.

It must be noted that s-error locatability only ensures that no two distinct sets of error patterns of size s or less have the same output pattern at the checks. It does not provide us with an efficient algorithm to diagnose, i.e., actually locate the errors from the output pattern at the checks. No efficient algorithm for diagnosis is currently known for a general s-error locatable DC graph. In fact, the known algorithm is enumerative in its approach

[1]All logarithms in this chapter are to the base 2.

and tries to enumerate various possible sets of errors and check if the output pattern at the checks can be caused by them [VJ89] and could be time-consuming for even small values of s. Therefore, we must *design DC* graphs explicitly for easy diagnosis. In Section 5.5, we introduce a class of s-error locatable *DC* graphs which allow easy diagnosis and show how RANDGEN can generate them with only a constant factor overhead in the number of checks.

Uniform checks [VJ91] which are all identical and check the same number of data elements have been shown to simplify ABFT design. In Section 5.6, we show how RANDGEN can be modified to produce such uniform checks for s-error detection/location. Finally, in Section 5.7 we present concluding remarks.

5.2 An algorithm for generating DC graphs

In this section, we propose a simple and efficient algorithm called RANDGEN (for RANDom GENeration) for generating DC graphs. By varying the input parameters of RANDGEN one can synthesize, using only a small number of checks, *DC* graphs with a wide range of properties that researchers in ABFT have found to be useful and important in their designs. The construction algorithm RANDGEN is novel in that it is probabilistic, i.e. it makes random decisions during the course of the construction by perhaps using a random number generator. In order to construct a *DC* graph with n data elements, algorithm RANDGEN takes two arguments: the number of checks, c, that can be used and a probability p. It creates the *DC* graph by adding each edge (u, v), $u \in D$ and $v \in C$ with probability p, where $|C| = c$. It is easy to see that RANDGEN is simple to implement and very fast.

Theorem 5.2.1 *Algorithm RANDGEN runs in time $O(cn)$, where c is the number of checks in the graph and n the number of data elements.*

It must be mentioned that since RANDGEN is probabilistic, our results will show that with an overwhelmingly large probability the *DC* graphs produced will have the required properties. The proof techniques used to prove these results are techniques from the theory of random graphs [ES74, B85]. It must be mentioned that there are many random constructions known in graph theory to construct graphs with certain properties, for example, expander graphs. While our constructions are similar in spirit, our contribution is that random constructions can be a simple and useful tool to solve problems that arise in the design of ABFT systems and can give better results than previously known deterministic solutions to these problems.

5.3 Error detectability

We state the following lower bound from [GRR90] without proof.

Theorem 5.3.1 *The number of checks, c, for s-error detectability is $\Omega(s \log n)$.*

The main theorem of this section shows that RANDGEN produces (with high probability) an s-error detectable DC graph when parameter $c = 3.8s \log n$ and parameter $p = \frac{1}{s}$ are used. Note that the number of checks used is asymptotically optimal. Before we prove the theorem, we illustrate the algorithm RANDGEN for typical problem values. In [BA86b], the problem of encoding a matrix of dimension 1024×1024 and analyzing the reliability of various matrix multiplication algorithms is considered. We will use the encoding of 1024×1024 data values for s-error detection and location as a running example to illustrate our constructions.

Example 5.3.2 *Suppose we would like to construct a DC graph which can detect up to* 3 $(= s)$ *errors for a data set consisting of a matrix of dimension* 1024×1024, *i.e.,* 1048576 *data elements. We take* 228 *checks and with each check we do the following. We consider every data element and we include a data element in the error set of this check with a probability of* $\frac{1}{3}$ $(= \frac{1}{s})$. *When we are done with this process we are left with a DC graph which is* 3*-error detecting with a probability of at least* $1 - \frac{1}{1048575}$, *which is very close to 1. As a basis for comparison, notice that the traditional matrix row and column checksum method, which can detect up to* 3 *errors, requires* 2047 *checks.*

It should be pointed out that efficient methods have already been given in [GRR90] for generating s-error detectable DC graphs specifically for the particular cases of $s = 2$, 3 and 4. Then they give a special method for detecting up to 7 errors. However, as a comparison, for this example their method would require 570 checks for detecting 5, 6 or 7 errors, whereas our method would require 380, 456 and 532 checks, respectively. For $s > 7$ they have given a general construction method. For s ranging from 8 to 15 they would require 9120 checks whereas our method would require only 608, 684, 760, 836, 912, 988, 1064 and 1140 checks, respectively. As the value of s and/or n increases, our method performs relatively even better than the method in [GRR90]. One must note, however, that their method is deterministic whereas ours is probabilistic.

We now prove the main theorem of this section. Throughout this chapter, e represents the transcendental number 2.7182818...

Theorem 5.3.3 *The algorithm RANDGEN, using parameter* $c = 3.8s \log n$ *and* $p = \frac{1}{s}$, *produces an s-error detectable DC graph with probability at least* $1 - \frac{1}{n-1}$. *The time complexity of constructing this graph is* $O(sn \log n)$.

Proof: The algorithm RANDGEN clearly works for $s = 1$, since $p = 1$ and every check is connected to all data elements. Of course, one such check would suffice. So we will assume that $s > 1$. We need to show that the DC graph satisfies the conditions of Definition 5.1.1, i.e., every non-empty set $S \subset D, |S| \leq s$, has a check z such that it is connected to exactly one element of S. Let E_S represent the event that there exists no such check for some set S. The probability that the DC graph is not s-error detectable is simply the probability of $\cup_S E_S$, where S takes on the value of all nonempty subsets of D with cardinality not more than s. We will split this union of events into smaller unions as follows and bound each separately.

Let event $A_i, 1 \leq i \leq s$, be $\cup_R E_R$, where R takes all subsets of D of cardinality i. For any single set $S, |S| = i$, and a particular check z, the probability that z is not connected to exactly one element of S, i.e., that this check is "bad", is $1 - ip(1-p)^{i-1}$. We now choose $p = \frac{1}{s}$ which minimizes this expression for $i = s$. Observe that for this value of p

$$1 - ip(1-p)^{i-1} = 1 - \frac{i}{s}(1-\frac{1}{s})^{i-1} \leq 1 - \frac{i}{s}(1-\frac{1}{s})^{s-1} \leq 1 - \frac{i}{se} \tag{11}$$

The last inequality follows by observing that

$$(1 + \frac{1}{s-1})^{s-1} \leq e$$

From the independence in choosing the edges, the probability that all checks are "bad" for a single set S is $(1 - ip(1-p)^{i-1})^c$. We next bound the probability of event A_i.

$$\begin{aligned} Prob(A_i) &\leq \sum_{S,|S|=i} Prob(E_S) \leq n^i(1 - ip(1-p)^{i-1})^c \\ &\leq n^i e^{-ci\frac{1}{s}(1-\frac{1}{s})^{i-1}} \leq n^i \cdot \frac{1}{n^{2i}} = \frac{1}{n^i} \end{aligned} \tag{12}$$

using Equation 11 and choosing $c = 3.8s \log n \geq \frac{2e}{\log e} s \log n$.
Thus the probability of the DC graph being "bad", i.e. not satisfying the conditions of Definition 5.1.1, is simply

$$Prob(\cup_{1 \leq i \leq s} A_i) \leq \sum_{1 \leq i \leq s} Prob(A_i) \leq \frac{1}{n} + \frac{1}{n^2} + \cdots + \frac{1}{n^s} \leq \frac{1}{n-1}$$

Therefore the probability of a "good" DC graph, i.e. one which does satisfy the conditions of Definition 5.1.1, is at least $1 - \frac{1}{n-1}$. RANDGEN's time complexity follows from Theorem 5.2.1. □

For some applications, one may want to decrease even further the probability that the constructed DC graph is not s-error detectable at the cost of adding more checks. One can decrease this probability very quickly by the addition of some extra checks. In our example, we can add 114 more checks to make the total number of checks 342 and the probability of a bad DC graph goes down rapidly to $\frac{1}{1048576^2-1} \approx \frac{1}{10^{12}}$. We can decrease this again by adding more checks if need be.

Corollary 5.3.4 *The algorithm RANDGEN, using* $c = (3.8s + 1.9sk)\log n$ *checks and* $p = \frac{1}{s}$, *produces an* s*-error detectable DC graph with probability at least* $1 - \frac{1}{n^{k+1}-1}$.

Proof: Follows from the proof of the previous theorem by substituting the new value for c at the appropriate step. □

From the above discussions it is clear that there is a close relationship between the probability of getting a good DC graph and the number of checks c. After fixing this probability to a value that one will be satisfied with, one can do the computation backwards and find the corresponding value of c.

One possible criticism of the above approach is that the probability of getting a good DC graph cannot be made 1, although it can be made arbitrarily close to 1. However, one should remember that any fault tolerant design has an *inherent* chance of failure. A system which is assured to catch s faults/errors will fail in the unlikely (but still probable) event that more than s faults/errors occur. As long as the probability of getting a bad DC graph is small compared to the other reasons for failure, there should be no cause to worry. Even so, the degradation in this construction is "gradual". Even a bad DC graph, improbable as it may be, will still detect most sets of s or fewer errors.

If the designer still insists on having a guarantee that the DC graph obtained by RANDGEN is in fact good, then one can use the analysis procedures from [NA88] which, when given a DC graph, can determine if it is s-error detectable or not. In the extremely rare cases where the DC graph is found to be bad, one can use RANDGEN once again. Similar arguments also hold for the subsequent sections where one can check if the conditions that need to be satisfied by the DC graph are actually satisfied by it. However, this approach of verifying the "goodness" of a DC graph may, in general, be time-consuming.

5.4 Error locatability

In this section we consider the problem of error locatability. Let n be the total number of data elements, i.e., $|D|$, as before. The following simple lower bound was observed in [BA86a].

Theorem 5.4.1 *The number of checks, c, for s-error locatability is $\Omega(s \log n)$.*

Proof: Clearly, from Definition 5.1.2, there must be at least as many possible output patterns as there are distinct sets of errors of cardinalities at most s.

$$2^c \geq \sum_{0 \leq j \leq s} \binom{n}{j} = \Omega((\frac{n}{s})^s)$$

The theorem follows by taking logarithms on boths sides and observing that $s \ll n$. $\Box$

Trivially, suppose $s = 1$. There is a simple way of achieving the lower bound of Theorem 5.4.1 of $\lceil \log(n+1) \rceil$ checks. We observe that the total number of distinct nonempty subsets of $\lceil \log(n+1) \rceil$ checks ($= 2^{\lceil \log(n+1) \rceil} - 1$) is at least n. We simply connect each vertex of D, i.e., each data element, to a distinct subset of the checks. One way of doing this is as follows. Let the data elements be denoted by $d_1, d_2, \cdots, d_n$ and let $q = \lceil \log(n+1) \rceil$. For a data element d_i consider the q-bit binary vector which represents i. Then d_i would be connected to all those checks which correspond to the 1's in the binary vector. A similar scheme was used in [GRR90] for 2-error detection (not location). When d_i is in error, exactly the checks in the corresponding unique subset have 1's. Hence the DC graph is 1-error locatable. We should add that in addition to 1-error locatability such a DC graph

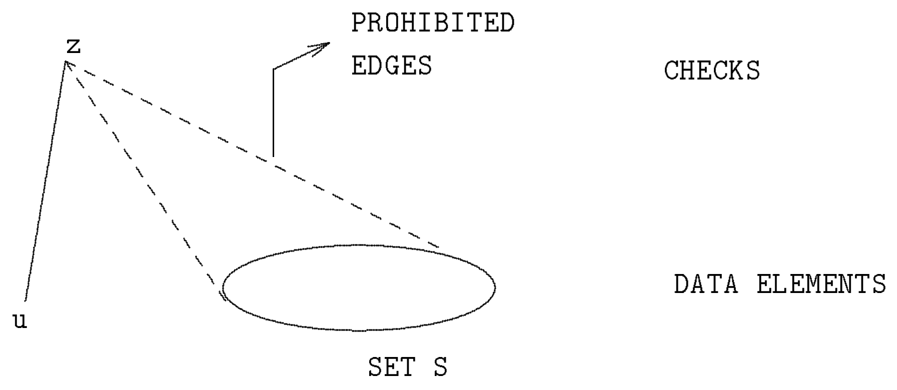

Figure 13: Sufficient conditions for s-error locatability.

also allows for easy diagnosis. The reason is that just by looking at the output pattern at the checks, we can immediately locate the data element in error. However, this construction does not extend in a natural way to $s \geq 2$.

The main result of this section is that RANDGEN with parameters $c = (7.6s^2 + 3.8s)\log n$ and $p = \frac{1}{2s}$ almost always produces an s-error locatable DC graph. The number of checks necessary for this algorithm is within a small constant factor from optimal since typically $s \ll n$. As before, we first illustrate the construction procedure by stepping through algorithm RANDGEN for our running example.

Example 5.4.2 *Suppose that we would like to design checks such that one can locate up to 3 (= s) errors in the data set which consists of a matrix of dimension* 1024×1024. *First take* 1596 *checks and with each check we do the following. We consider every data element and we include a data element in the error set of this check with a probability of* $\frac{1}{6}(= \frac{1}{2s})$. *At the end of this process we get a DC graph which is* 3*-error locating with a probability of at least* $1 - \frac{1}{1048576}$. *It was not previously known how to design s-error locatable DC graphs with a small number of checks for a general value of s. However, as a basis of comparison, it must be noted that one would require* 2047 *checks to even locate only* 1 *error using the traditional row and column checksum method. Our general method would require* 760 *checks to locate up to* 2 *errors and only* 228 *checks to locate* 1 *error. Actually, to locate only 1 error, we need not use this general method. From the method presented at the beginning of this section for this particular case, we need only* $\lceil \log(n+1) \rceil$ *= 21 checks.*

Before we prove our main theorem we need to state certain sufficient conditions for a DC graph to be s-error locatable.

Theorem 5.4.3 *For every* $S \subset D$ *such that* $|S| = 2s - 1$, *and for every* $u \in D, u \notin S$, *suppose that there exists a check z which is connected to u but not to any member of S. Then the DC graph is s-error locatable.*

Proof: The conditions in the theorem are pictorially depicted in Figure 1. Note that the condition also holds for every set S of cardinality less than $2s-1$, since the condition can be applied to an arbitrary superset of S of cardinality exactly $2s-1$. Consider any two distinct subsets of D, namely R and T, such that their cardinalities are not more than s. Take any element $v \in R \oplus T$, where $\oplus$ represents the symmetric difference[2]. Without loss of generality, let $v \in R$. Now by the conditions, there exists a check which is connected to v but not to any element of $R \cup T - \{v\}$, since the cardinality of this set is $\leq 2s-1$. This directly implies that this check outputs 1 when R is the set of errors and 0 when T is the set of errors, i.e., these sets have different output patterns at the checks. $\Box$

Theorem 5.4.4 *The algorithm RANDGEN, using $c = (7.6s^2 + 3.8s)\log n$ checks and $p = \frac{1}{2s}$, produces an s-error locatable DC graph with probability at least $1 - \frac{1}{n}$. The time complexity for constructing this graph is $O(s^2 n \log n)$.*

Proof: We will show that the DC graph satisfies the sufficient conditions of Theorem 5.4.3 with high probability.

Given a particular $u \in D$ and $S \subset D$, $u \notin S, |S| = 2s-1$, let $E_{u,S}$ be the event that no check in C satisfies the conditions of Theorem 5.4.3. The probability that a particular check does not satisfy the conditions is $1 - p(1-p)^{2s-1}$. We now choose p so as to minimize this probability. It can be easily checked that this is minimum for $p = \frac{1}{2s}$. For this value of p,

$$1 - p(1-p)^{2s-1} = 1 - \frac{1}{2s}(1 - \frac{1}{2s})^{2s-1} \leq 1 - \frac{1}{2se} \tag{13}$$

The last inequality follows from the fact that

$$(1 + \frac{1}{2s-1})^{2s-1} \leq e$$

As the edges for each check are chosen independently, the probability that no check satisfies the conditions of Theorem 5.4.3, i.e. $Prob(E_{u,S})$, is clearly $(1 - p(1-p)^{2s-1})^c$. Observe that the probability that the DC graph does not satisfy the sufficient conditions is simply the probability that at least one of the events $E_{u,S}$ occurs, for some u and S, i.e., it equals $Prob(\cup_{u,S} E_{u,S})$ where S takes all subsets of D of cardinality $2s-1$ and u takes on as values all elements in $D - S$. We bound this probability as follows,

$$\begin{aligned} Prob(\cup_{u,S} E_{u,S}) &\leq \sum_{u,S} Prob(E_{u,S}) \leq n^{2s}(1 - p(1-p)^{2s-1})^c \\ &\leq n^{2s} e^{-c\frac{1}{2s}(1-\frac{1}{2s})^{2s-1}} \leq n^{2s} \cdot \frac{1}{n^{2s+1}} = \frac{1}{n} \end{aligned}$$

by using Equation 13 and choosing $c = (7.6s^2 + 3.8s)\log n \geq (\frac{4e}{\log e}s^2 + \frac{2e}{\log e}s)\log n$. Thus the probability that the DC graph is s-error locatable is at least $1 - \frac{1}{n}$. The time taken by

[2] $A \oplus B = (A - B) \cup (B - A)$

RANDGEN follows from Theorem 5.2.1. □

As before, the probability that we get a "bad" *DC* graph, i.e., one which is not s-error locatable, can be reduced very rapidly by adding some extra checks. By adding $3.8sk\log n$ extra checks we can decrease this probability to $\frac{1}{n^{k+1}}$. To illustrate this through the previous example, we can add just 228 more checks to make a total of 1824 checks, then our chance of producing a "bad" *DC* graph, i.e., a graph which is not 3-error locating, falls from $\frac{1}{1048576}$ to $\frac{1}{1048576^2} \approx \frac{1}{10^{12}}$. We can continue to do this and our probability of producing a bad *DC* graph goes down extremely rapidly.

Corollary 5.4.5 *The RANDGEN algorithm, using* $c = (7.6s^2 + 3.8sk + 3.8s)\log n$ *checks and* $p = \frac{1}{2s}$*, produces an* s*-error locatable DC graph with probability at least* $1 - \frac{1}{n^{k+1}}$.

DC graphs with a combination of properties

Some researchers [NA89] have used *DC* graphs with a combination of detection and location properties for their design, i.e., *DC* graphs which are simultaneously s-error locatable and t-error detectable. One simple way to generate these graphs is, of course, to use RANDGEN twice: once for s-error locatability as shown in this section and once for t-error detectability as shown in the previous section. By putting together the checks we would have a *DC* graph which is both s-error locatable and t-error detectable with a total of $(7.6s^2+3.8s+3.8t)\log n$ checks. But in many cases this may not be necessary. Given specific values of s and t, one could choose p appropriately and calculate the minimum value of c needed to simultaneously satisfy the bounds for locatability in this section and the bounds for detectability in the previous section. This value of c may turn out to be smaller than what one would get by simply adding together the checks. But it is difficult to give a rule of thumb in general terms. We state below the case when $t \leq 2s$ in which we get detectability for free. This was first observed by Russel and Kime [RK75] in the different context of system-level diagnosis.

Lemma 5.4.6 *Any* s*-error locatable DC graph is also* $2s$*-error detectable.*

Proof: Consider any nonempty set of data elements S of cardinality at most $2s$. We need to show that some check is 1 if S is the set of data in error. One can always partition S into non-intersecting sets S_1 and S_2 such that the cardinalities of both the sets are less than or equal to s. Without loss of generality, let S_1 be non-empty. From the conditions of s-error locatability there must be a check, z say, that *must* be 1 when S_1 is the set of errors and *must* be 0 when S_2 is the set of errors. This means that check z is connected to exactly one element in S_1 and no element in S_2. Clearly, z must be 1 when S is the set of errors since it is connected to exactly one data element in S. Thus any set of at most $2s$ errors is detectable. □

From Lemma 5.4.6, we know that if $t \leq 2s$ it is sufficient to just design an s-error locatable graph using RANDGEN.

5.5 Designing *DC* graphs for easy diagnosis

We have so far dealt with the question of how to design *DC* graphs for s-error detectability and s-error locatability. Given that some data elements are in error, the checks take on binary values 0 or 1. Following convention, we will refer to this binary vector of check outputs as the *syndrome*. An s-error locatable *DC* graph assures us that no two distinct sets of errors in the data of cardinality less than or equal to s can give rise to the same syndrome. The problem of actually finding the set of data in error (or the set of processors that are faulty), given a particular syndrome, is called *diagnosis*. Note that an s-error locatable *DC* graph does not necessarily imply a simple and efficient method for diagnosis. It simply assures us that given enough time and/or hardware one can eventually diagnose (locate) the errors/faults.

A straightforward, but brute-force approach, for diagnosing any syndrome in any s-error locatable *DC* graph is to try all possible sets of errors of cardinality $\leq s$ and see which one is consistent with the given syndrome. Consistency of a set of errors with a syndrome is determined by checking that every check with a 0 has either no error or more than 1 error in its error set and every check with a 1 has one or more errors in its error set. This, of course, requires us to try $\sum_{1\leq i\leq s} \begin{pmatrix} n \\ i \end{pmatrix}$ different sets. For each set, checking the consistency of this set with the syndrome can be done in time proportional to the number of edges in the *DC* graph. This enumerative approach of trying all possible sets is too time-consuming and can be impractical even for moderate values of s. Although better results than this one are known [VJ89], it seems quite likely that no efficient non-enumerative algorithm to diagnose an arbitrary s-error locatable *DC* graph exists. This is our main motivation for designing *DC* graphs which are not only s-error locatable but *also allow easy diagnosis*. Therefore, we will propose a simple diagnosis algorithm and show how we can use RANDGEN to generate *DC* graphs that are not only s-error locatable but will also have the additional property that this simple diagnosis algorithm can be used to correctly locate errors in this graph.

We propose the following simple and efficient diagnosis algorithm called the *majority diagnosis algorithm*. Given a syndrome, for every data element $u \in D$ the algorithm decides if u is erroneous or error-free as follows. Let S represent the set of all checks that are connected to a data element u. If the majority (half or more) of the checks in S output a 1, u is declared to be erroneous. Otherwise u is declared to be error-free. This algorithm is very fast and runs in time linear to its input size, i.e. in time proportional to the number of vertices and edges in the *DC* graph. Besides, each data element is decided to be erroneous or not independently of the others and therefore the Majority Diagnosis algorithm can also be executed extremely efficiently in parallel. But this algorithm will not, of course, diagnose correctly for any general s-error locating *DC* graph. The challenge is to *synthesize s-error locating DC graphs with the added property that this majority algorithm can be used to correctly diagnose the errors.* Also, we would like to do it without sacrificing the results of Section 5.4, which are close to asymptotically optimal. We answer this question in the affirmative in Theorem 5.5.6 by showing that only a constant factor overhead is needed to accomplish this. First we define some concepts.

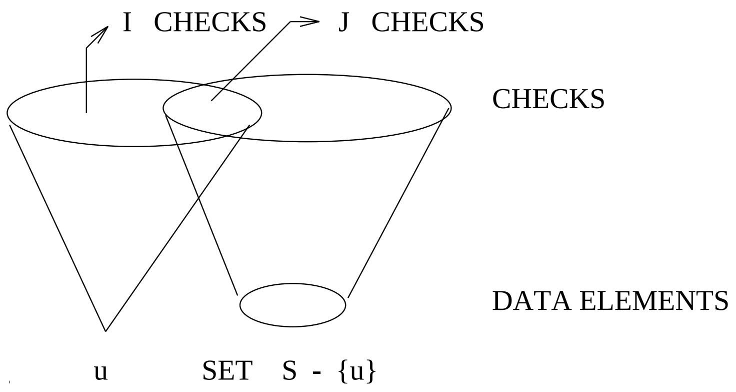

Figure 14: Necessary and sufficient conditions for s-majority diagnosability.

Definition 5.5.1 *A DC graph is said to be s-majority diagnosable iff the Majority Diagnosis algorithm can correctly locate any set of s or fewer errors from the syndrome.*

Lemma 5.5.2 *A DC graph is s-majority diagnosable iff for every $u \in D$ and $S \subseteq D$ such that $|S| = s$, the following is true: the cardinality of the set of checks connected to u but not to any data element in $S - \{u\}$ is greater than the cardinality of the set of checks which are simultaneously connected to u and some non-empty subset of $S - \{u\}$.*

Proof: Let I be the number of checks connected to the data element u and J be the number of checks which are simultaneously connected to u and some non-empty subset of $S - \{u\}$. Therefore $I - J$ checks are connected to u but not to any element in $S - \{u\}$. A pictorial representation of the conditions in the statement of the lemma is shown in Figure 14. We first prove the necessity of the conditions.
Only if: Assume that we have an s-majority diagnosable graph. For contradiction, assume that there is a set $S \subset D$ of cardinality s and a data element u such that the condition is not satisfied, i.e., $I \leq 2J$. Let all the data elements in $S - \{u\}$ be erroneous and the remaining data elements be error-free. Every check connected to only u and no element of $S - \{u\}$ necessarily outputs 0. There are $I - J$ such checks. The remaining checks that are connected to u *could* all output 1 since they are connected to at least one erroneous data element. There are J such checks. Since $I - J \leq J$ the majority diagnosis algorithm will diagnose u to be erroneous. This is a contradiction.
If: Now we prove that if the conditions are met then the DC graph is s-majority diagnosable. Suppose we are given a set of errors T. Without loss of generality, we can assume $|T| = s$. From the conditions in the statement of the lemma, we know that the number of checks connected to a data element u and not to any element of $T - \{u\}$ is greater than the number of checks simultaneously connected to u and some non-empty subset of $T - \{u\}$. The former set of checks always takes 1 or 0 depending on whether or not u is in the set of errors T. Thus the majority of the checks connected to u necessarily have values 1 or 0, respectively.
□

Example 5.5.3 *To illustrate the concepts we give an example of a* 1*-majority diagnosable DC graph. Consider a set of* 64 *data elements arranged in a* $4 \times 4 \times 4$ *cube. We associate a check with the data elements in the same row either in the* x, y *or* z *coordinate axis. So there are a total of* 48 *checks and each data element is connected to* 3 *checks, one in each of the three coordinate axes. Given any data element* u *and any other data element* v*, the number of checks connected to both* u *and* v *is at most* 1*, i.e., when* u *and* v *are in the same row along either* x, y *or the* z *axis. Since this is always less than the number of checks connected just to* u *and not to* v*, this arrangement is* 1*-majority diagnosable.*

We now show the distinction between majority diagnosability and just error locatability. It can be seen that the DC graph in this example is 2*-error locatable. A quick way to prove this is to note that the following algorithm always diagnoses up to 2 errors uniquely and correctly. For each data element, compute the number of checks connected to it that output a 1. Declare the data elements with the maximum such non-zero number as erroneous and the rest as error-free! However, the above example is not* 2*-majority diagnosable. To see this, assume that the data elements in positions* $(x, y, z+1)$ *and* $(x, y+1, z)$ *are in error and the others are error-free. Data element* (x, y, z) *will now be connected to* 2 *checks that output* 1 *and only one check that outputs* 0*. The majority diagnosis algorithm will incorrectly declare* (x, y, z) *to be in error!*

Note that an s-majority diagnosable *DC* graph is automatically s-error locatable since by Definition 5.5.1 the Majority Diagnosis Algorithm correctly and uniquely diagnoses any set of errors of cardinality s or less from the syndrome. Another way to prove this is to observe that the conditions in Lemma 5.5.2 imply the sufficient conditions for s-error locatability in Theorem 5.4.3.

Before we prove the main theorem of this section, we need two lemmas from probability theory. The following *Generalized Chernoff bounds* are from [Rag90].

Lemma 5.5.4 *Let* $X_1, X_2, \cdots, X_m$ *be independent Boolean random variables with* $Prob(X_i = 1) = p$ *and* $Prob(X_i = 0) = 1 - p$*. Let* $X = \sum_{1 \le i \le m} X_i$ *and* $\delta > 0$*. Then*

$$Prob(X > (1+\delta)\mu(X)) \le \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu(X)},$$

where $\mu(X)$*, the expected (or average) value of* X*, equals* mp*.*

Lemma 5.5.5 *Let* $X_1, X_2, \cdots, X_m$ *be independent Boolean random variables with* $Prob(X_i = 1) = p$ *and* $Prob(X_i = 0) = 1 - p$*. Let* $X = \sum_{1 \le i \le m} X_i$ *and* $\delta > 0$*. Then*

$$Prob(X \le (1-\delta)\mu(X)) \le \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu(X)},$$

where $\mu(X)$*, the expected (or average) value of* X*, equals* mp*.*

Theorem 5.5.6 *The algorithm RANDGEN, using* $c = 60.5(s^2 + 2s)\log n$ *checks,* $s > 1$*, and* $p = \frac{1}{8s}$*, produces a DC graph which is* s*-majority diagnosable with probability at least* $1 - \frac{1}{n}$*.*

Proof: We need to show that all the conditions of Lemma 5.5.2 are met with a high probability. As before we try to show that the probability that one of these conditions is not met is extremely small. Let the event $E_{u,S}$ represent the event that the conditions are *not* met for $u \in D$ and $S \subset D, |S| = s$, i.e., that the cardinality of the set of checks connected to u is less than or equal to twice the cardinality of the set of checks connected simultaneously to u and some non-empty subset of $S - \{u\}$. The probability that we get a "bad" DC graph, i.e. one which is not s-majority diagnosable, simply equals $Prob(\cup_{u,S} E_{u,S})$, where u takes on all values in D and S takes on all subsets of D of cardinality s.

Given a data element $u \in D$ and a $S \subset D, |S| = s$, we bound $Prob(E_{u,S})$ as follows. We will assume that $u \notin S$. The case when $u \in S$ is similar. Let I be the cardinality of the set of checks connected to data element u. Let J be the cardinality of the set of checks connected to data element u as well as some non-empty subset of $S - \{u\}$ ($= S$ for this case). Note that both I and J can be expressed as the sum of independent binary random variables. $I = \sum_i X_i$, where X_i is 1 if check i is connected to u and 0 otherwise.

$$\mu(I) = \mu(\sum_i X_i) = \sum_{1 \le i \le c} \mu(X_i) = pc$$

Similarly, $J = \sum_i Y_i$, where Y_i is 1 if check i is simultaneously connected to u and some non-empty subset of S and 0 otherwise. It can be seen that $\mu(Y_i) = p(1 - (1-p)^s)$. Thus

$$\mu(J) = \mu(\sum_i Y_i) = p(1 - (1-p)^s)c$$

Now clearly[3]

$$\begin{aligned} Prob(E_{u,S}) &= Prob(2J \ge I) \\ &\le Prob(I \le (1-\alpha)\mu(I)) + Prob(2J \ge I | I > (1-\alpha)\mu(I)) \\ &\le Prob(I \le (1-\alpha)\mu(I)) + Prob(2J > (1-\alpha)\mu(I)) \end{aligned} \tag{14}$$

where[4] α is chosen to be 0.3968. Now we use Lemma 5.5.5 to bound the first term in Equation 14.

$$\begin{aligned} Prob(I \le (1-\alpha)\mu(I)) &\le \left(\frac{e^{-\alpha}}{(1-\alpha)^{(1-\alpha)}} \right)^{\mu(I)} \\ = e^{-(\alpha + (1-\alpha)\frac{\log(1-\alpha)}{\log e})pc} &\le e^{-0.01148\frac{c}{s}} \le \frac{1}{n^{s+2}} \end{aligned}$$

using $p = \frac{1}{8s}$ and $c = 60.5(s^2 + 2s)\log n$. Observe that for $s > 1$,

$$\frac{\mu(I)}{\mu(2J)} = \frac{1}{2(1 - (1 - \frac{1}{8s})^s)} \ge \frac{1}{2(1 - (1 - \frac{1}{16})^2)} \ge 4.129 \tag{15}$$

[3]In what follows, $Prob(E|F)$ denotes the conditional probability of event E given event F.

[4]This number was chosen to optimize the bounds that follow.

Now we use Lemma 5.5.4 to bound the second term in Equation 14.

$$\begin{aligned}
Prob(2J > (1-\alpha)\mu(I)) &= Prob\left(2J > \frac{(1-\alpha)\mu(I)}{\mu(2J)} 2\mu(J)\right) \\
&\leq Prob(J > (1+1.49)\mu(J)) \\
&\leq \left(\frac{e^{1.49}}{(1+1.49)^{(1+1.49)}}\right)^{\mu(J)} \\
&\leq e^{-.7815(0.1175pc)} \\
&\leq e^{-.01147\frac{c}{s}},\ for\ p = \frac{1}{8s} \\
&\leq \frac{1}{n^{s+2}} \qquad (16)
\end{aligned}$$

using Equation 15, the fact that $\mu(J) \geq (1 - e^{-\frac{1}{8}})pc$ and finally substituting $c = 60.5(s^2 + 2s)\log n$.

The proof of these bounds for $Prob(E_{u.S})$ when $u \in S$ is similar. Each event $E_{u,S}$ gives rise to two terms and there are $n\begin{pmatrix} n \\ s \end{pmatrix}$ events in all. Thus the probability of a "bad" DC graph is

$$Prob(\cup E_{u,S}) \leq 2n \begin{pmatrix} n \\ s \end{pmatrix} \frac{1}{n^{s+2}} \leq 2n\, \frac{n^s}{s!}\, \frac{1}{n^{s+2}} \leq \frac{1}{n},\ for\ s > 1.$$

□

As before, we can reduce the probability of getting a "bad" *DC* graph to $\frac{1}{n^k}$ by using $c = 60.5(s^2 + ks + s)\log n$ checks.

5.6 Uniform checks

It has been noted in [VJ91] that if all the checks have the same error-detection capability and check the same number of data elements then their design is simplified. Another advantage of such uniform checks is that their hardware and time overheads can also be uniform. Note that in *DC* graphs produced by RANDGEN, two checks could possibly have different error set cardinalities and hence not be identical. One can easily modify RANDGEN to make it produce uniform checks. In RANDGEN we randomly included a data element in the error set of a check with some probability p. Instead, for every check we now simply pick as its error set a random subset of the data of a fixed cardinality leading to an algorithm called UNIFGEN (for UNIForm GENeration). This algorithm has two parameters: c, the number of checks and g, the cardinality of the error set of the uniform checks. For every check, UNIFGEN picks uniformly and at random a subset of the data of cardinality g. This will be the error set of this check. Like RANDGEN, by simply varying its two parameters, UNIFGEN can generate *DC* graphs with uniform checks having a variety of useful properties.

We first consider the problem of generating uniform checks for s-error detection. We will see that requiring "uniform" checks costs us nothing, i.e., UNIFGEN uses the same number of checks as RANDGEN for generating s-error detectable DC graphs.

Theorem 5.6.1 *The algorithm UNIFGEN, using $c = 3.8s \log n$ checks and $g = \frac{n}{s}$, produces an s-error detectable DC graph with probability at least $1 - \frac{1}{n-1}$.*

Proof: The proof is similar to that of Theorem 5.3.3 and notations from the proof of that theorem will be used here. As before we need to show that the DC graph satisfies the conditions of Lemma 5.1.1, i.e., every set $S \subset D, |S| \leq s$, has a check z such that it is connected to exactly one element of S. For any single set $S, |S| = i \leq s$, and a particular check z, we first evaluate the probability that z is not connected to exactly one element of S. The total number of ways of choosing the error set of check z is clearly $\binom{n}{\frac{n}{s}}$. Of these the number of subsets which contain exactly one element from S is clearly $\binom{n-i}{\frac{n}{s}-1} i$. Hence the probability that the check is not connected to exactly one element of S is[5]

$$1 - i\frac{\binom{n-i}{\frac{n}{s}-1}}{\binom{n}{\frac{n}{s}}} = 1 - \frac{i}{s}\frac{(n-i)^{\underline{\frac{n}{s}-1}}}{(n-1)^{\underline{\frac{n}{s}-1}}} \leq 1 - \frac{i}{se} \tag{17}$$

This is the same as what we had earlier in Equation 11. The rest of the proof is similar to that of Theorem 5.3.3. □

We can, of course, reduce the probability of getting a "bad" DC graph, i.e., one that is not s-error detectable, to less than or equal to $\frac{1}{n^{k+1}-1}$ by using $(3.8s + 1.9sk) \log n$ checks, as before.

We now turn to the problem of generating an s-error locatable DC graph with uniform checks. Unlike the case of error detection, UNIFGEN does have an overhead of a small constant factor in terms of the required number of checks when compared with RANDGEN for this problem.

Theorem 5.6.2 *The algorithm UNIFGEN, using $c = (10.6s^2 + 5.3s) \log n$ checks and $g = \frac{n}{2s}$, produces an s-error locatable DC graph with probability at least $1 - \frac{1}{n}$, when $n \geq 2.8s$.*

Proof: The proof is similar to that of Theorem 5.4.4 and notations from the proof of that theorem will be used here. We need to show that the DC graph satisfies the conditions of Theorem 5.4.3. We evaluate the probability that a particular check does not satisfy the conditions as follows. The total number of ways of choosing a subset of size $\frac{n}{2s}$ is clearly $\binom{n}{\frac{n}{2s}}$. The number of subsets which contain an element u but no element from set

[5] $x^{\underline{i}}$ denotes the i^{th} falling power of x, i.e. $x(x-1)\cdots(x-i+1)$

$S, |S| = 2s-1$, is clearly $\binom{n-2s}{\frac{n}{2s}-1}$. Thus the probability that a particular check does not satisfy the conditions of Theorem 5.4.3 is

$$1 - \frac{\binom{n-2s}{\frac{n}{2s}-1}}{\binom{n}{\frac{n}{2s}}} = 1 - \frac{1}{2s}\frac{(n-2s)^{\underline{\frac{n}{2s}-1}}}{(n-1)^{\underline{\frac{n}{2s}-1}}} \leq 1 - \frac{1}{2se^{\frac{4}{3}}} \tag{18}$$

when $n \geq 2.8s$. Compare this with what we had earlier in Equation 13. Now we continue in a manner similar to the proof of Theorem 5.4.4. We need to select $c = (10.6s^2 + 5.3s)\log n \geq (\frac{4e^{\frac{4}{3}}}{\log e}s^2 + \frac{2e^{\frac{4}{3}}}{\log e}s)\log n$ to complete the proof. $\Box$

As before, we can reduce the probability of getting a "bad" *DC* graph, i.e., one which is not s-error locatable, to less than or equal to $\frac{1}{n^{k+1}}$ by using $(10.6s^2 + 5.3sk + 5.3s)\log n$ checks.

5.7 Conclusions

In this chapter, we proposed an efficient and easy to implement algorithm RANDGEN for generating *DC* graphs with a small number of checks that satisfy a variety of properties that have been found to be useful in ABFT designs. Though we have stated the results in the context of ABFT, we feel that the techniques and ideas used here will also be useful in the context of other fault tolerance problems. We introduced the concept of majority diagnosability in an attempt towards explicitly designing *DC* graphs for easy diagnosis. This is a good example of how one can simplify many issues by restricting the space of possible designs. We also examined UNIFGEN, a variation of RANDGEN, that produces *DC* graphs with uniform checks.

Our constructions were probabilistic and necessarily have a small probability of not producing a *DC* graph with the required properties. We believe that since one can decrease this probability very rapidly by adding only a few extra checks, this should not be of much concern in practice. However the question of whether there are simple deterministic constructions for these problems is of independent interest.

Bibliography

[A87] J. A. Abraham et al., “Fault tolerance techniques for systolic arrays,” *IEEE Computer*, pp. 65-74, July 1987.

[AAB$^+$92] 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.

[AB91] Y. Aumann and M. Ben-Or. Asymptotically optimal PRAM emulation on faulty hypercubes. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 440–457. IEEE Computer Society Press, October 1991.

[AB92] Y. Aumann and M. Ben-Or. Computing with faulty arrays. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 162–169, May 1992.

[Aie92] W. A. Aiello, July 1992. Personal communication.

[AL91] W. Aiello and T. Leighton. Coding theory, hypercube embeddings, and fault tolerance. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 125–136, July 1991.

[Ale82] R. Aleliunas. Randomized parallel communication. In *Proceedings of the ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, pages 60–72, August 1982.

[ALM90] S. Arora, T. Leighton, and B. Maggs. On-line algorithms for path selection in a non-blocking network. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 149–158, May 1990.

[ALMN91] W. A. Aiello, F. T. Leighton, B. M. Maggs, and M. Newman. Fast algorithms for bit-serial routing on a hypercube. *Mathematical Systems Theory*, (4):253–271, 1991.

[Ann89] F. Annexstein. Fault tolerance in hypercube-derivative networks. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 179–198, June 1989.

[AS82] G. B. Adams and H. J. Siegel. The extra stage cube: A fault-tolerant interconnection network for supersystems. *IEEE Transactions on Computers*, C–31(5):443–454, May 1982.

[AU90] S. Assaf and E. Upfal. Fault-tolerant sorting network. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 275–284, October 1990.

[AV79] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155–193, April 1979.

[B85] B. Bollobas, *Random Graphs*, Academic Press, London, 1985.

[B88] P. Banerjee et al., "An evaluation of system-level fault tolerance on the Intel hypercube multiprocessor," in *Proc. Int. Symp. Fault Tolerant Comput.*, Tokyo, pp. 362-367, June 1988.

[BA86a] P. Banerjee and J. A. Abraham. Bounds on algorithm-based fault tolerance in multiple processor systems. *IEEE Transactions on Computers*, c-35(4):296–306, April 1986.

[BA86b] P. Banerjee and J. A. Abraham, "A probabilistic model of algorithm-based fault tolerance in array processors for real-time systems," in *Proc. Real-Time Systems Symp.*, pp. 72-78, 1986.

[Bat68] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[BBN86] Butterfly^{TM} Parallel Processor Overview. BBN Report No. 6148, Version 1, Cambridge, MA, March 1986.

[BCH91] J. Bruck, R. Cypher, and C.–T. Ho. Fault-tolerant meshes with minimal numbers of spares. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 288–295, December 1991.

[BCH92a] J. Bruck, R. Cypher, and C.–T. Ho. Efficient fault-tolerant mesh and hypercube architectures. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 162–169, July 1992.

[BCH92b] J. Bruck, R. Cypher, and C.–T. Ho. Tolerating faults in a mesh with a row of spare nodes. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, December 1992. To appear.

[BCLR92] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Tolerating faults in synchronization networks. Technical Report 92–14, University of Massachusetts, Amherst, MA, April 1992.

[BCS90] J. Bruck, R. Cypher, and D. Soroker. Running algorithms efficiently on faulty hypercubes. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 37–44, June 1990.

[BCS92] J. Bruck, R. Cypher, and D. Soroker. Tolerating faults in hypercubes using subcube partitioning. *IEEE Transactions on Computers*, 41(5):599–605, May 1992.

[BH85] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–145, February 1985.

[CA86] C.-Y. Chen and J. A. Abraham, "Fault-tolerant systems for the computation of eigenvalues and singular values," in *Proc. SPIE Adv. Alg. & Arch. for Signal Proc.*, vol. 696, pp. 228-237, Aug. 1986.

[CMS82] X. Castillo, S. R. McConnel, and D. P. Siewiorek. Derivation and calibration of a transient error reliability mode. *IEEE Transactions on Computers*, C-31:658–671, July 1982.

[CM88a] Y.-H. Choi and M. Malek, "A fault tolerant FFT processor," *IEEE Trans. Comput.*, vol. 37, pp. 617-621, May 1988.

[CM88b] Y.-H. Choi and M. Malek, "A fault tolerant systolic sorter," *IEEE Trans. Comput.*, vol. 37, pp. 621-624, May 1988.

[Dal87] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer, Hingham, MA, 1987.

[DH90] S. Dutt and J. P. Hayes. On designing and reconfiguring k-fault-tolerant tree architectures. *IEEE Transactions on Computers*, C–39(4):490–503, April 1990.

[DH91] S. Dutt and J. P. Hayes. Designing fault-tolerant systems using automorphisms. *Journal of Parallel and Distributed Computing*, 12:249–268, 1991.

[EL92] S. Even and A. Litman. Layered Cross Product – a technique to construct interconnection networks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 60–69, July 1992.

[ES74] P. Erdos and J. Spencer, *The Probabilistic Method in Combinatorics*, Academic Press, London, 1974.

[Fel85] M. R. Fellows. *Encoding Graphs in Graphs.* PhD thesis, Department of Computer Science, University of California, San Diego, CA, 1985.

[GE84] 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.

[GHR90] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg. Optimal embeddings of butterfly-like graphs in the hypercube. *Mathematical Systems Theory*, 1990.

[GL89] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. In Silvio Micali, editor, *Randomness and Computation.* Volume 5 of *Advances in Computing Research*, pages 345–374. JAI Press, Greenwich, CT, 1989.

[Got87] A. Gottlieb. An overview of the NYU Ultracomputer Project. In J. J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 25–95. Elsevier Science Publishers, B. V., Amsterdam, The Netherlands, 1987.

[GRR90] D. Gu, D. J. Rosenkrantz, and S. S. Ravi, "Design and analysis of test schemes for algorithm-based fault tolerance," in *Proc. Int. Symp. Fault Tolerant Comput.*, pp. 106-113, Newcastle-upon-Tyne, U.K., June 1990.

[HA84] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, c-33(6):518–528, June 1984.

[HLN87] J. Hastad, T. Leighton, and M. Newman. Reconfiguring a hypercube in the presence of faults. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 274–284, May 1987.

[HLN89] J. Hastad, T. Leighton, and M. Newman. Fast computation using faulty hypercubes. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 251–263, May 1989.

[Hoe56] W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27:713–721, 1956.

[IR83] R. K. Iyer and D. J. Rossett. Permanent cpu errors and systems activity: Measurement and modeling. In *Proc. Real-Time Systems Symposium*, December 1983.

[JA86] J.-Y. Jou and J. A. Abraham, "Fault tolerant matrix arithmetic and signal processing on highly concurrent computing structures," *Proc. IEEE*, vol. 74, pp. 732-741, May 1986.

[JA88] J.-Y. Jou and J. A. Abraham, "Fault tolerant FFT networks," *IEEE Transactions on Computers*, vol. 37, pp. 548-561, May 1988.

[KKL$^+$90] 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.

[KLM$^+$89] 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.

[Koc88] R. R. Koch. Increasing the size of a network by a constant factor can increase performance by more than a constant factor. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 221–230. IEEE Computer Society Press, October 1988.

[KS83] C. P. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, C–32(12):1091–1098, December 1983.

[L85] F. T. Luk, "Algorithm-based fault tolerance for parallel matrix equations solvers," in *Proc. SPIE Real Time Signal Proc.*, vol. 564, pp. 49-53, Aug. 1985.

[Lei85] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C–34(10):892–901, October 1985.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.

[Lin92] G. Lin. Fault tolerant planar communication networks. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 133–139, May 1992.

[LM92] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Transactions on Computers*, 41(5):578–587, May 1992.

[LMP91] T. Leighton, Y. Ma, and C. G. Plaxton. Highly fault-tolerant sorting circuits. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 458–469. IEEE Computer Society Press, October 1991.

[LMR88] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 256–271. IEEE Computer Society Press, October 1988.

[LMRR] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. Journal of Algorithms. To appear.

[LMS92a] 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. IEEE Computer Society Press, October 1992.

[LMS92b] T. Leighton, B. Maggs, and R. Sitaraman. On the fault tolerance of some popular bounded-degree networks. Technical Report CS–TR–385–92, Department of Computer Science, Princeton University, Princeton, NJ, Sept. 1992.

[LP86] F. T. Luk and H. Park, "An analysis of algorithm-based fault tolerance techniques," in *Proc. SPIE Adv. Alg. & Arch. for Signal Proc.*, vol. 696, pp. 222-228, Aug. 1986.

[LP88] F. T. Luk and H. Park, "Fault tolerant matrix triangularizations on systolic arrays," *IEEE Trans. Comput.*, vol. 37, pp. 1434-1438, Nov. 1988.

[LSGH87] M. Livingston, Q. Stout, N. Graham, and F. Harary. Subcube fault-tolerance in hypercubes. Technical Report CRL-TR-12-87, University of Michigan Computing Research Laboratory, September 1987.

[Lyu90] Y.–D. Lyuu. Fast fault-tolerant parallel communication and on-line maintenance using information dispersal. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 378–387, July 1990.

[Mat92] 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.

[MR84] T.E. Mangir and C.S. Raghavendra. Issues in the implementation of fault-tolerant VLSI and wafer-scale integrated systems. In *Proc. ICCD 84*, 1984.

[MS92] B. M. Maggs and R. K. Sitaraman. Simple algorithms for routing on butterfly networks with bounded queues. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 150–161, May 1992.

[NA88] V. S. S. Nair and J. A. Abraham, "A model for the analysis of fault tolerant signal processing architectures," in *Proc. 32nd Int. Tech. Symp. of SPIE*, San Diego, pp. 246-257, Aug. 1988.

[NA89] V. S. S. Nair and J. A. Abraham, "A model for the analysis, design and comparison of fault-tolerant WSI architectures," in *Proc. Workshop on Wafer Scale Integration*, Como, Italy, June 1989.

[NA90] V. S. S. Nair and J. A. Abraham, "Hierarchical design and analysis of fault-tolerant multiprocessor systems using concurrent error detection," in *Int. Symp. Fault Tolerant Comput.*, Newcastle-upon-Tyne, U.K., pp. 130-137, June 1990.

[NMT$^+$91] T. Nakata, S. Matsushita, N. Tanabe, N. Kajihara, H. Onozuka, Y. Asano, and N. Koike. Parallel programming on Cenju: A multiprocessor system for modular circuit simulation. *NEC Research & Development*, 32(3):421–429, July 1991.

[NSS89] R. Negrini, M.G. Sami, and R. Stefanelli. *Fault Tolerance through Reconfiguration in VLSI and WSI Arrays.* The MIT Press, 1989.

[OT71] D. C. Opferman and N. T. Tsao-Wu. On a class of rearrangeable switching networks–part II: Enumeration studies and fault diagnosis. *Bell System Technical Journal*, 50(5):1601–1618, May–June 1971.

[PBG+87] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. An introduction to the IBM Research Parallel Processor Prototype (RP3). In J. J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 123–140. Elsevier Science Publishers, B. V., Amsterdam, The Netherlands, 1987.

[Pip84] N. Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 127–136. IEEE Computer Society Press, October 1984.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), April 1989.

[Rag88] P. Raghavan. Probabilistic construction of deterministic algorithms: approximate packing integer programs. *Journal of Computer and System Sciences*, 37(4):130–143, October 1988.

[Rag89] 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.

[Rag90] P. Raghavan. Lecture notes on randomized algorithms. Research Report RC 15340 (#68237), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, January 1990.

[Ran87a] A. G. Ranade. Constrained randomization for parallel communication. Technical Report YALEU/DCS/TR-511, Department of Computer Science, Yale University, New Haven, CT, 1987.

[Ran87b] A. G. Ranade. Equivalence of message scheduling algorithms for parallel communication. Technical Report YALE/DCS/TR-512, Department of Computer Science, Yale University, New Haven, CT, 1987.

[Ran87c] A. G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 185–194. IEEE Computer Society Press, October 1987.

[RB90] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, vol. 39, pp. 1304-1308, Oct. 1990.

[RR88] D. J. Rosenkrantz and S. S. Ravi, "Improved upper bounds for algorithm-based fault tolerance," in *Proc. 26th Allerton Conf. Comm. Cont. & Comput.*, Allerton, IL, pp. 388-397, Sept. 1988.

[RK75] J. D. Russel and C. R. Kime, "System fault diagnosis: Closure and diagnosability with repair," *IEEE Trans. Comput.*, vol. C-24, pp. 1078-1089, Nov. 1975.

[Sch90] 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.

[Sch91] E. J. Schwabe. *Efficient Embeddings and Simulations for Hypercubic Networks.* PhD thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA, June 1991.

[SJ] R. K. Sitaraman, N. K. Jha. Optimal Design of Checks for Error Detection and Location in Fault Tolerant Multiprocessor Systems. *IEEE Transactions on Computers.* To appear.

[SJ91] R. K. Sitaraman, N. K. Jha. Optimal Design of Checks for Error Detection and Location in Fault Tolerant Multiprocessor Systems. In *Proceedings of the Fault Tolerant Computing Systems Conference*, pages 396–406, September 1991

[SNS$^+$89] H. Suzuki, H. Nagano, T. Suzuki, T. Takeuichi, and S. Iwasaki. Output-buffer switch architecture for asynchronous transfer mode. In *Proceedings of the 1989 IEEE International Conference on Communications*, pages 99–103, June 1989.

[Spe87] J. Spencer. *Ten Lectures on the Probabilistic Method.* SIAM, Philadelphia, PA, 1987.

[SR80] S. Sowrirajan and S. M. Reddy. A design for fault-tolerant full connection networks. In *Proceedings of the International Conference on Science and Systems*, pages 536–540, March 1980.

[SRHA86] Y. Savaria, N. C. Rumin, J. F. Hayes, and V. K. Agarwal. Soft-error filtering: A solution to the reliability problem of future VLSI digital circuits. *Proceedings of the IEEE*, 74(5):669–683, May 1986.

[SS89] T. Szymanski and S. Shaikh. Markov chain analysis of packet-switched banyans with arbitrary switch sizes, queue sizes, link multiplicities and speedups. In *Proceedings of the IEEE INFOCOM '89*, pages 960–971, April 1989.

[ST91] G. D. Stamoulis and J. N. Tsitsiklis. The efficiency of greedy routing in hypercubes and butterflies. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 248–259, July 1991.

[Tam92a] 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.

[Tam92b] 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, July 1992.

[THC90] D. L. Tao, C. R. P. Hartmann, and Y. S. Chen, "A novel concurrent error detection scheme for FFT networks," in *Proc. Int. Symp. Fault Tolerant Comput.*, Newcastle-upon-Tyne, U.K., pp. 114-121, June 1990.

[Tsa90] A. M. Tsantilas. *Communication Issues in Parallel Computation.* PhD thesis, Harvard University, Cambridge, MA, 1990.

[Upf84] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31(3):507–517, July 1984.

[Val82] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[Val90] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 943–971. Elsevier Science Publishers, B. V., Amsterdam, The Netherlands, 1990.

[VB81] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, May 1981.

[VJ89] B. Vinnakota and N. K. Jha, "Diagnosability and diagnosis of algorithm-based fault tolerant systems," in *Proc. 32nd Midwest Symp. Circuits & Systems*, Urbana, IL, pp. 28-31, Aug. 1989.

[VJ90] B. Vinnakota and N. K. Jha, "A dependence graph-based approach to the design of algorithm-based fault tolerant systems," in *Proc. Int. Symp. Fault Tolerant Comput.*, Newcastle-upon-Tyne, U.K., pp. 122-129, June 1990.

[VJ91] B. Vinnakota and N. K. Jha, "Design of multiprocessor systems for concurrent error detection and fault diagnosis," in *Proc. Int. Symp. Fault Tolerant Comput.*, Montreal, June 1991.

[Wak68] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, January 1968.

[WC92] A. Wang and R. Cypher. Fault-tolerant embeddings of rings, meshes and tori in hypercubes. Technical Report IBM RJ 8569, IBM Almaden Research Center, January 1992.

[WCM91] A. Wang, R. Cypher, and E. Mayr. Embedding complete binary trees in faulty hypercubes. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 112–119. IEEE Computer Society Press, December 1991.

[YFA87] M. M. Yen, W. K. Fuchs, and J. A. Abraham. Designing for concurrent error detection in VLSI: Application to a microprogram control unit. *IEEE J. Solid-State Circuits*, SC-22:595–605, August 1987.