LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-500

RANDOMNESS AND ROBUSTNESS IN HYPERCUBE COMPUTATION

Mark Joseph Newman

April 1991

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Randomness and Robustness in Hypercube Computation

Mark Joseph Newman

Submitted to the Department of Mathematics
on July 28, 1989, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis we explore means by which hypercubes can compute despite faulty processors and links. We also study techniques which enable hypercubes to simulate dynamically changing networks and data structures.

In chapter two, we investigate strategies for routing permutations on faulty hypercubes. We assume that each node or edge in the hypercube fails with fixed probability $p < 1 - \sqrt[6]{1/2}$ and that failures are independent of one another. We describe a constant $c > 0$ and a routing algorithm which successfully routes messages between working processors in $O(\log N)$ steps on an N-node faulty hypercube, with probability $1 - N^{-c}$. We also strengthen an algorithm due to Rabin which uses a redundant encoding of each message into $\log N$ pieces which are routed along node-disjoint paths. A destination can reconstruct the original message as long as at least $\log N/2$ pieces arrive intact. We show that all messages are reconstructable at their destinations with high probability, given that each node or edge fails with probability $O(1/\log N)$ and that each message has $\Omega(\log^2 N)$ bits. This guarantee obtains even if the components fail during the course of the algorithm.

In chapter three, we develop techniques for reconfiguring hypercubes in the presence of faults. Again assuming constant probabilities of failure and the independence of faults, we show that a faulty hypercube can simulate a fault-free hypercube of the same size with only constant delay. We exhibit both deterministic and randomized algorithms for hypercube reconfiguration. We show that there exists a constant $c' > 0$ such that with probability $1 - N^{-c'}$ the deterministic algorithm finds a one-to-one embedding with dilation 3 and $O(\log N)$ congestion. We also show that there exists a constant $c'' > 0$ such that with probability $1 - N^{-c''}$ the randomized algorithm finds an embedding with constant load and congestion with dilation 5.

In chapter four, we turn our attention to the embedding of dynamically growing data structures in the hypercube. Specifically, we show that an arbitrarily growing binary tree with a maximum of M nodes can be embedded in an N-node hypercube with load $O(\frac{M}{N}+1)$, congestion $O(\frac{M}{N}+1)$ and dilation 12, with high probability. We also show how to embed a dynamic M-node binary tree in an N-node butterfly with $O(\frac{M}{N}+\log N)$ load and dilation 2, with high probability.

Thesis Supervisor: F. Thomson Leighton
Title: Professor of Applied Mathematics

Acknowledgements

In my academic career, I have received far more than my fair share of support, both moral and intellectual. First credit goes to Tom Leighton, who has guided my development with a loose hand. Tom has always given me nothing but encouragement and good advice to get me through an often ego-threatening situation. He has also co-authored all the work appearing herein.

Others from whose effort I have benefited include Johan Hastad (with whom chapter three was written), Abhiram Ranade and Eric Schwabe (co-authors of chapter four). For five years I depended on Bill Aiello to help me through difficult proofs, to exchange neat tricks and to listen to my unintelligible attempts at new ideas. The proof of lemma 2.7 is the product of joint work with Bill and Satish Rao. I also learned much from talking with Ravi Boppana, Tom Cormen, Lance Fortnow, Bruce Maggs, Seth Malitz, James Park and Peter Shor. Despite a full schedule filled with his own advisees, Charles Leiserson has always been willing to lend an ear and to suggest fruitful avenues of attack. David Shmoys and I first met when I was a freshman in college and he a senior. Since then, he has never neglected an opportunity to teach me or to steer me in the right direction. It has been my luck that our tenure at MIT coincided so neatly.

Ron Rivest has made the theory group at the Laboratory for Computer Science the most dynamic, supportive, and friendly of all the academic environments I have seen. In addition to befriending all the students she can, Be Hubbard makes sure that everything anyone needs is right where he needs it. I suspect that this is a much more difficult job than she lets on. Whenever I had any computing questions, Sally Bemus came to my aid immediately.

At three vastly different times, Irwin Kaufman, Hilda Singer and Douglas West returned my mathematical interest with a dedication surpassing any expectation. I

was fortunate enough to receive such an extreme amount of attention from each of them.

I will always remain thankful for the friendship of Bill Aiello, Johan Hastad, Ingrid Johnson, Karen Parrish, Nir Shavit, David Shmoys and Éva Tardos, which saw me through these past five years. For the past four years, James Park has been an exemplary roommate and fellow jazz novitiate. My deepest thanks and love must be reserved for Gayle Augenbaum, who has always absorbed more complaints and frustration than is her due. Her warm smile and healing hand have often been the difference between happiness and unhappiness.

In addition to giving unstintingly of their love, my parents have always encouraged me to pursue knowledge for no end other than its own. Even though their workdays were spent teaching others, they never relented in the education of their own children. Despite nine years of physical separation, I feel as close to my sisters Amy and Nancy as ever. To have two older sisters is to be spoiled in the best of ways - with affection, good advice and happy times. Now that they have families of their own, I have the added benefit of two older brothers, two nieces and a nephew as well. All along, my uncle Don has been the only relative with whom I can discuss my work. Last, I would like to thank my uncle Alan for understanding what I care about most and for killing all the triple word scores he can find.

Contents

1 Introduction **8**

1.1 Hypercubes 8

1.2 Robustness 10

1.3 Fault-Tolerant Routing 12

1.4 Reconfiguration 14

1.5 Dynamic Load Balancing 16

2 Routing in the Presence of Faults **18**

2.1 Introduction 18

2.1.1 Summary of Results 19

2.1.2 Overview 22

2.2 Fast Routing Around Faults 22

2.2.1 Valiant-Brebner Routing 24

2.2.2 Jump Edges 27

2.2.3 Offset Routing 29

2.2.4 The Length of Offset Paths 30

2.2.5 Delay From Other Packets 33

2.3 Information Dispersal Routing 40

2.3.1 Routing Along Parallel Paths 41

2.3.2 Fault-Tolerant Encoding of Pieces 43

2.3.3 Fault-Tolerance via Parallel Paths 44

2.4 Remarks 46

3 Reconfiguration in the Presence of Faults **47**

3.1 Introduction 47

3.1.1 Summary of Results 51

3.1.2 Overview 53

3.2 Embeddings for Small p with Dilation 3 53

3.2.1 Mapping Dead Nodes to Live Nodes 53

3.2.2 Analysis of the Borrowing 54

3.2.3 Embedding Edges 56

3.3 Embeddings with Dilation 3 for $p < 1/2$ 57

3.3.1 Analyzing Stages 1 and 2 58

3.3.2 Analyzing Stages 3 and 4 61

3.4 Routing Using Only Live Nodes 63

3.5 An Algorithm for Constant Delay Embedding 66

3.5.1 Assigning Nodes to Live Neighbors 67

3.5.2 Assigning Edges to Paths 70

3.6 Implementing the Constant Delay Embedding 75

3.7 Extensions and Remarks 79

4 Embedding Trees Dynamically **81**

4.1 Introduction 81

4.1.1 Summary of Results 82

4.1.2 Overview 84

4.2 The Basic Growth Algorithm 84

4.2.1 Preliminary Scheme 84

4.2.2 Flip Bits 85

4.3 Embedding in the Butterfly 89

4.3.1 A Level-Balancing Transformation 89

4.3.2 Analysis of Tree Balancing 90

4.3.3 Effectiveness of Flip Bits 93

4.4 An Improved Hypercube Embedding 95
4.4.1 Embedding the Butterfly and Star Covers 95
4.4.2 Modifying the Embedding Algorithm 96
4.4.3 Redistributing Load Within Stars 99
4.5 A Lower Bound for Deterministic Algorithms 102
4.6 Remarks 103

Chapter 1

Introduction

1.1 Hypercubes

The hypercube has emerged as one of the most effective and popular network architectures for large scale parallel computers. The Connection Machine, manufactured and sold by Thinking Machines Corp., is a hypercube-based machine containing 2^{16} processing elements. Machines based on hypercube architectures have been built by Intel, Ncube, Caltech and others. It has been predicted that in the not-too-distant future, hypercube-based machines containing up to a million processors will be available. Thus, current conditions point to the utility of more advanced methods for hypercube computation.

The n-dimensional hypercube H_n is a graph with $N = 2^n$ nodes and $Nn/2$ edges. The nodes of H_n are labeled with n-bit binary strings, and two nodes are linked by an edge if the associated strings differ in precisely one bit. If the differing bit is in the i^{th} position ($1 \leq i \leq n$) then the associated edge is called a *dimension i edge.* The neighbor of a node v across the i^{th} dimension will be denoted by v^i. Similarly $v^{i_1 i_2 \ldots i_k}$ will denote the node reached from v by traversing dimensions $i_1, i_2, \ldots, i_k$ (that is, by flipping those bits). We will use n and $\log N$ interchangeably. Pictures of labeled two and three dimensional hypercubes and an unlabeled four dimensional hypercube appear in figures 1-1 and 1-2.

In hypercube-based machines, the nodes of the graph are replaced by processors

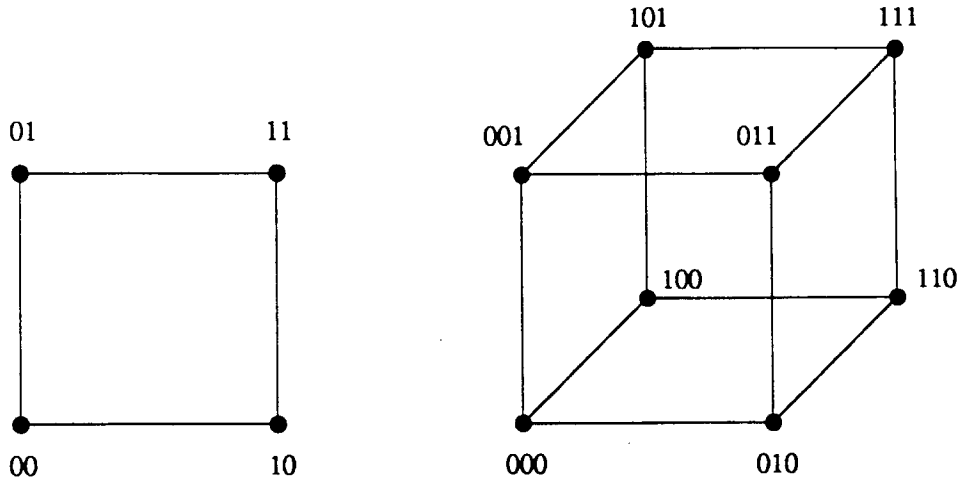

Figure 1-1: Labeled 2- and 3-dimensional hypercubes.

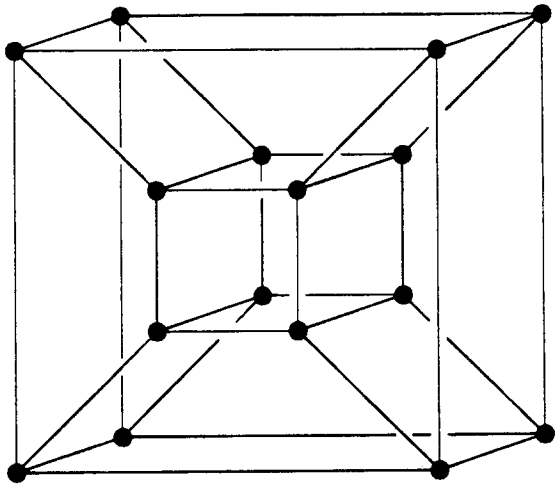

Figure 1-2: A 4-dimensional hypercube.

and the edges are replaced by links between the processors. For example, in the Connection Machine each node of a 12-dimensional hypercube contains a group of 16 processors.

The effectiveness of the hypercube for parallel computation arises from the wealth of special-purpose algorithms written for it, its support of algorithms written for shared-memory machines and its ability to simulate a host of other networks. Many algorithms which run quickly on the hypercube already exist. Further, the hypercube's recursive structure and high connectivity make it likely that fast hypercube algorithms will continue to be invented in other contexts.

Hypercubes have demonstrated their usefulness as general-purpose computers as well. Fast routing algorithms ([VB], [Ran], [P]) allow for low-overhead interprocessor communication. These algorithms enable the hypercube to simulate a parallel random access machine, or PRAM, with only logarithmic delay. Since any set of messages are deliverable in $O(\log N)$ time, each set of memory accesses can be simulated in $O(\log N)$ time as well, even if the PRAM's processors and memory locations are spread arbitrarily among the hypercube's processors.

Hypercubes perform even more admirably when simulating special-purpose networks. The hypercube can simulate meshes, multidimensional arrays, binary trees, x-trees, pyramid graphs, butterflies, cube-connected cycles and other networks, all with constant delay. In many cases, these other networks are actually subgraphs of the cube. In these instances, the hypercube can simulate the special-purpose network with no delay at all.

1.2 Robustness

In this thesis we will describe three ways in which the hypercube is robust in a changing computational environment. Specifically, we show how the hypercube can support fault-tolerant routing, how the hypercube can be easily reconfigured in the presence of faults and how the hypercube can handle dynamically changing load requirements. In the first two cases, the network itself changes due to the accumulation of faulty

processors and links. We show how the network can absorb these faulty components while exhibiting little or no degradation of performance. In the third case, the computation we expect the network to perform changes in accordance with the data in an unpredictable fashion. We show how the network can distribute the resulting computational load as optimally as if it had been completely specified beforehand. In all three cases, a probabilistic approach helps us to achieve our results. In some cases, we prove that these results would be impossible if randomness were not available.

In chapters two and three, we explore fault tolerant properties of the hypercube. We assume that each node or edge has some constant probability of failure. In chapter two we exhibit two randomized algorithms for routing permutations on hypercubes in the presence of faulty components. Both algorithms are based on Valiant and Brebner's ([VB]) original randomized algorithm for routing permutations on hypercubes. In the first algorithm, we modify the fault-free algorithm so that messages avoid faults. In the second algorithm, packets are broken into pieces containing redundant information. Since only a constant fraction of the pieces need to get through to reconstruct the original packet, the algorithm can tolerate the loss of many pieces due to faults. To route a permutation, neither algorithm takes more than a constant factor more time than is required to route without faults.

Chapter three is devoted to reconfiguration algorithms. The effect of these algorithms is that the nonfaulty processors of a hypercube with faults simulate the processors of a completely functioning hypercube. The link connecting two processors in the completely functioning hypercube appears as a functioning path between the nodes simulating them in the cube with faults. In chapter three, we describe reconfiguration algorithms which enable a hypercube with many faults to compute as efficiently as a hypercube of the same size without faults.

The efficient simulation of dynamically evolving computation structures is the subject of chapter four. We show that a hypercube can simulate an arbitrarily growing binary tree with only constant overhead. As the tree evolves, new nodes are assigned to hypercube processors. Neighbors in the tree are simulated by hypercube processors only a constant distance apart. For any tree, the randomized algorithm assigns only

a constant number of tree nodes to each processor with a probability that can be made arbitrarily close to 1. Thus both computation and communication overhead are minimized.

In sections 1.3 - 1.5, we give an overview of the results in each of chapters two, three and four.

1.3 Fault-Tolerant Routing

Given a network with a large number of components, we must assume that some of these components will fail. These faults may be introduced when the machine is first built, or might accumulate over time. We would like the machine to work despite the faults.

Currently, when a processor or connection in the Connection Machine fails, the board containing the offending component is removed and replaced with a functional board. At some point in the future, if and when very large machines are in general use, fault-tolerant algorithms may well provide a viable alternative to wholesale replacement. Such algorithms might enable the machine to correct itself, with no outside intervention.

Fault-tolerant behavior will be a major focus of our work. Routing in the presence of faults, which we study in chapter two, requires techniques for either stepping around faults or coping with messages which run into faults. Attempts have been made on both of these fronts. We consider a routing algorithm successful if every packet sent from a working processor to another working processor arrives intact. Of course, this view presupposes that the higher-level algorithm in effect is also tolerant of faulty processors. For example, a PRAM algorithm would have to tolerate some pattern of faults among the PRAM's processors. Such algorithms have yet to be designed.

Throughout chapter two, we assume that there is some fixed probability p (either a constant or a function of the number N of nodes in the network) such that each component of the hypercube fails with probability p. Furthermore, we will assume that the failure of any given component is independent of the status of other parts

of the network. In some cases, this independence assumption may be unreasonable. Components which share a physical location such as a chip or a board might have a greater chance of failing in tandem. In this situation, our results can scale to work in a hierarchical fashion. We may regard any hypercube as a hypercube whose nodes are themselves hypercubes (a cross product of hypercubes). Thus we may treat the chips or boards as nodes in a more coarse-grained hypercube.

Many of our algorithms are randomized as well. These algorithms have access to a source of randomness and we only guarantee that they achieve desired results an overwhelmingly large fraction of the time. Specifically, we guarantee that each algorithm succeeds with probability at least $1 - N^{-k}$; i.e. that each fails with a probability that is an inverse polynomial in N. If we can make the exponent k as large as we like (perhaps by relaxing constants in the performance we desire), then we say that the algorithm succeeds with high probability.

In [VB], Valiant and Brebner define a set of paths from sources to destinations which, with high probability, allow all packets to arrive at their destinations in $O(\log N)$ steps. Two different variations on Valiant and Brebner's ideas allow us to route in the presence of faults. These variations use different assumptions about the prevalence of faults, the capability of processors to detect faults in neighboring components, and the minimum size of the packets that we can route. In the first case, we assume that faults occur independently and with constant probability p, that each processor can detect in one time unit whether or not an adjacent node or link has failed, and that messages have length $\Omega(\log N)$. Our idea is for packets to follow close to the paths defined in [VB], but loosely enough that they can avoid faults as they encounter them. We show that if each packet avoids faults by taking random steps away from its Valiant-Brebner path, then with high probability each packet uses a path with only $O(\log N)$ edges and encounters only $O(\log N)$ other packets on its path. This shows that each packet arrives at its destination in $O(\log N)$ steps.

We devote the second half of chapter two to our improvements of an idea of Rabin ([R]). In this case, we assume that each edge of the hypercube fails with probability $p = O(1/\log^2 N)$, that processors remain ignorant of changes in the topology of the

network, and that packets have size $\Omega(\log^2 N)$. Under these assumptions, Rabin showed that if each packet is split into $\log N$ pieces and the pieces are routed to the packet's original destination by node-disjoint paths, then a constant fraction of each packet's pieces will arrive intact at the destination. This assumes that each piece makes no attempt to avoid faults. A piece arrives at the destination if and only if no faults lie on its path. Coupled with a method for recovering a packet from a constant fraction of its pieces, this strategy allows us to choose paths as if the faults were not there. We describe a very simple way to choose the paths—we use $\log N$ paths parallel to the Valiant-Brebner path. We are then able to simplify the proof, to allow node failures as well, and to increase the allowable failure rates to include probabilities as high as $p = O(1/n)$. (Recently, Giladi has reported similar results ([G]).)

1.4 Reconfiguration

Network reconfiguration involves assigning to working components the tasks that the failed components would otherwise perform. The goal is to leave the network's processing power undiminished in the eyes of the outside world, except perhaps for a minor slowdown in speed to allow some components to perform multiple duty. Alternatively, we can view reconfiguration as the embedding of a fault-free network H'_n of the same size into the working parts of the faulty network H_n. We can show that even if a constant fraction of the hypercube's processors and links fail, what remains keeps the original cube's processing power with only a constant factor degradation in speed, with high probability.

We make the same probabilistic fault assumptions in chapter three that we made in chapter two. Each component fails with constant probability and independently of other components.

Some of our techniques may be of use with other hypercube-related problems. In particular, there is one simple observation that is used in two forms in section 3.5. Although the observation has probably been made by others, it is basic enough that

we think it worth highlighting as a paradigm for distributed match-making.

We will describe the result in its most basic form. Consider a collection of $\Theta(N)$ men and $\Theta(N)$ women at a dance. Assume that each man has at least $\Omega(X)$ female friends and that each woman has at most $O(X)$ male friends. By Hall's marriage theorem, it is possible to schedule $O(1)$ rounds of dances so that every man dances with at least one friend and every woman dances at most $O(1)$ times. Unfortunately, the problem of scheduling dance partners requires substantial global coordination. For our purposes, we focus on a scenario where pairing is accomplished simply by a man asking a woman to dance. If many men ask a woman to dance at once, she accepts as many as she can, making sure not to exceed her capacity of $C = O(1)$ dances for the evening. If she can only accept some of the men, she prefers the tallest among them. Each man chooses a friend randomly for each dance (without knowledge of which women are tired or which women other men are asking) until he dances. The result (which we call the Dance Hall Theorem—pun intended) is that if $X = \Omega(\log N)$, and there are $\Omega(\log N)$ dances, then with high probability every man will dance during the course of the evening. That is, for any lower bound bX on the number of female friends each man has, any upper bound $b'X$ on the number of male friends each woman has and any constant k, there is a C such that for sufficiently large N, with probability $1 - N^{-k}$ a capacity of C is sufficient.

The Dance Hall Theorem scenario first arises in our analysis when we attempt to embed the nodes of H'_n in the functioning nodes of H_n. The nodes of H'_n correspond to men and the functioning nodes of H_n correspond to women. If a man dances with a woman, then the corresponding node of H'_n will be simulated by the corresponding node of H_n. We need the Dance Hall Theorem to ensure that the load of the embedding is $O(1)$ (i.e. every woman dances with $O(1)$ men) and to ensure that the embedding can be constructed quickly with local control (no global matchmaker). We also need some other as-yet-undescribed properties of the Dance Hall Theorem schedule to ensure that the hypercube's edges are not overtaxed by the embedding, but these are more technical in nature and will be dealt with in the main text.

1.5 Dynamic Load Balancing

The desire for the optimal use of computational resources is often modelled as an embedding problem. We construct a graph whose nodes represent the data and processes. An edge connects two processes which trade information. To minimize computation time, we would like to divide the processing requirements evenly among the processors of our network. To minimize communication time, we would like to assign neighboring processes to processors which are fairly close. These two requirements may conflict.

For one solution, we might build a network which perfectly mirrors the processes involved and embed each process in its own processor. There are two problems with this approach. First, every algorithm would require a different network structure depending upon how it divided up the work. Worse, the same algorithm might generate a different process graph for different input data. In this case no foresight could help in network construction. One (far from unique) example can be found in the context of branch-and-bound algorithms. The search tree developed during each run of a branch-and-bound algorithm changes based on which subtrees are cut and which are chosen for further exploration. We could not hope to build a processor tree which could handle all potentialities unless it were far larger than any one tree that might be generated during any particular run.

As a second solution, we might build a network into which all similarly sized trees can be embedded. A practical network would allow us to embed a tree dynamically. As we embed the tree, we have no knowledge of which branches will develop many nodes in the future, and which will cease to exist at all. We must allow sufficient room for all possibilities.

In chapter four, we demonstrate a randomized algorithm which, with high probability, embeds an arbitrary dynamic binary tree in a hypercube so that the computation and communication overhead are both constant. A simplified version of the algorithm embeds a dynamically growing tree in a butterfly smaller by a logarithmic factor. Both computation and communication are slowed by only a logarithmic fac-

tor, the best possible. Thus hypercubes and butterflies can run tree-based algorithms as quickly as can PRAMs.

Chapter 2

Routing in the Presence of Faults

2.1 Introduction

To successfully simulate shared memory, a parallel network must have the ability to route information between different origin processors and destination processors at the same time. Since processors trade information throughout the course of parallel computations, the overhead due to the transmission of information over the network shows up as a multiplicative factor in the time to perform many tasks. Thus the routing question is one of fundamental importance.

In practice and theory, the store-and-forward model of communication is often used. In this model, once a node begins transmission of a message unit across a link, it continues to transmit until the entire message is sent. Treating messages as inviolable *packets* allows us to ignore some significant issues of control at the cost of time. Since time bounds for packet-switched networks are often stated in units of packet steps, such bounds must be multiplied by the length of the longest message to produce a bound in bit steps.

Many algorithms have appeared for routing on hypercubes and networks derived from hypercubes (such as the butterfly). In 1981, Valiant and Brebner ([VB]) presented an algorithm for routing $\Omega(\log N)$-bit packets on the $\log N \times N$-node butterfly (and hence the N-node cube) which could route permutations from the top level to

the bottom in $O(\log N)$ packet steps, with high probability.[1] Here a permutation means that the mapping from origins to destinations is bijective. Their algorithm, which we will review in section 2.2, introduced the paradigm of routing each packet first to a random intermediate destination and then to its true final destination. The algorithm routes obliviously: each packet's path is chosen without regard for the paths of any other packets.

This simple addition of randomness is enough to overcome the proven delays involved with deterministic routing algorithms. Borodin and Hopcroft ([BH]) showed that any deterministic oblivious algorithm must necessarily take $\Omega(\sqrt{N}/(\log N)^{3/2})$ bit steps, in the worst case, for any N-node network.

Since Valiant and Brebner's pioneering work, significant improvements have been made. Pippenger ([P]) showed how to route permutations of a fully loaded $\log N \times N$ butterfly in $O(\log N)$ steps with high probability. That is, each node in the butterfly can generate a packet, not only the nodes in the top level. In Pippenger's algorithm, only a constant number of packets reside in a queue at any time. Ranade ([Ran]) produced an algorithm which routes arbitrary mappings on a fully-loaded butterfly using combining, again with constant size queues and in $O(\log N)$ packet steps with high probability. Both of these algorithms make fundamental use of the paradigm of routing to random intermediate destinations.

2.1.1 Summary of Results

In this chapter, we consider the problem of packet routing on a hypercube with faults. We assume that every node and link of a hypercube fails independently with constant probability p. Under this assumption, with probability exponentially close to 1, a constant fraction of the components of the cube will fail. In the presence of such a large number of faults, we would like to route packets so that any packet generated by a working node and sent to a working node arrives safely within the stated time bound.

[1] We use the phrase Q is less than $O(g)$ with high probability to mean "For every k there exists a constant d independent of N such that the probability that Q exceeds dg is less than N^{-k}."

We describe and analyze a randomized packet routing algorithm that adaptively routes packets around faults as they are encountered in an N-node hypercube that contains $\Theta(N)$ randomly located faulty nodes and $\Theta(N \log N)$ randomly located faulty edges. We assume that each processor can decide if an adjacent node or link has failed. Also, each processor can choose a random element from a set with as many as $\log N$ elements according to the uniform distribution. We define the property of local routability, a characterization of the connectivity of the network after some components have failed. There exists a constant c_1 such that the hypercube remains locally routable with probability $1 - N^{-c_1}$. We prove that, given that the hypercube is locally routable, the algorithm routes any permutation on the working processors in $O(\log N)$ steps with high probability. That is, under the assumption of local routability, we reproduce Valiant and Brebner's results in the presence of faulty components. Packets which start or end at faulty nodes are eventually determined to be undeliverable. All the deliverable packets arrive at their destinations provided that they are not located in the immediate vicinity of a processor at the moment it fails. The algorithm is fault-tolerant in the sense that no advance knowledge of the locations of the faults is needed for the path selection, but it is susceptible to nodes which fail while holding packets. The algorithm is of interest because during most steps, few processors will fail and almost all deliverable packets will be delivered. In addition, the algorithm itself is quite simple and is the first adaptive routing algorithm for which an $O(\log N)$ bound on the routing time has been achieved.

Work on adaptive routing for faulty hypercubes is potentially applicable outside the setting of fault-tolerance. Except for the algorithm we present, all known $O(\log N)$ packet step routing algorithms for the hypercube are inherently nonadaptive. Whereas $\Omega(\log N)$ packet steps are also a lower bound on the time to route (since the diameter of the hypercube is $\log N$), the implied $O(\log^2 N)$ bit step bound for $O(\log N)$-size packets is not provably optimal. Recently, we have proven a lower bound of $\Omega(\log^2 N / \log \log N)$ bit steps for all nonadaptive algorithms ([ALN]). Thus, serious improvement on the upper bound will have to come from an adaptive algorithm.

There has been other work on packet routing on faulty hypercubes. Most notably, Rabin ([R]) has devised an elegant scheme called information dispersal routing wherein each packet to be routed is decomposed into $\log N$ pieces. The pieces are routed in a randomized nonadaptive fashion to their destinations and then recombined to form the original message. A key aspect of the scheme is that the packet decomposition uses error-correcting codes. Therefore only a constant fraction of the pieces of any packet need to get through to the destination for the packet to be reconstructed.

Rabin makes different assumptions about both the nature of fault detection and the size of the packets. His model assumes no detection of nearby faults is possible. In his algorithm, each node chooses $\log N$ node-disjoint paths on which to send its pieces without regard for faults they may contain. If a packet encounters a fault, it is lost. Rabin's scheme is useful only if the original packets represent relatively long bit streams. Because routing information alone uses $\Theta(\log N)$ bits, each of the $\log N$ pieces into which a packet is divided must contain $\Omega(\log N)$ bits. Thus the original packets must have length $\Omega(\log^2 N)$. Additionally, Rabin's analysis depends on the failure rate p to be $O(1/\log^2 N)$ and allows only edge faults. At most $\Theta(N/\log N)$ edge faults can be absorbed. Under these conditions, the Rabin algorithm provides a fully fault-tolerant routing of N packets in $O(\log N)$ steps with high probability.

In section 2.3, we show how to achieve Rabin's results with a simpler algorithm and analysis. Our analysis permits both node and edge faults and requires p to be $O(1/\log N)$ so that the routing can absorb up to $\Theta(N)$ edge faults as well as $\Theta(N/\log N)$ node faults. (A similar result based on Rabin's original algorithm has recently been discovered by Giladi ([G]).) We also briefly sketch a way to potentially improve its tolerance to faults in as many as a constant fraction of components by combining the decomposition scheme with our adaptive algorithm for routing around faults.

All of chapter two represents joint work with Tom Leighton. In addition, lemma 2.7 is the result of work with Bill Aiello and Satish Rao.

2.1.2 Overview

Section 2.2 contains the $O(\log N)$ time adaptive routing algorithm. In section 2.3, we show how to improve Rabin's fault-tolerant results with a simpler algorithm.

2.2 Fast Routing Around Faults

In this section we examine the problem of routing a permutation on a faulty hypercube. We describe a variant of Valiant-Brebner routing on the hypercube that we call *offset routing*. The success of the algorithm depends on local routability, a condition of the nonfaulty processor's connectivity. We show that with a probability close to 1 a faulty hypercube remains locally routable and that if it does, the routing algorithm works with high probability.

We make several assumptions about the nature of faults and about the abilities of the network's processors. Every node and edge of the hypercube is assumed to fail independently of other components and with a constant probability $p < 1 - \sqrt[6]{1/2}$. Every node is able to detect whether a neighboring node or the link to it is faulty by simply sending a one bit message and waiting for a response. It does not matter if the node cannot detect whether the fault lies in the neighbor or the link. We make the minimal assumptions about the messages themselves. Since routing information uses $\Theta(\log N)$ bits and must accompany each message, we assume that each packet contains $\Omega(\log N)$ bits.

The idea of the offset routing algorithm is to route around the faulty components. Say a hypercube node v holds a message from some source and that the route to the destination dictates that the message be sent to its neighbor v^k across the k^{th} dimension. Further assume that the edge (v, v^k) has failed. One way to pass the message on would be to find a dimension $i \neq k$ for which all components in the path (v, v^i, v^{ik}, v^k) are nonfaulty. A picture of this path appears in figure 2-1.

Unfortunately, if some node on the path from source to destination has failed and paths like that shown in figure 2-1 are used exclusively, the message will not get through. To allow for the existence of node faults, we make sure that once we

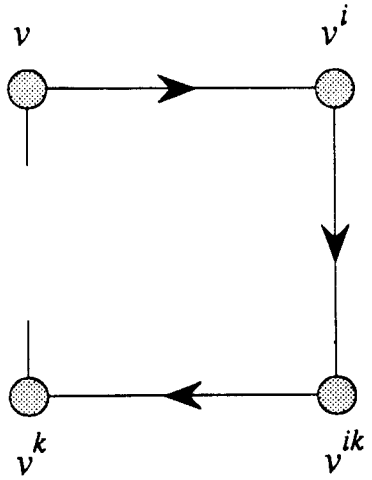

Figure 2-1: A path of length three avoiding a faulty edge.

have decided on a path from the source to the destination, the message never resides in any of the processors along the path until it reaches its destination. The path is treated as a virtual path. Instead of residing in some node v along the virtual path, the packet will reside in some neighboring node v^i. That is, it will be offset by the dimension i. If dimension k is to be traversed, some other offset j will be chosen for which the entire path $(v^i, v^{ij}, v^{ijk}, v^{jk})$ is fault-free. Thus, instead of residing in node v^k, the packet will be offset by dimension j. In this fashion, the offset path skirts around the virtual path but never meets it until the packet reaches its destination.

The offset routing algorithm uses randomness in two different ways. First, randomness is used to select virtual paths from sources to destinations. The virtual paths we will use are precisely the paths chosen by the Valiant-Brebner algorithm. Second, the offsets used along the way will be chosen from among those which create a live path of length three to the next offset node.

In section 2.2.1, we define butterflies and we review the Valiant-Brebner routing algorithm. We prove some important bounds on the number of messages the algorithm is likely to rout through small sets of edges. In section 2.2.2, we define another network, the butterfly with jump edges, which helps us to think about the offset routing algorithm on the hypercube. In section 2.2.3, we describe the offset routing

algorithm explicitly. Section 2.2.4 proves a limit of $O(\log N)$ on the length of any offset path. Finally, section 2.2.5 shows that only $O(\log N)$ other messages use any of the edges of a particular message's path. This proves that the offset routing algorithm finishes in $O(\log N)$ routing steps.

2.2.1 Valiant-Brebner Routing

The virtual paths we will use are those dictated by the Valiant-Brebner routing algorithm. Since that algorithm is viewed more intuitively as a butterfly algorithm, we will present it that way. First, we review some basic butterfly concepts. Next we present the Valiant-Brebner routing algorithm. Last, we prove two lemmas about how uniformly the algorithm uses edges. These lemmas will be useful when we examine the usage of edges by the offset routing algorithm.

The $\log N \times N$-node or $\log N$-dimensional *butterfly* is obtained from the N-node hypercube by replacing each node v of the cube by a cycle $(v_0, v_1, \ldots, v_{n-1}, v_0)$. We replace each edge (v, v^i) by a pair of edges (v_{i-1}, v_i^i) and (v_{i-1}^i, v_i) $\pmod n$. We can visualize the set of nodes $\{v_i | v \in H_n\}$ as sharing a *level* of the butterfly. We call edges of the form (v_{i-1}, v_i) *straight* edges and those of the form (v_{i-1}, v_i^i) *cross* edges. All edges connect nodes in adjacent levels (mod n).

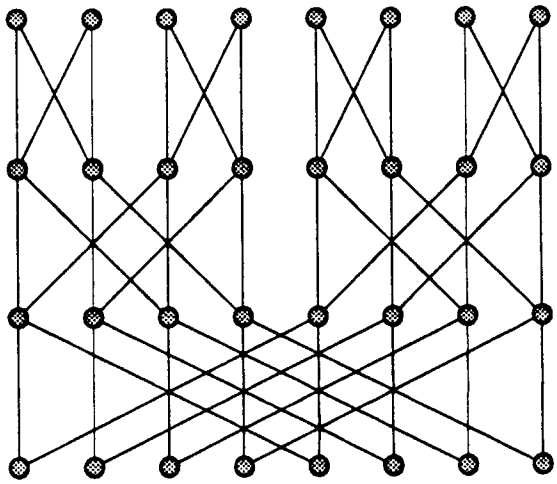

Figure 2-2: A three level butterfly. (The top and bottom rows are identified.)

All dimension i hypercube edges connect the $(i-1)^{st}$ level with the i^{th} level. Thus any hypercube algorithm which only uses one dimension during each step and only uses consecutive dimensions during consecutive steps can run on the butterfly just as quickly. Any butterfly algorithm works as well on the hypercube from which the butterfly was obtained. We may regain this hypercube by collapsing columns of the butterfly.

The Valiant-Brebner hypercube routing algorithm is also a butterfly routing algorithm. A packet starts at some node v_0 and ends at some node w_0. We think of the column of nodes $\{v_i\}$ as being shared by the hypercube node v, which assigns each node in the column a different queue from a set of n queues. If a message traverses the straight edge (v_{i-1}, v_i) in some butterfly step, then it is passed from the node v's $(i-1)^{st}$ queue to its i^{th} queue in the hypercube step. If the message traverses the cross edge (v_{i-1}, v_i^i) in some butterfly step, then it is passed from v's $(i-1)^{st}$ queue to v^i's i^{th} queue in the hypercube step.

Routing from v_0 to w_0 is simplified by the fact that there is a unique path of length n between those two nodes. The i^{th} step in the path connects a node at level $i-1$ with one at level i. If v and w agree in the i^{th} bit, the edge is a straight edge. If they differ, a cross edge is used. For example, to route from the node $(1,1,0)_0$ to the node $(0,1,1)_0$ we would use the path $(1,1,0)_0, (0,1,0)_1, (0,1,0)_2, (0,1,1)_0$.

In the first phase of the Valiant-Brebner routing algorithm, each node in level 0 first sends its packet to a random node in the same level using the unique path of length n. In the second phase, the packet is routed along the unique path to its true destination. In [VB] it was shown that this algorithm takes $O(n)$ steps to complete and uses total queue length $O(n)$ at every hypercube node, with high probability.

We will worry about congestion, or the total number of messages using a given set of edges, in the offset routing schedule. A message can congest an edge only if its virtual path brings it close to that edge. It will then congest the edge only if particular choices of offset are made. To bound the congestion, we will first bound the number of messages whose virtual paths come close to a given set of edges. We need only the following two bounds on the number of messages traversing small sets

of edges via their Valiant-Brebner paths.

Lemma 2.1. *Take an arbitrary set of h edges on one level of the n-dimensional butterfly. Then with high probability the Valiant-Brebner routing scheme routes only $O(h+n)$ messages through edges in the set.*

Proof. Note that each message can congest at most one edge in the set. The following analysis applies to the first phase of the routing algorithm. The analysis for the second phase is almost identical.

Say the edges share level l of the butterfly. Then we can partition the butterfly's first l levels into $N/2^l$ nonintersecting butterflies $B_1, B_2, \ldots, B_{N/2^l}$ each built from a subcube with 2^l nodes. For a message to route through one of the h edges, it must start in the same butterfly as the edge. Say that h_i of the edges lie in butterfly B_i. Because paths are chosen uniformly, each message is equally likely to traverse any of the edges in a level of B_i. Thus each message starting in butterfly B_i has probability $p_i = h_i/2^l$ that it will hit one of the edges in the set.

For a node v, let $X_v = 1$ if v's packet congests an edge in the set and 0 otherwise. We wish to bound the value of $X = \sum_v X_v$. To do so, we bound the moment generating function $M(\lambda) = E[e^{\lambda X}]$ for positive λ. We can then bound $Pr[X > kh] = Pr[e^{\lambda X} > e^{\lambda kh}] \leq e^{-\lambda kh} E[e^{\lambda X}]$. This bound follows directly from Markov's inequality $Pr[Y > b] \leq E[Y]/b$ for any nonnegative random variable Y and nonnegative bound b. We will first bound the moment generating functions $M_v(\lambda) = E[e^{\lambda X_v}]$ We can then use the fact that, since the X_v are independent, $M(\lambda) = \prod M_v(\lambda)$.

The moment generating function $M_v(\lambda)$ will depend on the butterfly B_i to which v belongs. If $v \in B_i$ then $M_v(\lambda) = E[e^{\lambda X_v}] = (\frac{h_i}{2^l} e^{\lambda} + 1 - \frac{h_i}{2^l})$. Precisely 2^l nodes in the butterfly share this moment generating function. Thus the moment generating function $M(\lambda)$ for X satisfies

$$\begin{aligned} M(\lambda) &= \prod_{i=1}^{N/2^l} \left(\frac{h_i}{2^l} e^{\lambda} + 1 - \frac{h_i}{2^l} \right)^{2^l} \\ &= \prod_i \left(1 + \frac{(e^{\lambda} - 1) h_i}{2^l} \right)^{2^l} \end{aligned}$$

$$\leq \prod_i e^{(e^\lambda - 1)h_i}$$
$$= e^{(e^\lambda - 1)h}$$

The inequality between lines two and three follows from the inequality $1 + x \leq e^x$ for all x.

Thus $Pr[X \geq kh] \leq e^{(e^\lambda - 1)h} e^{-kh\lambda} = (e^{e^\lambda - k\lambda - 1})^h$. Setting $\lambda = \ln k$, this implies $Pr[X \geq kh] \leq (e^{k(1-\ln k)-1})^h$, a bound which can be made as small as desired by increasing the constant k.

If $h > n$, then the probability that more than kh messages pass through the edges is less than $N^{-O(k \ln k)}$. Similarly, if $h < n$, the chance of having more than kn messages crossing the set is also less than $N^{-O(k \ln k)}$. ■

Lemma 2.2. *Take an arbitrary set of $O(n^3)$ edges in the n-dimensional butterfly. With high probability the Valiant-Brebner routing scheme routes only $O(n^3)$ messages through edges in the set, counting a message once for each time it traverses an edge in the set (i.e. counting according to multiplicity).*

Proof. We will examine each level separately and then sum across levels. Say level l has e_l edges from the set. By lemma 2.1, for any k there is a c such that there is at most an N^{-k} chance that more than $c(e_l + n)$ messages traverse the e_l edges from the set at level l. Summing over all levels, with probability at least $1 - nN^{-k}$, the number of messages crossing edges from the set at any level is no more than $c(\sum_l e_l + n^2)$. ■

2.2.2 Jump Edges

As we mentioned earlier, the second use of randomness involves evading faults which lie on the virtual path chosen by the Valiant-Brebner routing algorithm. When we route on the hypercube, we have access to many more edges out of each node than we do when we route on the butterfly. We can use these edges to route around faulty components. While bits are changed consecutively by traversing virtual paths, arbitrary bits are changed during fault avoidance. We create the butterfly with jump

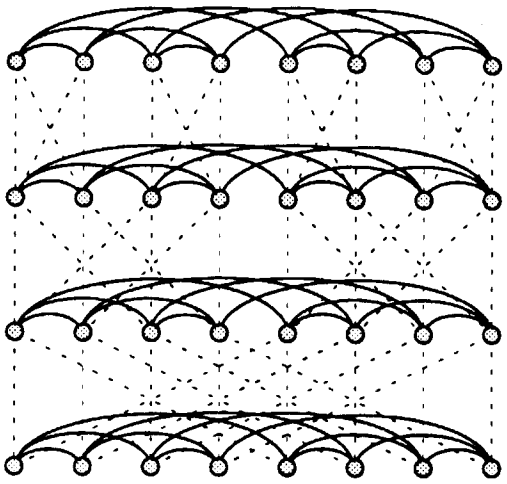

Figure 2-3: Jump edges. These edges form the hypercube connections for the nodes on each level. The dashed edges are from the underlying butterfly.

edges to accentuate the changing of adjacent bits in the virtual path while allowing for the changing of arbitrary bits in the offset path.

A *jump* edge is an edge of the type (v_j, v_j^i). Jump edges are not butterfly edges. A packet traversing such an edge would be sent (in the hypercube) from the j^{th} queue of v across the edge (v, v^i) and deposited in the j^{th} queue of v^i. Note that all n jump edges of the type (v_j, v_j^i) ,j varying, are actually manifestations of a single hypercube edge from v to v^i. This means that every hypercube edge is represented $n+2$ times in the butterfly with jump edges: as n different jump edges and 2 cross edges. Figure 2-3 depicts the jump edges for the 3×8 butterfly.

If we collapse the levels of the butterfly with jump edges, we regain the hypercube. Any algorithm we create for the butterfly with jump edges works as well on the hypercube. We need only be especially careful about congestion, or multiple packets crossing the same edge. A cross edge or jump edge traversed by a given packet is actually one out of several appearances of a hypercube edge in the butterfly with jump edges. Any congestion on another manifestation of the hypercube edge could slow the packet down. Among other things, we will concern ourself with the total congestion on a hypercube edge traversed by a packet, not just the congestion on the particular cross edge or jump edge it traverses.

2.2.3 Offset Routing

In the offset routing algorithm, each packet remains fairly close to its Valiant-Brebner path. A packet's location always differs from where their algorithm would send it by some offset which is a random dimension. The offset routing algorithm retains the two-phase structure of Valiant and Brebner's algorithm.

We first describe how packets are routed from level to level in the butterfly with jump edges. Recall that the path traversed by a packet in the Valiant-Brebner scheme is its virtual path. In the offset routing algorithm, a packet whose virtual path would pass through the $(k-1)^{st}$ level at the node v_{k-1} will pass through the level at some node v_{k-1}^i instead. If its virtual path would leave v_{k-1} via a straight edge, then the offset path will traverse three edges of the type $(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ij}, v_k^j)$. It finds such a path by randomly choosing a dimension $j \neq i$ and attempting to route across the appropriate three edges. If the packet encounters a fault in any of the three edges or the nodes along those three edges, it returns to the node v_{k-1}^i, which chooses another random dimension and tries again. Note that this means that a packet might have to traverse many more than three edges to pass from one level to the next. If the virtual path would leave v_{k-1} via a cross edge, then the offset path traverses three edges of the type $(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ijk}, v_k^{jk})$ instead. Note that no matter whether straight edges or cross edges are used in the virtual path, the node ends with a random offset j from its virtual location. If necessary, the k^{th} bit is changed to agree with the k^{th} bit of the destination. Figure 2-4 presents an offset path between adjacent levels.

Each packet must choose an initial offset to leave its source and must remove its final offset to reach its destination. To begin, the message generated by node v repeatedly chooses a random dimension j and attempts to route across the edge (v_0, v_0^j) until it successfully finds an initial offset. Say that the message reaches the 0^{th} level at the end of the second phase with offset i (i.e. it reaches the node w_0^i). Then to conclude, the message finds an offset j for which the path $(w_0^i, w_0^{ij}, w_0^j, w_0)$ is fault-free.

The offset routing algorithm combines Valiant and Brebner's strategy of changing adjacent bits with a means for avoiding faults. In our analysis, we will make fun-

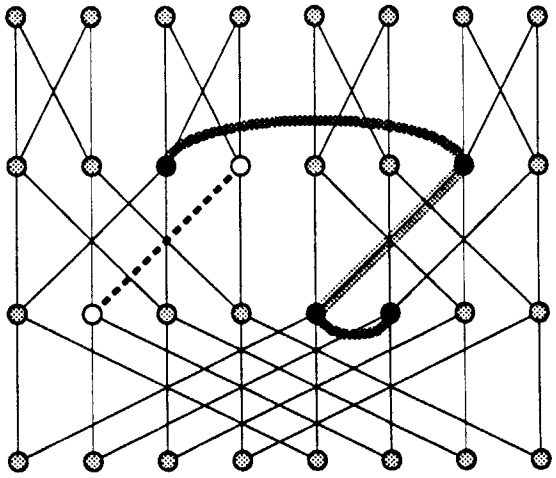

Figure 2-4: A virtual edge between adjacent levels (shown as a dashed line) and a possible offset path (shaded). In this example, $i = 1$ and $j = 3$.

damental use of the property of the distribution of virtual paths proven in lemma 2.2. The even distribution of virtual paths will help to ensure the even distribution of offset paths over the edges of the hypercube, assuming random offsets are chosen.

2.2.4 The Length of Offset Paths

If a packet is to arrive at its destination within $O(\log N)$ steps, certainly the path it takes must have length $O(\log N)$. In Valiant and Brebner's algorithm, the length of paths is fixed at $2 \log N$. Offset paths are of variable length, depending on faults encountered along the way. In this section we describe the condition of local routability. We prove that if the hypercube is locally routable, then with high probability all packets traverse offset paths of length $O(\log N)$.

Essentially, a hypercube is locally routable if every node always has ample opportunity to send a packet to the next level in the butterfly with jump edges. Consider a path $(v^i_{k-1}, v^{ij}_{k-1}, v^{ijk}_k, v^{jk}_k)$. We assume that a message has successfully arrived at v^i_{k-1} and so there are six components—three nodes and three edges—in the path that must all work properly. If the probability of failure is $p < 1 - \sqrt[6]{\frac{1}{2}}$ (about 0.11) and the faults are independent, then each such path has probability $p' = 1 - (1-p)^6 < \frac{1}{2}$

that it has a faulty component. For subsequent analysis, we would like it to be the case that for all pairs v_{k-1}, i there are at least a constant fraction of offset dimensions j which lead the message on a functioning path $(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ijk}, v_k^{jk})$. We would also like to know that at least a constant fraction of the paths $(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ij}, v_k^j)$ are fault-free for all pairs v_{k-1}, i. To begin the routing, we need that for all nodes v_0, a constant fraction of the edges (v_0, v_0^j) function properly. To end the routing, we need that for all pairs v_0, i, a constant fraction of offset dimensions j lead the message on a functioning path $(v_0^i, v_0^{ij}, v_0^j, v_0)$. Define the following sets of paths: $P_{v_{k-1},i} = \{(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ijk}, v_k^{jk}),\ j \text{ varying}\}$, $P'_{v_{k-1},i} = \{(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ij}, v_k^j),\ j \text{ varying}\}$, $Q_{v_0} = \{(v_0, v_0^j),\ j \text{ varying}\}$ and $Q_{v_0,i} = \{(v_0^i, v_0^{ij}, v_0^j, v_0),\ j \text{ varying}\}$. Fix an $\epsilon_p > 0$. If all possible sets $P_{v_{k-1},i}$, $P'_{v_{k-1},i}$, Q_{v_0} and $Q_{v_0,i}$ all have cardinality at least $\epsilon_p n$ we say the butterfly is *locally routable.*

Lemma 2.3. *Assume that the probability that any component fails is less than* $1 - \sqrt[6]{\frac{1}{2}}$ *and that all failures occur independently. Then there exists sufficiently small* $\epsilon_p > 0$ *and* $c_1 = c_1(\epsilon_p, p)$ *such that with probability* N^{-c_1} *the butterfly is locally routable.*

Proof. The set $P_{v_{k-1},i}$ of paths available at v_{k-1}^i are node-disjoint. (The same argument holds for sets of paths P', Q and Q'.) Thus the faultiness of any path is independent of other paths in the set.

The probability that fewer than $\epsilon_p n$ paths $P_{v_{k-1},i,j}$ are fault-free is

$$\sum_{i=0}^{\epsilon_p n} \binom{n}{i} (1-p')^i p'^{n-i}$$

The ratio of successive terms is $\frac{t_i}{t_{i-1}} = \frac{(n-i+1)(1-p')}{ip'}$, which is greater than and bounded away from 1 for small enough ϵ_p. Thus the sum is bounded by a constant times the last term. Let $\exp_2(x)$ denote 2^x. Then the last term $t_{\epsilon_p n}$

$$\begin{aligned}
&= \binom{n}{\epsilon_p n} (1-p')^{\epsilon_p n} p'^{(1-\epsilon_p)n} \\
&\leq \frac{(ne)^{\epsilon_p n}}{(\epsilon_p n)^{\epsilon_p n}} (1-p')^{\epsilon_p n} p'^{(1-\epsilon_p)n} \\
&= \exp_2(\epsilon_p n \log e - \epsilon_p n \log \epsilon_p + \epsilon_p n \log(1-p') - \epsilon_p n \log p' + n \log p') \\
&= \exp_2(h(\epsilon_p, p')n + n \log p')
\end{aligned}$$

We can bound this last expression using the fact that each path has a low probability of containing a failure. Since $p' < \frac{1}{2}$, $\log p' = -1 - c_2$ for some $c_2 > 0$. For $\epsilon_p = 0$, $h(\epsilon_p, p) = 0$ and the above expression equals N^{-1-c_2}. Since $h(\epsilon_p, p)$ is continuous, there is a $\epsilon_p > 0$ such that the above expression is bounded by N^{-1-c_3} with $c_3 > 0$. Since there are only $O(N \log^2 N)$ pairs v_{k-1}, i, any $c_1 < c_3$ makes the lemma true. ■

With probability N^{-c_4} for some fixed c_4, some node has only faulty neighbors. Thus we cannot strengthen lemma 2.3. For the remainder of section 2.2, we assume the butterfly is locally routable. Under this assumption, we will prove that the algorithm succeeds quickly with high probability.

Lemma 2.4. *Say a butterfly has faulty components but is locally routable. With high probability each message in the offset routing traverses a path of length $O(n)$.*

Proof. We will prove that any given message's path is of length $O(n)$ with high probability. Since there are only N messages, this will imply the lemma. Assume that at some point in its route, the packet is at the node v^i, where v is the node it would traverse in the Valiant-Brebner scheme. Assume as well that the packet is scheduled to traverse dimension k. (If the straight edge is to be used or if the packet is at the beginning or end of the route, the analysis is identical.) Then if the packet successfully chooses to jump across dimension j, the path $(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ijk}, v_k^{jk})$ must have no faults. Since the butterfly is locally routable, $\epsilon_p n$ of the possible paths to choose are fault-free. If a faulty path is chosen, the packet encounters the fault and returns to v_{k-1}^i using no more than six edges. Since a random dimension is chosen at each step, the probability that a packet takes more than $6b(2n+2)$ steps is less than the probability of at least $(b-1)(2n+2)$ heads in a sequence of $b(2n+2)$ tosses of a coin with probability ϵ_p of landing tails. This probability is less than

$$\begin{aligned} & \binom{b(2n+2)}{2n+2} (1-\epsilon_p)^{(b-1)(2n+2)} \\ < & \left(\frac{eb(2n+2)}{2n+2}\right)^{2n+2} (1-\epsilon_p)^{(b-1)(2n+2)} \\ < & \left(eb(1-\epsilon_p)^{b-1}\right)^{2n+2} \end{aligned}$$

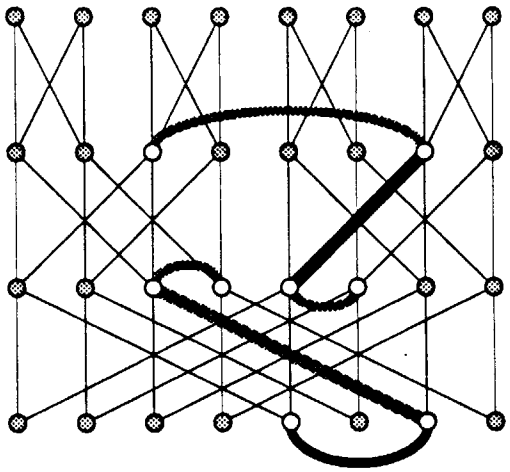

Figure 2-5: One packet might delay another packet's progress even if they never cross paths in the butterfly with jump edges. One hypercube edge is replicated $n+2$ times in the butterfly with jump edges. The two sections of paths shown here intersect in the hypercube because the darkened edges are actually one edge in the cube.

an inverse polynomial in N for large enough b. ■

2.2.5 Delay From Other Packets

Now that we know each message moves a distance of $O(\log N)$ during an offset routing phase, we need to show that its forward movement is delayed by at most $O(\log N)$ other packets. These facts together will bound the packet's time to its destination. We will show that few other packets choose virtual paths in such a way that they have a non-zero probability of selecting an offset path which congests a given node's path. We will then show that even fewer of those actually congest the path when they use offset paths.

Recall that a cross edge or jump edge traversed by a given packet is actually one out of several appearances of a hypercube edge in the butterfly with jump edges. Any congestion on another manifestation of the hypercube edge will slow the packet down. Therefore we group all $n+2$ copies of the edge together and refer to the group as one hypercube edge.

Lemma 2.5. *Consider a set E of $O(n)$ hypercube edges and butterfly straight edges. Let S be the set of butterfly edges such that any packet whose virtual path crosses an edge in S has a non-zero probability of congesting an edge in E as a butterfly edge in its offset path. Then with high probability, there are $O(n^3)$ packets whose virtual paths traverse any of the edges in S, counting a packet several times if it traverses several edges in S.*

Proof. If (w_{l-1}, w_l^l) is a butterfly edge traversed by a packet's offset path then the packet's virtual path must use an edge of the form $(w_l^{ij}, w_{l+1}^{ijl})$ for some pair i, j. There are only n^2 such pairs. The same reasoning would hold if the edge in question were a straight edge. Since $|E| = O(n)$, $|S| = O(n^3)$. By lemma 2.2, only $O(n^3)$ packets traverse edges in S, with high probability. ■

Lemma 2.6. *Let T be the set of butterfly edges such that any packet whose virtual path crosses an edge in T has a non-zero probability of congesting some edge in E as a jump edge in its offset path. Then with high probability, there are $O(n^3)$ packets whose virtual paths traverse any of the edges in T, again counting according to multiplicity.*

Proof. Say (w_{k-1}, w_{k-1}^l) is a jump edge traversed by a packet. Let $(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ijk}, v_k^{jk})$ or $(v_{k-1}^i, v_{k-1}^{ij}, v_k^{ij}, v_k^j)$ be the path used by the packet when it traverses the jump edge. Then (w, w^l) is either the first or the last edge traversed in the path. If it is the first, then $w_{k-1} = v_{k-1}^i$, $w_{k-1}^l = v_{k-1}^{ij}$ and therefore $l = j$. The edge traversed in the virtual path would have been (v_{k-1}, v_k^k) or (v_{k-1}, v_k) for some k. There are n choices for v such that $v_{k-1} = w_{k-1}^i$ and n choices for k. Thus there are only $O(n^2)$ elements of T whose traversal in some packet's virtual path gives the packet a non-zero probability of traversing the edge (w, w^l) as a jump edge. The same reasoning holds for use of the jump edge as a third edge. Again, since $|E| = O(n)$, $|T| = O(n^3)$. By lemma 2.2, only $O(n^3)$ packets traverse edges in T, with high probability. ■

Lemmas 2.5 and 2.6 also hold for the set of edges incident to the set of nodes $\{v_k\}$ for some hypercube node v. If we bound the number of packets congesting these edges then we bound the number of packets ever residing in queues in the node v (the

queue size of v).

We would like to bound the number of packets which congest the path p_0 of some message m_0. There are $O(n^3)$ packets with a nonzero probability of congesting some edge along p_0. Focus on one of these packets m_r. The packet m_r will cause congestion along the path p_0 only if an unfortunate pair of offsets i and j are chosen for it. When the packet traverses from level l_r to the level $l_r + 1$, its offset i at the l_r^{th} level is inherited from a choice made by some node in the $(l_r - 1)^{st}$ level. Then the node at level l_r chooses an offset j which will route the packet m_r to the $(l_r+1)^{st}$ level. Because any given fault can affect the routes of several different packets, offset choices made by different packets are dependent. However, the packet m_r is guaranteed by local routability to have $\sigma(r) \geq \epsilon_p n$ choices of offset i available at level $l_r - 1$. One of those offsets, say i_s, will be chosen uniformly to route the packet m_r to level l_r. Once there, some number α_{rs} of the offsets j will cause the packet m_r to cross the path p_0, if the packet is routed to level $l_r + 1$ using one of those α_{rs} offsets. Since we wish to minimize the probability that such congestion occurs too often, we are concerned that choices for i_s are made which leave too many unfortunate choices of j. By lemmas 2.5 and 2.6, we know that there exists a constant d such that with high probability $\sum_r s\alpha_{rs} \leq dn^3$. This follows because summing the α_{rs} is a second way to count virtual paths traversing edges in S and T, counting a path once for each time it traverses an edge in S or T. Finally, each α_{rs} is at most n, since there will be a total of n offsets j from which to choose. In the next technical lemma, we use these bounds on the α_{rs}, or number of unfortunate choices of offset j, to bound the number of bad choices of j's left once all packets have had the offsets i chosen for them.

Lemma 2.7. *Consider a family of nonnegative integers* $\{\alpha_{rs} | 1 \leq r \leq z, 1 \leq s \leq \sigma(r)\}$ *where* $\sigma(r) \geq \epsilon_p n$ *for all* r, $\sum_{rs} \alpha_{rs} \leq dn^3$ *and* $\alpha_{rs} \leq n$ *for all pairs* r, s. *If exactly one index* s_r *is chosen uniformly in* $[1, \sigma(r)]$ *for each index* r *then with high probability* $\sum_r \alpha_{rs_r} = O(n^2)$.

Proof. Let $X_r = \alpha_{rs_r}$. A picture of the choice of the X_r appears in figure 2-6.

We wish to bound the value of $X = \sum_r X_r$. As in lemma 2.1, we bound the moment generating function $M(\lambda) = E[e^{\lambda X}]$ and then we bound $Pr[X > bn^2] =$

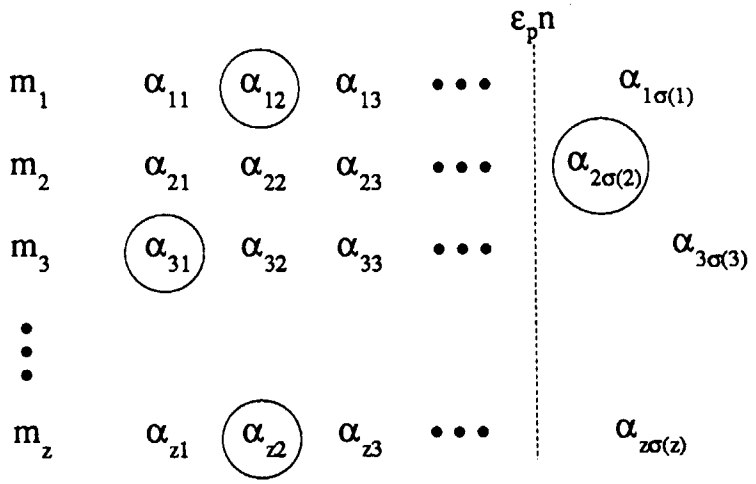

Figure 2-6: The α_{rs} and a possible choice of X_r. Each row represents the choice for some packet. The entry α_{rs} counts the number of offsets j which, if chosen in conjunction with the offset i_s, would cause the packet m_r to congest the path p_0. Circled entries represent the selections from each row.

$Pr[e^{\lambda X} > e^{\lambda bn^2}] \leq e^{-\lambda bn^2} E[e^{\lambda X}]$. As before, we will first bound the moment generating functions $M_r(\lambda) = E[e^{\lambda X_r}] = \frac{1}{\sigma(r)} \sum_{s=1}^{\sigma(r)} e^{\lambda \alpha_{rs}}$. Again, since the X_r are independent, $M(\lambda) = \prod M_r(\lambda)$.

If we could find $\alpha_{rx} \leq \alpha_{ry}$ and a positive integer δ such that $0 \leq \alpha_{rx} - \delta, \alpha_{ry} + \delta \leq n$ then by transfering δ units from the smaller α_{rx} to the larger α_{ry} we could only increase $M_r(\lambda)$ (for positive λ). This follows because $e^{\lambda \alpha_{rx}} - e^{\lambda(\alpha_{rx}-\delta)} = e^{\lambda(\alpha_{rx}-\delta)}(e^{\lambda\delta} - 1) < e^{\lambda \alpha_{ry}}(e^{\lambda\delta} - 1) = e^{\lambda(\alpha_{ry}+\delta)} - e^{\lambda \alpha_{ry}}$. The resultant change in $M_r(\lambda)$, $(e^{\lambda(\alpha_{ry}+\delta)} - e^{\lambda \alpha_{ry}}) - (e^{\lambda \alpha_{rx}} - e^{\lambda(\alpha_{rx}-\delta)})$, would be strictly positive. By this reasoning, if $A_r = \sum_s \alpha_{rs}$ is fixed, we maximize $M_r(\lambda)$ by setting all terms except possibly one equal to either 0 or n. Thus

$$E[e^{\lambda X_r}] \leq \begin{cases} \frac{1}{\sigma(r)}(e^{\lambda A_r} + \sigma(r) - 1) & \text{if } A_r < n \\ \frac{1}{\sigma(r)}(\lceil \frac{A_r}{n} \rceil e^{\lambda n} + \sigma(r) - \lceil \frac{A_r}{n} \rceil) & \text{if } A_r \geq n \end{cases}$$

For the rest of the proof we fix $\lambda = \frac{1}{n}$. If $A_r < n$ then $M_r(\frac{1}{n}) \leq \frac{1}{\sigma(r)}(e^{\frac{A_r}{n}} + \sigma(r) - 1)$ $\leq \frac{1}{\sigma(r)}(1 + \frac{2A_r}{n} + \sigma(r) - 1) \leq 1 + \frac{2A_r}{\epsilon_p n^2}$. (The second inequality uses the fact that for $0 \leq \gamma \leq 1$, $e^{\gamma} \leq 1 + 2\gamma$.) If $A_r \geq n$ then $M_r(\frac{1}{n}) \leq \frac{1}{\sigma(r)}(\frac{2A_r}{n} e + \sigma(r)) \leq 1 + \frac{2eA_r}{\epsilon_p n^2}$.

In either case the bound is at most $1+\frac{2eA_r}{\epsilon_p n^2} \leq \exp(\frac{2eA_r}{\epsilon_p n^2})$. Thus $M(\frac{1}{n}) \leq \prod_r \exp(\frac{2eA_r}{\epsilon_p n^2})$ $\leq N^{\frac{2ed}{\epsilon_p}}$. Continuing the reasoning of the first paragraph of the proof, $Pr[X > bn^2]$ $\leq e^{-bn} N^{\frac{2ed}{\epsilon_p}}$. We can make this probability an arbitrarily large negative power of N by letting b be a large constant. ■

Each packet m_r may take several attempts before reaching level $l_r + 1$ safely. On each attempt, the packet m_r may congest the path p_0. The packet always has at least a ϵ_p chance that it will make it to level $l_r + 1$ on any given attempt. Thus the number of trials it requires to succeed will be distributed somewhat like the geometric distribution with parameter ϵ_p. For ease of notation, set $\beta_r = \alpha_{r s_r}$ from the previous lemma. In each attempt by the packet m_r, there are β_r choices of offset which will produce congestion. The following technical lemma will help bound these multiple contributions.

Lemma 2.8. *Consider a family of nonnegative integers* $\{\beta_r | 1 \leq r \leq z\}$ *where* $\sum_r \beta_r = O(n^2)$ *and* $\beta_r = O(n)$ *for all* r. *Let* $\{g_r\}$ *be a set of random variables with geometric distributions* $g_r \sim G(\epsilon_p)$ *(i.e.* $g_r = a$ *with probability* $\epsilon_p(1-\epsilon_p)^{a-1}$*) Then with high probability,* $\sum_r g_r \beta_r = O(n^2)$.

Proof. Order the integers by increasing size $\beta_1 \leq \beta_2 \leq \ldots \leq \beta_z$. Then since

$$\beta_{kn+1}, \beta_{kn+2}, \cdots, \beta_{(k+1)n-1}$$

are all at least as large as β_{kn}, we know that $\sum_{k=1}^{\lfloor \frac{z}{n} \rfloor - 1} \beta_{kn} = O(n)$. We assume that $\beta_z = O(n)$, so the sum $\beta_z + \sum_{k=1}^{\lfloor \frac{z}{n} \rfloor} \beta_{kn} = O(n)$.

Now with high probability, all sums $\sum_{r=1}^{n} g_{kn+r}$ are $O(n)$. We know that

$$\sum_r g_r \beta_r \leq \sum_k (\sum_{r=1}^{n} g_{kn+r}) \beta_{(k+1)n}$$

Thus, with high probability, $\sum_r g_r \beta_r = O(n^2)$. ■

Theorem 2.9. *If we route using offset routing and the hypercube is locally routable, then with high probability, all packets are delivered in* $O(\log N)$ *steps and all nodes have total queue size* $O(\log N)$.

Proof. Focus on the path p_0 of a particular message m_0. We will show that the congestion along p_0 from various sources is $O(n)$ with high probability.

Lemmas 2.5 and 2.6 bound the number of messages which have the potential to congest an edge of m_0's path while passing between levels on their own paths. Enumerate the packets $m_1, m_2, \ldots, m_z$ which have a non-zero probability of congesting p_0 while traversing an edge from an even level to an odd level in their virtual paths. A particular packet may appear several times in the enumeration—once for each even level node along its virtual path from which it might congest an edge of p_0.

The packet m_r has at least $\epsilon_p n$ paths which would successfully route it to the next level. Arbitrarily designate exactly $\epsilon_p n$ of these paths as special. For the purposes of our analysis, we require m_r to choose a special path before we allow it to route to the next level. This can only increase the amount of congestion placed on any edge, since it increases the number of attempts made by each packet. However, once m_r does choose a special path, we always place it in the last node of the first fault-free path it found. Thus m_r winds up in the same place on the next level as if no special requirements had been made.

Consider the choice of offsets made by the message m_r at even level l_r. Let q_r be the number of choices of pairs of offset dimensions (i, j) for the message m_r which would congest an edge in m_0's path. Then $\sum q_r = O(n^3)$ by lemmas 2.5 and 2.6. (as described in the discussion immediately preceding lemma 2.7, $\sum q_r$ is a second way to count the number of edges in S and T according to multiplicity.)

The choice of the dimension i was actually made for m_r at level $l_r - 1$. The choice was made randomly and uniformly from the set of offsets which led to a fault-free path to level l_r. The exact selection of offsets i are dependent from packet to packet and, for a particular packet, from one level to the next. However, no matter how we condition on previous events, there are always enough offsets to choose from at any given moment. Also, the bounds on the probabilities of congesting p_0 will hold regardless of previous events. Let $i_1 < i_2 < \ldots < i_{\sigma(r)}$, $\sigma(r) > \epsilon_p n$, be the choices of offsets at level $l_r - 1$ which lead to a fault-free path to level l_r. Let α_{rs} equal the number of offsets j such that if m_r is routed from level $l_r - 1$ to level l_r using offset

i_s and next to level $l_r + 1$ using offset j, then congestion results in m_0's path. Then since $\sum_s \alpha_{rs} = q_r$, $\sum_{rs} \alpha_{rs} = O(n^3)$. Since the total number of offsets j is n, clearly $\alpha_{rs} \leq n$. Let i_{s_r} be the offset for m_r actually chosen at level $l_r - 1$. Lemma 2.7 implies that with high probability $\sum_r \alpha_{rs_r} = O(n^2)$. For convenience of notation, set $\beta_r = \alpha_{rs_r}$.

At level l_r, whether the message m_r chooses a path from the set of $\epsilon_p n$ special paths or the set of $(1 - \epsilon_p)n$ nonspecial paths, it has at most β_r choices which congest m_0's path. Thus whether we condition whether the choice was special or nonspecial, the probability that message m_r will congest m_0's path is bounded by $\frac{\beta_r}{\epsilon_p n}$.

Now that we have bounded the probability that the packet m_r will congest the path p_0 during one of its attempts to route to level l_r+1, we can bound the probability that too many packets actually congest p_0. The number of routing attempts made by m_r is $g_r \sim G(\epsilon_p)$. On each attempt, the probability that m_r will congest m_0's path is at most $\frac{\beta_r}{\epsilon_p n}$. Each attempt is an independent trial and the sum of the probabilities of congestion in the trials is at most $\frac{1}{\epsilon_p n} \sum g_r \beta_r$, which is $O(n)$ by lemma 2.8. By a moment generating function argument identical to that in lemma 2.1 and 2.7, with high probability $O(n)$ attempts actually did congest m_0's path. Since each attempt involves at most six edges, each attempt can add at most six to the congestion on m_0's path. Thus with high probability, the total congestion on the path from routing attempts at even levels is $O(n)$.

Next examine the congestion on p_0 from other packets beginning and ending their paths. For a packet to congest an edge as the first jump edge of its path, it has to be generated by one of the edge's endpoints. Thus there are at most $O(n)$ such packets. Now consider those packets congesting p_0 during the ending of their paths. Each of the three jump edges used to finish off a path has an endpoint which is at distance one from the virtual destination. Thus at most $O(n)$ packets exist which have the potential to congest any given edge as the first, second or third of these jump edges. Therefore a total of $O(n^2)$ packets have a non-zero probability of congesting some edge of p_0 as they finish their routes. An argument along the lines of the one bounding congestion at even levels shows that congestion from these sources is $O(n)$

as well.

The same argument bounds congestion from routing attempts at odd levels, and also bounds congestion on edges incident to any fixed node. ■

2.3 Information Dispersal Routing

The offset routing algorithm cannot tolerate faults which occur during a particular routing phase. If a packet resides in a node as it fails, that packet is irretrievably lost. Rabin ([Rab]) discovered how to use the technique of information dispersal to route even in the presence of failing nodes, provided each fault occurs with probability no more than $O(1/n^2)$.

In this section we will present a simpler variation of Rabin's algorithm. We also show how our algorithm handles faults occurring with probability $O(1/n)$. First, we will briefly sketch the main ideas of the original routing algorithm. Each packet is dispersed into n pieces sent along node-disjoint paths to different locations and then along node-disjoint paths to the final destination.

Since every piece needs to carry $\Omega(n)$ bits of routing information, the original packets must necessarily be large. For concreteness we assume that all packets contain $L = \Omega(n^2)$ bits. Any piece created will contain $O(L/n)$ bits. We also assume that all links and nodes have the capacity to hold a constant number of the original packets (and therefore $\Theta(n)$ pieces).

Rabin proves that with high probability, the number of pieces crossing any node or link never exceeds its capacity. No piece's progress is ever delayed by a full queue in the node ahead. This guarantees that each piece can move during every step and that the entire routing will take no more than $2(n+1)$ steps—$n+1$ steps for each piece to arrive at its random intermediate location and another $n+1$ to arrive at its final destination.

As Rabin points out, routing with dispersal of information can tolerate faults if the dispersal into pieces is done with more redundancy. The pieces may actually be constructed in such a way that the arrival of half (or some other constant fraction)

of them is enough to reconstruct the original message. Rabin shows how to do this through matrix multiplication. He then proves that if each link has probability $1/n^2$ of failure, then with probability $1-2N(4e/n)^{n/4}$ all messages will be safely reconstructed at their destinations.

2.3.1 Routing Along Parallel Paths

Our improvement of Rabin's results stems from a more uniform and efficient selection of paths for the routing of pieces. The n pieces are first sent sent to the neighbors of the node which generated the packet. These pieces are then routed along parallel paths to the neighbors of a random intermediate node. From there the pieces are routed along parallel paths to the neighbors of the intended destination, and from there to the destination itself. Except for the dispersal of the pieces to the neighbors of the source and the recovery of the pieces from the neighbors of the destination, the algorithm can be viewed as a butterfly algorithm. We will use the butterfly for our analysis. A picture of a set of parallel paths appears in figure 2-7.

If v and w are two hypercube nodes, let $\pi_i(v,w)$ be the path from v^i to w^i used in one phase of the Valiant-Brebner scheme. Let $\Pi(v,w) = \{\pi_i(v,w) | 1 \leq i \leq n\}$ be the set of all possible such paths. We will first show that if each node v chooses a node v' uniformly and then routes a different piece along **each** of the n paths in $\Pi(v,v')$ that only $O(n)$ pieces reside in any node's queue at any time step.

Lemma 2.10. *Consider the collection of all paths in the N sets $\Pi(v,v')$ (varying over v), where each hypercube node v has chosen a node v' randomly and uniformly. For any node u and any integer $0 \leq j \leq n$, with high probability u is the j^{th} node along only $O(n)$ paths in the collection.*

Proof. If u is the j^{th} node along the path $\pi_i(v,w)$ then $u^i = w_1 w_2 \ldots w_j v_{j+1} \ldots v_n$. Separate the two cases in which either $i \leq j$ or $i > j$. If $i \leq j$, then it must be that $v_{j+1} \ldots v_n = u_{j+1} \ldots u_n$. Precisely 2^j nodes satisfy this condition for v. If one of these nodes chooses a w such that $w_1 \ldots w_{i-1} \bar{w}_i w_{i+1} \ldots w_j = u_1 \ldots u_j$ for some $i \leq j$, then u will be the j^{th} node along exactly one path $\pi_i(v,w)$. Otherwise, u will be the

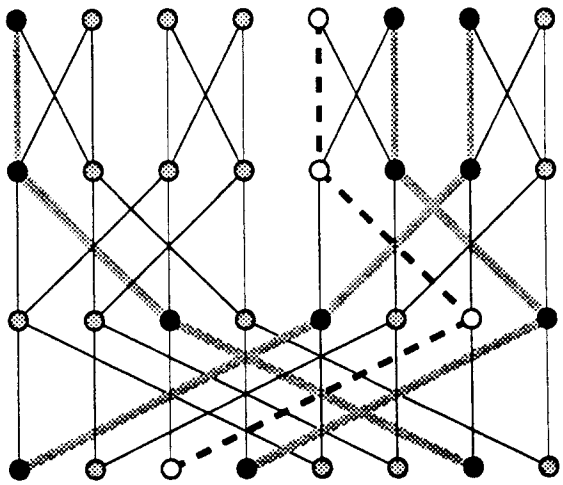

Figure 2-7: A path (dashed lines) and its adjacent set of parallel paths (shaded).

j^{th} node along none of the paths $\pi_i(v,w)$, $i \leq j$. Thus for each of the 2^j nodes, the probability of exactly one such path is $j/2^j$ and the probability of no such paths is $1 - j/2^j$.

If $i > j$, then $v_{j+1} \ldots v_{i-1}\bar{v}_i v_{i+1} \ldots v_n = u_{j+1} \ldots u_n$ for some $i > j$. Precisely $(n-j)2^j$ nodes satisfy this condition. All reasoning is the same as in the previous case, except now w must be chosen so that $w_1 \ldots w_j = u_1 \ldots u_j$. Thus the probability that u is the j^{th} node along exactly one such path is $1/2^j$. The probability that no path $\pi_i(v,w)$, $i > j$ crosses u in this fashion is $1 - 1/2^j$.

We now need only consider the sum of 2^j 0-1 random variables each with probability $\frac{j}{2^j}$ of equalling 1 and $(n-j)2^j$ 0-1 random variables each with probability $\frac{1}{2^j}$ of equalling 1. Call this sum X. Then the moment generating function $M(\lambda)$ for X satisfies $M(\lambda)$

$$\begin{aligned}
&= \left(\frac{j}{2^j}e^{\lambda} + 1 - \frac{j}{2^j}\right)^{2^j} \left(\frac{1}{2^j}e^{\lambda} + 1 - \frac{1}{2^j}\right)^{(n-j)2^j} \\
&= \left(1 + \frac{(e^{\lambda}-1)j}{2^j}\right)^{2^j} \left(1 + \frac{(e^{\lambda}-1)}{2^j}\right)^{(n-j)2^j} \\
&\leq e^{(e^{\lambda}-1)j} e^{(e^{\lambda}-1)(n-j)} \\
&= e^{n(e^{\lambda}-1)}
\end{aligned}$$

Thus $Pr[X \geq bn] \leq e^{n(e^{\lambda}-1)}e^{-bn\lambda} = (e^{e^{\lambda}-b\lambda-1})^n$. Setting $\lambda = \ln b$, this implies

$Pr[X \geq bn] \leq (e^{b(1-\ln b)-1})^n$, an inverse polynomial whose exponent can be made as small as desired by increasing the constant b. ■

The i^{th} piece created from v's packet is sent to v^i, along the path $\pi_i(v,w)$ to w^i and then to w. By lemma 2.10, at no time do more than bn pieces cross a given hypercube node, with high probability. Since the packets traversing any link all come from one of the link's endpoints, no more than $2bn$ pieces cross the link during any step of the routing. If all links and nodes have the capacity to hold $2b$ original packets, then with high probability no buffering is necessary and no piece waits in a queue.

This analysis assumes that each node routes its packet to a random destination. If we use two phases as in the Valiant-Brebner scheme, the results extend to arbitrary permutation routings:

Theorem 2.11. *If all packets are divided into n pieces which are routed along parallel paths in both phases of the routing algorithm, then for an arbitrary permutation, with high probability the two-phase routing takes $2(n+1)$ steps. No piece waits at any time.* ■

2.3.2 Fault-Tolerant Encoding of Pieces

By giving the pieces more structure, we can make the information dispersal fault-tolerant. We partition each packet F into n $O(L/n)$-bit pieces, but in such a way that if any $m = n/2$ of the pieces arrive at the destination, the original packet may be reconstructed.

Matrix multiplication is used to encode and decode the pieces. We need an $n \times m$ matrix A every m rows of which are linearly independent. We use the Hilbert matrix $A_{ij} = 1/(x_i + y_j)$, where $x_i \neq x_{i'}\ \forall i \neq i'$, $y_j \neq y_{j'}\ \forall j \neq j'$ and $x_i + y_j \neq 0\ \forall i,j$. For all these distinctness conditions to hold we need a large field. We will use the field $GF(2^s)$, with $s \approx \log\log N = \log n$.

Let A' be the matrix formed by rows $i_1, i_2, \ldots, i_m$ of A. Then

$$|A'| = \frac{\prod_{k<l}(x_{i_k} - x_{i_l})(y_k - y_l)}{\prod_{k,l}(x_{i_k} + y_l)}$$

Using this identity and Cramer's rule, we can invert any m rows of A in $O(m^2s^2)$ steps.

To take advantage of the Hilbert matrix A we block the bits of the message F into a matrix over the field $GF(2^s)$. Write $F = b_1 \ldots b_l$, where $l = L/s$ and each b_i is an s-bit byte interpreted as an element of $GF(2^s)$. Group the b_i's into l/m columns of m bytes each, and call this matrix B. Each source u computes $F_1, \ldots, F_n$ as the rows of the matrix product AB:

$$\begin{bmatrix} a_{11} & \cdots & a_{1m} \\ a_{21} & \cdots & a_{2m} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \cdot \begin{bmatrix} b_1 & b_{m+1} & \cdots & b_{l-m+1} \\ b_2 & b_{m+2} & \cdots & \vdots \\ \vdots & \vdots & & \vdots \\ b_m & b_{2m} & \cdots & b_l \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{bmatrix}$$

Given m pieces (rows F_i), the destination w can reconstruct B (i.e. the packet F) since the corresponding m rows of A are linearly independent. The destination just inverts the matrix containing those m rows.

Note that it takes $O(m^2 l/m) = O(nl)$ word operations to multiply these matrices, or $O(nls \log s)$ bit operations. The routing itself will take only $O(nls)$ bit operations, so there exists a $\log s = \log\log\log N$ gap between the complexity of the encoding and the routing stages of the protocol.

2.3.3 Fault-Tolerance via Parallel Paths

If we encode the original packet in the pieces via Rabin's matrix multiplication, then we can bound the probability that v's packet is lost by the probability that some $n/2$ of its pieces run into faulty components. But if that many pieces are lost, then at least $n/4$ are lost during one of the two phases of the routing algorithm. Assume they are lost in the first phase; the reasoning for phase 2 is identical. There are at most $(2n+3)n$ different components (nodes or links) encountered by pieces from v during the first phase. We need the following bound on the number of intersections between the routes of different pieces.

Lemma 2.12. *For any hypercube node $u \neq v, w$, no more than two paths in $\Pi(v, w)$*

cross u.

Proof. Count the nodes along the path $\pi_i(v,w)$ starting with v^i as the 0^{th} node. Say that the k^{th} node along $\pi_i(v,w)$ is the same as the l^{th} node along $\pi_j(v,w)$ for $i<j$. Then $w_1^i w_2^i \ldots w_k^i v_{k+1}^i \ldots v_n^i = w_1^j w_2^j \ldots w_l^j v_{l+1}^j \ldots v_n^j$, where $v_q^{q'} = v_q$ iff $q \neq q'$ and similarly for $w_q^{q'}$.

There are four cases. If $k,l \leq j$ then $v_j^j = v_j^i$, a contradiction. Similarly, if $k,l \geq i$ then $w_i^j = w_i^i$, a contradiction. If $k<i, l>j$ or if $l<i, k>j$ then it must be true that $w_i = \bar{v}_i$, $w_j = \bar{v}_j$ and $w_h = v_h$ for $i<h<j$. Thus all $\pi_h(v,w)$ with $i<h<j$ are precluded from crossing u (otherwise $w_h = \bar{v}_h$, a contradiction). Therefore three paths cannot all cross u. ■

Since no component's failure will affect more than two pieces, it must be true that at least $n/8$ of the $(2n+3)n$ components have failed. Only in a small fraction of fault patterns will so many failures occur in such a small set of components.

Theorem 2.13. *For any constant $k>0$ there is a sufficiently large constant $b>0$ such that if each component of the hypercube fails independently with probability $1/bn$ before or during some permutation routing, then with probability $1-N^{-k}$ the routing will be successfully completed. That is, a given packet will arrive at its destination iff both its origin and destination do not fail.*

Proof. Whether or not the i^{th} component fails gives rise to a 0-1 random variable whose moment generating function is $M_i(\lambda) = (\frac{1}{bn}e^{\lambda} + (1-\frac{1}{bn}))$. The moment generating function for the sum of these random variables is $M(\lambda)$

$$\begin{aligned} &\leq \left(1+\frac{e^{\lambda}-1}{bn}\right)^{(2n+3)n} \\ &\leq e^{\frac{(e^{\lambda}-1)(2n+3)}{b}} \end{aligned}$$

Thus we can bound the probability that more than $n/8$ of the components fail by $\exp(\frac{(e^{\lambda}-1)(2n+3)}{b} - \frac{\lambda n}{8})$. Setting $\lambda = \ln\frac{b}{16}$, we see that the probability of so many failures is no more than $(e^{\frac{1}{8}}(16/b)^{\frac{1}{8}})^n$. The exponent of this inverse polynomial in N can be made as low as desired by increasing the constant b. ■

2.4 Remarks

Offset routing and information dispersal are complementary techniques. By combining this simplified variant of information dispersal with offset routing, still better results are possible. The combined routing algorithm tolerates the failure of a constant fraction of the hypercube's components during the course of the routing of a single permutation. To send a packet, the node first disperses pieces to a well defined set of n nodes at distance three (instead of neighbors). The packets are then routed along parallel offset paths to the symmetric set of n nodes close to the destination. Finally, the pieces are combined at the destination. If each node or link fails independently of other components and if in the case it fails it does so at a random time during the routing then this combined algorithm tolerates the failure of a constant fraction of the hypercube's components.

Chapter 3

Reconfiguration in the Presence of Faults

3.1 Introduction

In this chapter, we continue our investigation of the tolerance of the hypercube to randomly distributed faults. The techniques we develop assume a long-run view. Given that faults have accumulated in a hypercube over time, each component independently faulty with probability p, we would like to be able to program the machine while ignoring whatever faults exist. We show how to use the functioning parts of a hypercube with faults to simulate a hypercube without faults at a surprisingly low cost. More precisely, we show how to embed a hypercube in the functioning part of a hypercube with faults so that features such as locality are preserved.

Before we can state our results formally and assess their value, we first must describe the constraints, assumptions and objectives of network reconfiguration and/or simulation in the presence of faults. We divide the discussion into six general topic areas: preservation of locality, load balancing, message traffic, simulation overhead, algorithms for implementation, and modelling of faults.

An *embedding* of a network G_1 into another network G_2 is a map $\phi : G_1 \mapsto G_2$ that maps each node of G_1 to a node of G_2 and each edge of G_1 to a path in G_2 between the images of its endpoints.

We call the pattern of faults F. That is, we include each component of H_n in F independently and with probability p. Therefore the functional part of the hypercube is $H_n - F$. An embedding of H_m into $H_n - F$ is a map $\phi : H_m \mapsto H_n - F$ that maps nodes of H_m to *functioning nodes* of H_n and edges of H_m to *functioning paths* of H_n. The precise definition of functioning nodes and paths will vary, although the general interpretation is straightforward.

Preservation of locality. Because communication in hypercube-based machines is mostly local, and because communication is a dominant factor in measuring the performance of a parallel machine, it is crucial that a good embedding of H_m in $H_n - F$ preserve locality. In other words, neighboring processors in H_m should be mapped to nearby processors in $H_n - F$. In order to quantify this notion, we say that an embedding has *dilation* D if for each edge e in H_m, the path $\phi(e)$ has length at most D in $H_n - F$. Of course, it is most desirable to find embeddings with small dilation. At the very least, the dilation of an embedding ϕ is a lower bound on the time required for $H_n - F$ to simulate a single step of H_m if the computation of each node $v \in H_m$ is performed by $\phi(v)$ in $H_n - F$.

The notion of dilation can also be extended to paths. We will describe natural embeddings of H_{n-1} in $H_n - F$ for which nodes separated by distance d in H_{n-1} are mapped to nodes separated by distance $d + 2$ in $H_n - F$. These embeddings have dilation 3.

Balancing the load. We will consider embeddings which allow several nodes of H_m to be mapped to a single node of $H_n - F$. Mappings that are one-to-one are the most desirable since then each processor of $H_n - F$ only has to simulate the action of a single processor of H_m. In general, we define the *max load* of an embedding to be the maximum number of processors of H_m mapped to any single node of $H_n - F$. One algorithm we describe discovers embeddings with max load 1 (i.e. one-to-one mappings) while the other finds embeddings with constant max load.

In addition to having small max load, it is desirable to use as many of the functioning cells of $H_n - F$ as possible. The use of live cells is partly described by the max load. To further characterize this quantity, we define the *expansion* of an em-

bedding to be the ratio of the size of the largest one-to-one hypercube we could hope to embed in $H_n - F$ to the size of the hypercube that we do embed. Since the size of a hypercube is always a power of two, the expansion is

$$2^{\lfloor \log (1-p)2^n - m \rfloor} = 2^{\lfloor n - m - \log \frac{1}{1-p} \rfloor}$$

We focus on embeddings of H_{n-1} in $H_n - F$ for $p < \frac{1}{2}$. Such embeddings have expansion one, which is the best possible.

Message traffic. In addition to balancing the processing load among the functioning processors, it is desirable to balance the message routing load among the wires. In particular, it would not be good if many paths $\{\phi(e)|e \in H_m\}$ traversed a single wire of $H_n - F$ since local communication along these paths would require the use of the same wire. To formalize this notion we say that an embedding has *congestion* C if every edge of $H_n - F$ is contained in at most C paths of $\{\phi(e)|e \in H_m\}$. We consider embeddings with congestion as much as $\Theta(\log N)$ and as little as $O(1)$.

Congestion is a lower bound on the time required for the functioning part of H_n to simulate H_m if messages traversing e in H_m are routed along $\phi(e)$ in $H_n - F$. For some specific applications, however, we can do better. For example, hypercubes are often used to simulate bounded-degree networks such as arrays and trees. In such applications, only a constant number of wires incident to any node are used in any parallel step of H_m. Hence, the effective congestion in the corresponding embedding may be much less than it seems at first. To capture this notion, we say that an embedding has *induced congestion* I if every edge of $H_n - F$ is contained in at most I paths of $\{\phi(e)|e \in H_m\}$ for which the edges $e \in H_m$ are node-disjoint. The two main algorithms in this chapter find embeddings with constant induced congestion. Such embeddings are particularly useful for simulating trees, arrays, normal hypercube algorithms and other structures with bounded processor degrees.

Simulation overhead. One obvious use of a hypercube with faults is to simulate a hypercube without faults. This can always be done given enough slowdown and duplication of resources, but the goal is to make the simulation as efficient as possible. The key factors influencing the efficiency of the simulation are dilation, max load and

congestion. By achieving good bounds on these values, we show that any step of a hypercube H_n can be simulated by $O(1)$ steps of $H_n - F$. In addition, we use the notion of induced congestion to show that a hypercube with faults can simulate trees, arrays and other bounded-degree networks of the same size with only constant slowdown.

Algorithms for implementation. In addition to proving that there is an efficient embedding of H_m in $H_n - F$, it is desirable to develop an efficient algorithm for finding the embedding. Ideally, the algorithm would be deterministic, fast, easy to implement, and decentralized (i.e., using only local control). In fact, we describe such algorithms in sections 3.2 and 3.3 We also describe a fast, local probabilistic algorithm in section 3.5.

Modelling of faults. In general, we might consider three types of faults in H_n. The most serious fault would be one that completely destroys a node and all wires incident to it. We call such faults *total.* A less serious fault would be one that destroys just the computational portion of a node, and leaves the communication (i.e. switching or routing) portion of the node intact as well as the incident wires. We call such faults *partial.* (Note that it does not make sense to consider a fault that destroys just the communication portion of the node. The computation portion would then also be useless since it would be disconnected from the rest of the network.) Last, faults could occur in individual wires.

In our model, no malicious faults occur. Any node can determine if a neighboring node or link has failed by probing the link in $O(1)$ time.

Along with the type of fault, the distribution of faults must also be specified. As with routing in chapter two, we consider a model in which faults occur independently among components with probability p. We restrict our attention to the situation where $p < \frac{1}{2}$, although the methods can be extended for larger p. In addition, we consider the case in which the number of faults is smaller than a constant fraction of the total number of nodes. The assumption concerning independence of faults is crucial to our analysis, but the methods can also be applied in a hierarchical setting where entire subcubes of nodes may fail at once. Such extensions might be useful in

a practical setting where the actual machine may consist of a collection of boards, each of which consists of a collection of chips, and so on.

The material we present is philosophically related to previous work in fault tolerance of arrays in the context of wafer-scale integration ([Gr],[GG],[LL1],[LL2]), although the techniques and results are quite different. For example, constant dilation reconfiguration is not possible for arrays and trees. There has been relatively little previous work on the fault-tolerant reconfiguration of hypercubes (to our knowledge). An exception is the work of Becker and Simon ([BS]), who consider fault-free subcubes of a hypercube containing worst case faults. The constraint that the embedded cube be a subcube (i.e., dilation 1) is very restrictive, as is the assumption that faults are located in a worst case fashion. Hence, the techniques and results of [BS] are quite different from those presented here. Another exception is the work of Dolev, Halpern, Simons and Strong ([DHSS]) who also study worst case bounds. Their model of communication also differs from ours in that they assume that after the faults occur, the new connections must be chosen from a predetermined set of routings.

3.1.1 Summary of Results

At first, we consider an N-node hypercube containing random partial processor faults. We describe algorithm 3.1, an algorithm for embedding an $N/2$-node hypercube in the functioning processors.

Theorem 3.3. *Algorithm 3.1 is a local, deterministic $O(\log N)$ step algorithm. If the nodes of H_n fail independently and partially with probability $p \leq \frac{1}{4}$ then with probability at least 15/16 algorithm 3.1 constructs a one-to-one embedding of H_{n-1} into $H_n - F$ with dilation 3, congestion $2 \log N$, and induced congestion 2.*

Next we improve this algorithm so that it embeds an $N/2$-node hypercube in the functioning processors with the same performance with probability $1 - N^{-c_{11}}$ provided that processors are faulty with probability $p < 1/2$, for sufficiently large N. The algorithm for finding the embedding is deterministic, easy to implement, runs in $O(\log N)$ parallel steps, and uses only local control. As a result, we extend the results

of Bhatt, Chung, Leighton and Rosenberg [BCLR] and others to be fault-tolerant. In particular, we show that a hypercube with partial processor faults can simulate any binary tree or mesh of the same size with only constant factor slowdown. The most surprising (and potentially most useful) feature of the embedding is the degree to which it preserves locality.

Next, we extend the results to handle total faults. We describe an embedding for which the dilation is 7, the max load is 1, the congestion is $O(\log N)$, and the induced congestion is $O(\log N/\log\log N)$ with high probability (for sufficiently large N). The algorithm for achieving these results is probabilistic, runs in $O(\log N)$ steps, and uses only local control.

Finally, we address the issue of congestion. We demonstrate a probabilistic algorithm which with high probability finds an embedding in a hypercube containing totally faulty processors for which the dilation is 5, the max load is $O(1)$, and the congestion is $O(1)$.

Theorem 3.22. *For each $p < 1-\sqrt[4]{.5}$ (about .16) there is an $O(\log N)$ step algorithm such that if each of the nodes of an N-node hypercube fails with probability p then with probability $1-N^{-c_{15}}$ the algorithm finds an embedded fully functioning N-node cube with constant load, dilation and congestion. The paths which simulate the edges of the cube only use live nodes.*

As a consequence, a faulty hypercube can simulate a functioning hypercube of the same size with constant delay.

These last two algorithms actually work in a semi-worst case setting. As long as a constant fraction of each node's neighbors remain alive and a constant fraction of a specified set of paths for each node have no faults along them, the good embeddings exist.

Chapter three is the result of joint work with Johan Hastad and Tom Leighton.

3.1.2 Overview

In section 3.2, we consider only partial faults which occur with probability $p \leq 1/4$. We extend the algorithm to handle failure probabilities up to $1/2$ in section 3.3. A probabilistic algorithm for reconfiguring with total faults appears in section 3.4 and section 3.5 contains a probabilistic algorithm achieving constant delay reconfiguration with total faults. In section 3.6 we describe a way to implement the algorithms of section 3.5 so that they run in $O(\log^2 N \log\log N)$ time. In section 3.7 we extend our results to the cases where the probability of failure is very low and also to the case where edge faults occur.

3.2 Embeddings for Small p with Dilation 3

In this section we consider the less severe model of partial faults where it is possible to use the faulty processors as switches and to route through them. We assume that the probability that any given processor fails is less than or equal to $1/4$ and we present an algorithm which with probability $15/16$ constructs a one-to-one embedding of H_{n-1} in H_n with dilation 3, congestion $2n$ $(= 2\log N)$ and induced congestion 2.

3.2.1 Mapping Dead Nodes to Live Nodes

Let $\tilde{H}_{n-1}$ be the subhypercube on $N/2 = 2^{n-1}$ nodes induced by the nodes with first coordinate zero. For each node v in H_n, let v' be the node with first coordinate different from v's whose coordinates otherwise agree with those of v (i.e. v's neighbor across the first dimension). Also, for a node $y = (y_1, y_2, \ldots, y_{n-1})$ in H_{n-1}, let $\tilde{y}$ be the node in H_n with coordinates $(0, y_1, y_2, \ldots, y_{n-1})$.

Given some pattern of failure for the nodes in H_n, say a node $v \in \tilde{H}_{n-1}$ is *rich* if both v and v' are live and *poor* if both v and v' are dead. If every node in $\tilde{H}_{n-1}$ were not poor, we could easily embed H_{n-1} in $H_n - F$ by mapping $(y_1, \ldots, y_{n-1})$ to whichever of $\{(0, y_1, \ldots, y_{n-1}), (1, y_1, \ldots, y_{n-1})\}$ were alive. Unfortunately, there will be a constant fraction of poor nodes in $\tilde{H}_{n-1}$ with very high probability since each

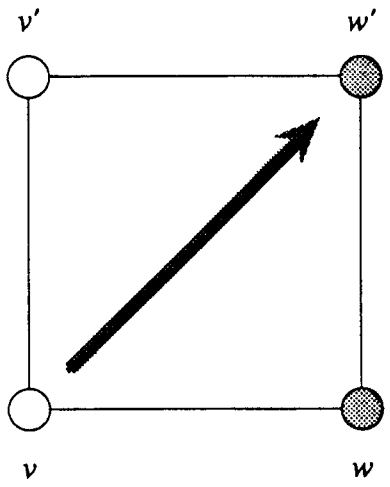

Figure 3-1: Borrowing from a neighbor. Dead nodes are shown as white. The arrow points from the simulated node to the simulating node.

node in $\tilde{H}_{n-1}$ is poor with probability p^2.[1]

We handle the existence of poor nodes by mapping each poor node v to a neighboring rich node w. At most one v is mapped to a given node w. Hence v can borrow w'. In this fashion we will be able to embed H_{n-1} in $H_n - F$ much as if there were no poor nodes at all. At step k algorithm 3.1, shown in figure 3-2, will attempt to assign v to v^{k+1}, if v and v^{k+1} are as yet unassigned.

for $k \leftarrow 2$ to n for all nodes v
  if v is poor and unassigned and v^k is rich and unassigned assign v to v^k

Figure 3-2: Algorithm 3.1.

3.2.2 Analysis of the Borrowing

If processors fail with probability less than or equal to 1/4, algorithm 3.1 will construct an embedding with probability at least 15/16. We show this by proving two simple

[1] We use the phrase Q is more than $\Omega(g)$ with very high probability to mean "There exist constants k and d independent of N such that the probability that Q does not exceed dg is less than 2^{-kN}."

lemmas. First we will prove a lemma which will be of crucial importance to the later analysis.

Lemma 3.1. *At step $k-1$, what has happened to v is independent of what has happened to any node which differs from v in at least one of the last $n-k$ coordinates.*

Proof. At step i, nodes that affect each other have all coordinates identical except the ith coordinate. Thus at step $k-1$, if we divide the nodes into groups each having identical last $n-k$ coordinates, all previous communication has taken place within the groups. Thus nodes in different groups cannot affect each other. ■

Lemma 3.2. *The probability that a given node v is poor and unassigned after the i^{th} step is at most $(2p)^i p^2$.*

Proof. For each node v let $p_i = \Pr[v \text{ is poor and unassigned after step } i]$ and $q_i = \Pr[v \text{ is rich and unassigned after step } i]$. Then $p_0 = p^2$ and $q_0 = (1-p)^2$. A node v will be poor and unassigned after step $i+1$ if and only if it was poor and unassigned after step i and the node it requested in step $i+1$ was not rich and unassigned. A similar statement holds for whether a node is rich and unassigned after step $i+1$. Thus, since these probabilities are independent,

$$q_{i+1} = q_i(1-p_i)$$
$$p_{i+1} = p_i(1-q_i)$$

Subtracting the two equations yields $q_{i+1} - p_{i+1} = q_i - p_i$, which is natural since the surplus of rich nodes over poor nodes is constant. Thus the difference is $q_i - p_i = q_0 - p_0 = 1-2p$, or $q_i = 1-2p+p_i$. Therefore $p_{i+1} = p_i(2p-p_i) \leq (2p)p_i$ and so $p_i \leq (2p)^i p_0 = (2p)^i p^2$. ■

The probability that an individual node is poor and unassigned at the end of the algorithm is less than $(1/2)^{n-1}(1/16) = 1/8N$. Thus the probability that some node is poor and unassigned is no more than $(N/2)(1/8N) = 1/16$.

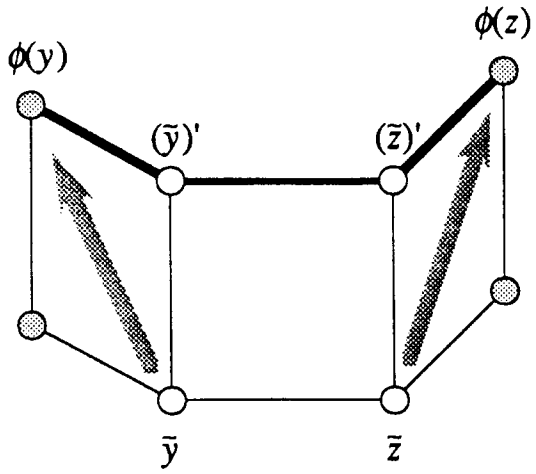

Figure 3-3: Mapping an edge to a path. The heavy edges form the path chosen to simulate the edge between the two simulated nodes at bottom.

3.2.3 Embedding Edges

If the algorithm successfully assigns each poor node to a rich node call the assignment ψ. Embed H_{n-1} in H_n with the embedding ϕ which maps nodes in H_{n-1} to nodes in H_n by

$$\phi : y \mapsto \begin{cases} \tilde{y} & \text{if } \tilde{y} \in H_n - F \\ (\tilde{y})' & \text{if } (\tilde{y})' \in H_n - F \text{ but } \tilde{y} \in F \\ \psi(\tilde{y})' & \text{otherwise} \end{cases}$$

and maps edges in H_{n-1} to paths in H_n by

$$\phi : (y,z) \mapsto \begin{cases} (\phi(y), \phi(z)) & \text{if } \phi(y) = \tilde{y}, \phi(z) = \tilde{z} \\ (\phi(y), \phi(y)', \tilde{y}, \phi(z)) & \text{if } \phi(y) \neq \tilde{y}, \phi(z) = \tilde{z} \\ (\phi(y), (\tilde{y})', (\tilde{z})', \phi(z)) & \text{if } \phi(y) \neq \tilde{y}, \phi(z) \neq \tilde{z} \end{cases}$$

Although the mapping of the edges looks complicated, every edge simply maps to a shortest path between the corresponding nodes. Figure 3-3 depicts an instance of the third possibility. Since in all cases the length of the path is at most 3 the embedding has dilation 3.

To check the congestion, observe that a given edge is used on a given path $\phi((y,z))$ only when one of its endpoints is $\phi(y)$, $\tilde{y}$, or $(\tilde{z})'$. Checking cases shows that no edge

could lie on three paths corresponding to node-disjoint edges. Thus, since we can partition the edges of the hypercube into n matchings, the congestion can be no worse than $2n$. By this argument it also follows that the induced congestion of the embedding is at most 2.

Theorem 3.3. *Algorithm 3.1 is a local, deterministic $O(\log N)$ step algorithm. If the nodes of H_n fail independently and partially with probability $p \leq 1/4$ then with probability at least $15/16$ algorithm 3.1 constructs a one-to-one embedding of H_{n-1} into $H_n - F$ with dilation 3, congestion $2 \log N$, and induced congestion 2.*

3.3 Embeddings with Dilation 3 for $p < 1/2$

In this section we extend the algorithm of section 2 so that it can handle independent faults with probabilities exceeding $1/4$ but less than $1/2$. This is best possible in the sense that if $p \geq 1/2$, then more than half of the nodes will fail with probability at least $1/2$. In that case it would be impossible to achieve a one-to-one embedding of H_{n-1} in $H_n - F$.

Call a node $v \in \tilde{H}_{n-1}$ a *topnode* if v is dead but v' is alive. We now handle the existence of poor nodes by mapping each poor node v to a neighboring node w which is either rich or a topnode. If w is a topnode, we make sure that w has a rich neighbor u so that w can borrow u'. We call this process *pushing* a topnode.

Algorithm 3.2, shown in figure 3-5, will carry out the program outlined above in 4 stages. The only additional feature is that poor nodes without enough topnode neighbors will be treated separately.

Observe that conflicts can only occur during stage 4, and can be easily resolved by having the node with lower index win.

Lemma 3.4. *Assume that nodes fail independently with probability $p < 1/2$ and that N is sufficiently large. Then there is a constant $c_5 > 0$ such that after algorithm 3.2 terminates, with probability $1 - N^{-c_5}$: (i) every poor node is assigned to a neighbor which is either a topnode or a rich node, and (ii) every topnode which has been assigned to a poor node is pushed to a rich neighbor.*

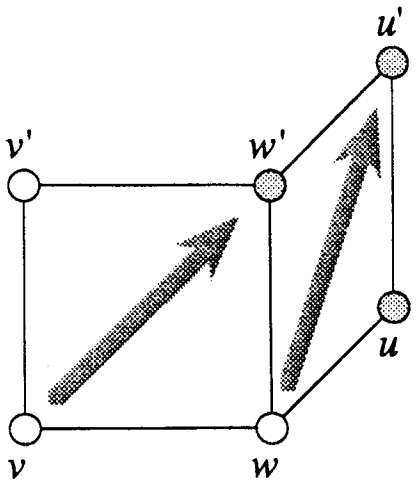

Figure 3-4: Pushing a topnode. Dead nodes are shown as white. The arrows point from the simulated nodes to the simulating nodes.

3.3.1 Analyzing Stages 1 and 2

Stages 1 and 2 comprise a "first pass" to assign poor nodes. In stage 1, those poor nodes with few top node neighbors are given first crack at assignments, since these nodes will have much less ability to push neighboring topnodes later in stage 4. Stage 2 replicates algorithm 3.1. We expect that the vast majority of nodes will find assignments during this stage.

Let ϵ be a small positive constant depending on p and let $d = d(\epsilon, p)$ and $c = c(\epsilon, p)$ be suitable positive constants depending only on ϵ and p. Throughout the argument we will assume that N is sufficiently large. The *neighborhood* of a point is the set of points at distance 1 from the point, and a *sphere* denotes a sphere in the Hamming metric.

Lemma 3.5. *For $\epsilon < p(1-p)/4\sqrt{2}$ there is a positive integer constant $d = d(\epsilon, p)$ such that for sufficiently large N, with probability $1 - 1/N$ no sphere of radius 6 contains more than d nodes processed in stage 1.*

Proof. Take any sphere of radius 6 and any d points in this sphere. The union of their neighborhoods is of size at least $dn - d^2 > dn/2$. ($N = 2^n$ is assumed to be sufficiently large and any two neighborhoods do not have more than 2 points in

Stage 1:
for every poor node v which has fewer than ϵn topnodes as neighbors across dimensions $> \epsilon n$ do
for $k \leftarrow 2$ to n
if v^k is rich and unmarked, mark it v
Stage 2:
for $k \leftarrow 2$ to ϵn for all nodes v
if v is poor and unassigned and v^k is rich and unassigned assign v to v^k
Stage 3:
for every node v which was processed in stage 1
assign v to the node which was marked v
for every node w assigned to a marked node during stage 2
w becomes unassigned
Stage 4:
for all unassigned poor nodes v do
for $k \leftarrow \epsilon n + 1$ to n
if v^k is an unpushed topnode and there is an unassigned rich node v^{kj} for some $j > \epsilon \log N$
assign v to v^k and push v^k to v^{kj}

Figure 3-5: Algorithm 3.2.

common). By assumption, at most $2d\epsilon n$ nodes in this neighbor set can be topnodes. Since the probability that an individual node is a topnode is $p(1-p)$ the probability of having exactly i topnodes in a set of size $dn/2$ is

$$p_i = \binom{\frac{dn}{2}}{i} (p(1-p))^i (1-p(1-p))^{\frac{dn}{2}-i}$$

Observe that $p_{i+1}/p_i \geq \sqrt{2}$ for $i \leq p(1-p)dn/2\sqrt{2}$. Using this and the fact that any p_i is less than 1 we get

$$\begin{aligned}
\sum_{i=0}^{2d\epsilon n} p_i &\leq \sum_{i=0}^{2d\epsilon n} p_{2d\epsilon n} 2^{\frac{i-2d\epsilon n}{2}} \\
&\leq 4 p_{2d\epsilon n} \\
&\leq 4\sqrt{2}^{2d\epsilon n - \frac{p(1-p)dn}{2\sqrt{2}}} p_{\frac{p(1-p)dn}{2\sqrt{2}}} \\
&\leq 4\exp(-c_6 dn)
\end{aligned}$$

The probability that fewer than $2d\epsilon n$ nodes in a set are topnodes decreases as we make the set larger than $dn/2$. Thus this bound holds no matter what the actual size

of the union of the neighborhoods may be. Since there are at most N possible spheres and at most $\binom{n^\delta}{d}$ ways of choosing d points, the lemma follows for large enough d. ■

Next we show that stage 1 has a good probability of success for the nodes to which it is applied.

Lemma 3.6. *The probability that there exists a node which has fewer than $2\epsilon n$ neighbors across dimensions greater than ϵn which are either rich or topnodes is bounded by N^{-c_7} for sufficiently small $\epsilon > 0$ and $c_7 > 0$.*

Proof. This proof uses reasoning similar to that of the proof of lemma 2.3. Choose $\epsilon' > 0$ such that $(1-\epsilon')\log p = -1 - c_8$ for $c_8 > 0$. The probability of having fewer than $\delta \log N$ neighbors across dimensions greater than $\epsilon' \log N$ which are either rich or topnodes is

$$\sum_{i=0}^{\delta \log N} \binom{(1-\epsilon')\log N}{i} (1-p)^i p^{(1-\epsilon')\log N - i}.$$

We can compare consecutive terms to show that this sum is bounded by a constant times the last term. The last term is

$$\begin{aligned}
& \binom{(1-\epsilon')\log N}{\delta \log N} (1-p)^{\delta \log N} p^{(1-\epsilon'-\delta)\log N} \\
\leq\ & \frac{((1-\epsilon')\log N)^{\delta \log N}}{(\frac{\delta \log N}{e})^{\delta \log N}} (1-p)^{\delta \log N} p^{(1-\epsilon'-\delta)\log N} \\
=\ & \exp_2(\delta \log e(1-\epsilon')\log N - \delta \log \delta \log N \\
& + \delta \log N \log(1-p) - \delta \log N \log p + (1-\epsilon')\log N \log p) \\
=\ & \exp_2(h(\delta,p)\log N + (1-\epsilon')\log N \log p)
\end{aligned}$$

For $\delta = 0$, $h(\delta,p) = 0$ and the above expression is N^{-1-c_8}. Since $h(\delta,p)$ is continuous, there is a $\delta > 0$ such that the above expression is bounded by N^{-1-c_7} with $c_7 > 0$. Finally set $\epsilon = \min\{\epsilon', \delta/2\}$ and observe that decreasing ϵ' to ϵ can only decrease the probability of having at most $\delta \log N$ topnode neighbors across dimensions $\geq \epsilon' \log N$. ■

By lemma 3.6, with probability $1 - N^{-c_7}$ any node processed in stage 1 must have at least ϵn rich neighbors. The only way it could fail to mark one of these is if they were all marked by other nodes. This is impossible since by lemma 3.5 there is only

a constant number of nodes processed during stage 1 within distance 6. Thus each node participating in stage 1 successfully marks a rich node.

Let us next analyze stage 2. Note that stage 2 is independent of stage 1. By lemma 3.2, after stage 2 the probability that an individual node is poor and unassigned is $< N^{-c_9}$ while the probability that it is rich and unassigned is $> (1-2p)$.

Let $\alpha \in \{0,1\}^{\epsilon n}$ and let H_α be the $(1-\epsilon)n$ dimensional hypercube which has the i^{th} coordinate α_i, $i = 1, \ldots, \epsilon n$. Observe that by lemma 3.1 the status of nodes in an individual hypercube are independent during stage 2.

Lemma 3.7. *There is a constant $d = d(p)$ such that the probability that there is a sphere of radius 4 in any H_α which contains more than d unassigned poor nodes after stage 2 is less than $1/N$.*

Proof. There are N ways to choose a sphere over all H_α's and at most $\binom{n^4}{d}$ ways of choosing d points in each sphere. The probability that these d nodes are poor and unassigned is $N^{-dc''}$ and the lemma follows for sufficiently large d. ■

3.3.2 Analyzing Stages 3 and 4

Stages 3 and 4 are responsible for assigning those nodes which remain unassigned after stage 2. In stage 3, nodes assigned in stage 1 negate some of stage 2's assignments. Bumped nodes find new assignments in stage 4.

Lemma 3.8. *With high probability, after stage 3 there are only $2d$ unassigned poor nodes in any sphere of radius 4 in any H_α.*

Proof. This follows from lemmas 3.5 and 3.7. The only additional unassigned poor nodes come from the nodes whose assignments are stolen in stage 3. But since the thief is at distance two, lemma 3.5 bounds the number of poor nodes subject to theft in any sphere of radius 4. ■

To prove lemma 3.4 observe finally that stage 4 only works inside an individual H_α. Fix a poor unassigned node at the beginning of stage 4. It has ϵn topnode neighbors. The probability that each individual topnode neighbor does not have ϵn

rich unassigned neighbors is $N^{-c_{10}}$. Thus with probability $1 - \frac{1}{N}$ each unassigned poor node has $\frac{\epsilon}{2}n$ topnode neighbors with ϵn unassigned rich neighbors each. Stage 3 reduces the number of rich neighbors to each topnode only by a constant. By lemma 3.8 we know that during stage 4 only $2d$ other unassigned poor nodes can interfere. Therefore stage 4 is successful and lemma 3.4 follows. ■

If the algorithm successfully assigns each poor node to a rich node or topnode and each pushed topnode to a rich node, call the assignment ψ. Embed H_{n-1} in $H_n - F$ with the embedding ϕ which maps nodes in H_{n-1} to nodes in $H_n - F$ by

$$\phi : y \mapsto \begin{cases} \tilde{y} & \text{if } \tilde{y} \in H_n - F \\ (\tilde{y})' & \text{if } (\tilde{y})' \in H_n - F,\ \tilde{y} \in F \text{ and } y \text{ is not pushed} \\ \psi(\tilde{y})' & \text{otherwise} \end{cases}$$

and maps edges in H_{n-1} to paths in $H_n - F$ precisely as discussed in section 2.

Theorem 3.9. *Algorithm 3.2 is a local, deterministic algorithm. For any $p < \frac{1}{2}$ there is a sufficiently small constant $c_{11} > 0$ such that if the nodes of H_n fail independently and partially with probability p, for sufficiently large N the following is true with probability $1 - N^{-c_{11}}$. Algorithm 3.2 takes $O(\log N)$ steps and constructs a one-to-one embedding of H_{n-1} into $H_n - F$ with dilation 3 and congestion $2 \log N$. The embedding has the property that if a constant degree network C is embedded in H_{n-1} then the induced embedding in $H_n - F$ has constant congestion.*

The only part of the theorem which we have not yet checked is the number of steps stage 4 takes. Figure 3-6 gives a more detailed description of the implementation of stage 4. First, each unassigned poor node is tentatively assigned to a constant number of topnode neighbors. Each topnode chosen attempts to tentatively assign itself to one of its unassigned rich neighbors. Each poor node then finds a topnode to which it is tentatively assigned which successfully was assigned to a rich node.

Since by lemma 3.8 there are few unassigned poor nodes in any small sphere and we know that most topnodes will have many rich neighbors, the above procedure will assign every unassigned poor node to a topnode with high probability.

for all poor nodes v unassigned after stage 3 do
  for $k \leftarrow \epsilon n + 1$ to n
    if v^k is an unassigned topnode
      assign v to v^k unless v is already assigned to $8d$ neighbors
for all assigned topnodes u do
  for $k \leftarrow \epsilon n + 1$ to n
    if u^k is an unassigned rich node assign u to u^k and stop
for all poor nodes v unassigned after stage 3 do
  for $k \leftarrow \epsilon n + 1$ to n
    if v^k was assigned to v and succeeded in being assigned to a rich node w
      push v_k to w and assign v to v^k

Figure 3-6: Stage 4.

3.4 Routing Using Only Live Nodes

If we consider total faults instead of partial faults, algorithm 3.2 fails in several places. In fact, any path in the embedding which does not consist of only a single edge has at least one dead node internal to it. In order to handle total faults we will replace the paths of length 3 in $H_n - F$ which constitute the edges of the embedded hypercube with paths of length 7 which use only live nodes.

In the remainder of the chapter we will use probabilistic algorithms. To guarantee the performance of these algorithms, we will need to know that certain assumptions about the distribution of faults hold true. These assumptions are stated in several lemmas (for example, lemmas 3.10 and 3.11). Given that these distribution assumptions hold (which they do except with inverse polynomial probability), the algorithms work with high probability. The errors arising during particular executions of the algorithms are thus in some sense independent of the existence of unusual fault patterns.

First we establish that all nodes have a reasonably large number of live neighbors.

Lemma 3.10. *There exists a $c_{12} > 0$ such that for any node v, the set N_v of live neighbors of v has cardinality at least ϵn with probability $1 - N^{-c_{12}}$.*

Proof. A calculation almost identical to the one in the proof of lemma 3.6. ■

With probability close to 1, all pairs of nodes are connected by many paths each

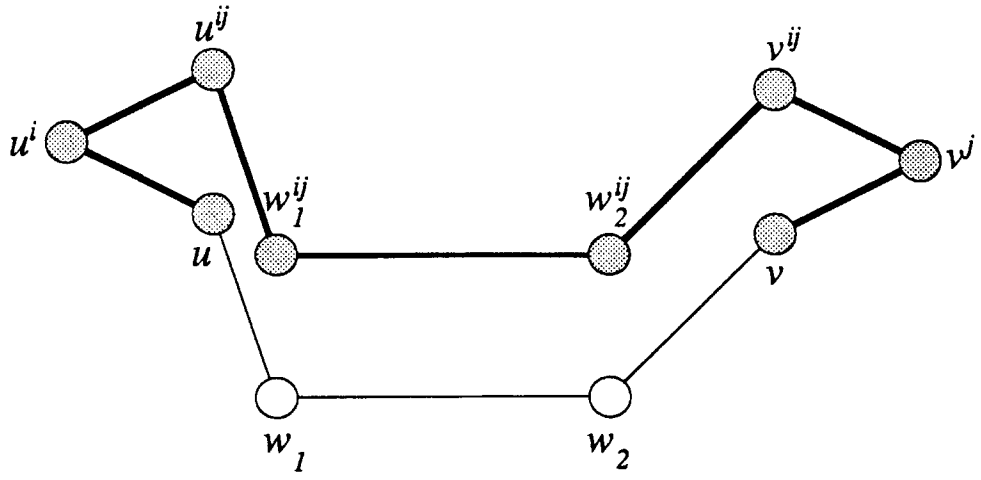

Figure 3-7: A live path P^{ij}. The darkened path simulates an edge (u, v).

of which contains only live nodes.

Lemma 3.11. *Suppose every node fails with probability $p < 1/2$. Then with probability $1 - N^{-c_{12}}$ there are $\Omega(n^2)$ live paths of length at most 7 between any points u and v within Hamming distance 3, where we choose c_{12} as in lemma 3.10.*

Proof. We will prove only the case where the distance is 3; the other cases are similar. Let $P = (u, w_1, w_2, v)$ be a path of length 3 between u and v. The paths we will consider are of the type $P^{ij} = (u, u^i, u^{ij}, w_1^{ij}, w_2^{ij}, v^{ij}, v^j, v)$. Let N_u be the set of dimensions k for which u^k is live and similarly for N_v. Take the larger of the two sets $\{P^{ij} | i \in N_u, j \in N_v, i < j\}$ and $\{P^{ij} | i \in N_u, j \in N_v, i > j\}$. By lemma 3.10 this set has cardinality $\epsilon^2 n^2/2$. The interior 4 nodes of these paths are disjoint for different pairs i, j (if we discard i, j where either i or j is a dimension used along P) and the outer 4 nodes are all alive. Thus with high probability $\Omega(n^2)$ of these paths use only live nodes. ■

Once we have established the existence of live paths it is a simple matter to find them algorithmically. However, if we look for them deterministically it is difficult to bound the congestion. A random algorithm which uniformly chooses a random live path for each pair of nodes is easier to analyze. Before we show how well the random algorithm performs, we prove a simple lemma about balls and boxes.

Lemma 3.12. *If each of α balls is placed randomly and uniformly into one of β boxes, then with probability $1 - (\alpha e/\beta\gamma)^\gamma$ there are fewer than γ balls placed in the first box.*

Proof. The probability that there are more than γ balls in the first box is no more than $\binom{\alpha}{\gamma}(\frac{1}{\beta})^\gamma \leq \frac{(\alpha)^\gamma}{\gamma!}(\frac{1}{\beta})^\gamma = (\frac{\alpha}{\beta})^\gamma \frac{1}{\gamma!}$. ■

Theorem 3.13. *If we uniformly choose a random live path between each pair of chosen nodes at the end of algorithm* 3.2, *then with high probability the resulting embedding will have congestion $O(\log N)$ and induced congestion $O(\log N/\log\log N)$.*

Proof. The estimates will follow from lemma 3.12. The balls correspond to the edges of the paths of the embedding and the boxes are the edges of H_n. The paths which potentially share a given edge can be separated into classes. We assign a path to a class depending on which position in the associated live path the edge would occupy if the live path were actually routed through the edge. We will then show that with high probability the congestion due to the live paths associated with any one class is $O(n)$. Since there will be only four classes, the result will follow.

Fix an edge (s,t). Given a path P, put P in class r if (s,t) is the r^{th} edge along P^{ij} (reading from the closest end) for some pair (i,j). There are four cases we need consider.

r = 1: Then $s = u$. Since there are at most $n-1$ paths beginning at u, there are only $n-1$ paths of this sort even in the worst case.

r = 2: Then $(s,t) = (u^i, u^{ij})$. There are $n-1$ possible values for u, each an endpoint of at most $n-1$ paths P. Since there are at least $(\epsilon n)^2/2$ choices for (i,j) for each of these $O(n^2)$ paths, lemma 3.12 applies to show that the probability that more than $O(\log N/\log\log N)$ of these paths are actually chosen to go through (s,t) is at most $O(N^{-k})$.

r = 3: Then $(s,t) = (u^{ij}, w_1^{ij})$. If $w_1 \in H'_{n-1}$, then the path P was embedded for the edge (u, w_1). Thus only one path of this type exists for each pair (i,j). If $w_1 \in H''_{n-1}$, then the path P was embedded for an edge incident to w_1^0. Thus only $n-1$ paths of this type exist for each pair (i,j). Therefore the total number of paths P in this class

is no more than n^3 for any edge (s,t). Again the probability that any one of these paths will actually be chosen to go through (s,t) is no more than $2/(\epsilon n)^2$. By lemma 3.12 the probability that more than $O(\log N)$ of these paths are chosen that way is at most $O(N^{-k})$.

r = 4: Then $(s,t) = (w_1^{ij}, w_2^{ij})$. There are two cases. If both $w_1, w_2 \in H''_{n-1}$, then P was embedded for (w_1^0, w_2^0). Thus only one path of this type exists for each pair (i,j). If $w_1 \in H'_{n-1}, w_2 \in H''_{n-1}$, then P was embedded for an edge incident to w_1. Thus only $n-1$ paths of this type exist for each pair (i,j). The rest of the analysis is identical to that of the previous case.

Thus the probability that the congestion is more than $O(\log N)$ is at most $O(N^{-k}) \times N \log N/2 = O(N^{-k+1} \log N)$.

To prove the induced congestion is $O(\log N/\log\log N)$, note that only one path from class 1 can contribute to the induced congestion. Note also that classes 3 and 4 have only $O(n^2)$ paths in them which can contribute to the induced congestion, since the original edges could not have been adjacent. Thus the analysis for induced congestion due to classes 3 and 4 reduces to that of case 2 above. ■

3.5 An Algorithm for Constant Delay Embedding

In section 3.4 we resorted to probabilistic means to find fault-free communication paths. We will use probabilistic methods again in this section, together with a more uniform view of the nodes of the cube. We allow the max load to rise to a constant, and in return we achieve constant congestion.

To achieve a constant delay embedding, we need the load, dilation and congestion to all be constant. The embedding we will find will have a load and congestion which depend strongly on the probability of failure - clearly the more nodes that fail, the more nodes that have to be simulated by any one processor. However, the dilation will always remain five, and each processor will be simulated by one of its neighbors, provided that $p < 1 - \sqrt[4]{.5}$ (about .16).

In order to simplify the analysis, each node (live or dead) finds a neighbor to

simulate it. We first assign nodes to live neighbors so that no node simulates more than a constant number of its neighbors. Then each pair of nodes simulating neighbors finds a live path between them of length five so that no more than a constant number of these paths congest any edge. We will use two similar algorithms to accomplish these two tasks.

3.5.1 Assigning Nodes to Live Neighbors

Let A_p and s_p be constants (to be determined later) which depend only upon the probability p of failure. Call a node *unsaturated* if it is live and if it has been assigned to simulate fewer than A_p of its neighbors. Otherwise, it is *saturated.*

The assignment algorithm proceeds in rounds. During a round, a previously unsaturated node might be picked by enough unassigned nodes so as to exceed its capacity A_p. In such a case, we require the node to accept enough of the simulation requests to saturate it. Algorithm 3.3 performs the first phase.

for $i = 1$ to $s_p n$
  for each unassigned node w
    w picks one of its neighbors uniformly
  each unsaturated node v agrees to simulate as many nodes as it can
  without exceeding its capacity
  all excess nodes remain unassigned

Figure 3-8: Algorithm 3.3.

Since the algorithm never assigns a saturated node to simulate another node, no node simulates more than A_p nodes. Thus, a constant load embedding results.

To facilitate our proofs, we will first formulate a sequential algorithm similar to algorithm 3.3. We will prove that this new algorithm assigns to each node a neighboring node to simulate it. We will then show that, except for a small proportion of executions, the algorithms behave the same.

In each round of algorithm 3.4, unassigned nodes act sequentially. Each node chooses a neighbor to simulate it only after all lower ordered nodes have chosen. We

```
for i = 1 to s_p n
    for unassigned nodes w in arbitrary order
        if w has fewer than α_p n unsaturated neighbors
            arbitrarily dedicate enough (saturated) neighbors
        w picks one of its neighbors uniformly
        if the chosen node is unsaturated or dedicated
            w is assigned to that node
        else w remains unassigned
```

Figure 3-9: Algorithm 3.4.

would like to ensure that all nodes have a large number of choices that will result in a successful assignment. Let α_p depend only upon the probability p. If some node w has fewer than $\alpha_p n$ unsaturated neighbors to choose from during its turn, we designate an arbitrary set of saturated neighbors as *dedicated* to w during its turn. If w chooses a dedicated node during that particular turn, the dedicated node agrees to simulate w even though it is saturated. We dedicate enough nodes so that w has at least $\alpha_p n$ neighbors which, if chosen, will agree to simulate it.

We will show below that with high probability no nodes are ever dedicated during algorithm 3.4. In that case, the result is the same whether unassigned nodes choose sequentially or in parallel. Thus we will show that algorithms 3.3 and 3.4 produce the same output.

The following lemma proves that algorithm 3.4 terminates quickly.

Lemma 3.14. *With high probability all nodes have been assigned after $s_p n$ steps of algorithm 3.4, for sufficiently large s_p.*

Proof. Because each node always has at least $\alpha_p n$ neighbors which will simulate it if chosen, the probability that a given node is assigned during some step is at least α_p, regardless of what has occurred in previous steps. Thus the probability that a node remains unassigned after $s_p n$ steps is no more than $(1-\alpha_p)^{s_p n}$. This quantity is less than N^{-k} as long as $s_p > k/\alpha_p$. ■

Lemma 3.15. *For $p < 1 - \sqrt[4]{5}$, there exists an ϵ_p and a constant $c_{13} > 0$ such that*

with probability at least $1 - N^{-c_{13}}$ *each node has at least* $\epsilon_p n$ *live neighbors.*

Proof. The probability that a node has fewer than ϵn live neighbors equals

$$\sum_{i=0}^{\epsilon n} \binom{n}{i} (1-p)^i p^{n-i}$$

Since the ratio of consecutive terms is always greater than $(1-p)/p$, this sum is bounded by a constant times its last term. That term is

$$\binom{n}{\epsilon n} (1-p)^{\epsilon n} p^{(1-\epsilon)n} \leq \binom{n}{\epsilon n} p^{(1-\epsilon)n}$$

The second term in the product can be made less than $N^{-1-c_{14}}$ for some c_{14} by taking ϵ small enough. The first term in the product can be made less than $N^{c_{14}/2}$ by taking ϵ small enough as well. The probability that some node has too few neighbors is bounded by the sum of the probabilities for the individual nodes. This multiplies the above bound by N. Thus for any ϵ below both of these thresholds, the theorem applies. ■

The following two lemmas show that with high probability algorithm 3.4 never dedicates saturated nodes. Thus with high probability algorithms 3.3 and 3.4 behave identically. This proves that algorithm 3.3 assigns all nodes with high probability. Similar reasoning proves the Dance Hall Theorem described in the introduction.

Lemma 3.16. *Given a failure rate* p*, assume that every node has at least* $\epsilon_p n$ *live neighbors. Then with high probability a given node* v *never has fewer than* $\alpha_p n$ *unsaturated neighbors available during algorithm 3.4, for* $\alpha_p = \frac{\epsilon_p}{2}$.

Proof. For v to have fewer than $\alpha_p n$ unsaturated neighbors at some point during algorithm 3.4, at least $(\epsilon_p - \alpha_p)n = \alpha_p n$ of v's neighbors must have become saturated during the course of the algorithm.

Each node always has at least $\alpha_p n$ neighbors (including dedicated nodes) to which it might be assigned during any step. Further, if it is assigned, it is equally likely to be assigned to any one of those neighbors. Thus no node has a probability greater than $1/\alpha_p n$ that it will be assigned to any given neighbor, no matter what other assignments have been made previously.

To saturate $\alpha_p n$ of v's neighbors, there must be at least $A_p \alpha_p n$ nodes at Hamming distance two from v each of which is assigned to a neighbor of v. There are no more than n^2 nodes which might be assigned to some node in v's neighborhood. Each one of these nodes has at most two neighbors of v to which it might be assigned. Although the probabilities of such selections are dependent, the probability a given node is assigned to a neighbor of v is at most $2/\alpha_p n$, no matter what choices the other nodes made. The probability that at least $\alpha_p n$ of v's neighbors become saturated is thus no more than

$$\binom{n^2}{A_p \alpha_p n} \left(\frac{2}{\alpha_p n}\right)^{A_p \alpha_p n} \leq \left(\frac{2e}{A_p \alpha_p^2}\right)^{A_p \alpha_p n}$$

For A_p large enough, this quantity is an inverse polynomial in N. ■

Lemma 3.16 implies that with high probability algorithms 3.3 and 3.4 behave identically. We know that algorithm 3.4 successfully assigns each node to a neighbor with high probability and that algorithm 3.3 never assigns more than A_p nodes to any node. We conclude that algorithm 3.3 achieves a constant load embedding with high probability.

3.5.2 Assigning Edges to Paths

Once we've assigned simulating nodes, we need to find paths to simulate the edges in the hypercube. Say that v^b simulates v and $v^{kb'}$ simulates v^k. Then to simulate the edge (v, v^k), the nodes v^b and $v^{kb'}$ choose a path between them of the form $P(v, v^k, b, b', r) = (v^b, v^{br}, v^r, v^{rk}, v^{rkb'}, v^{kb'})$. To avoid ambiguity, we will refer to the choice of r as if it were made by v and v^k even though v^b and $v^{kb'}$ actually choose.

For two adjacent nodes v and v^k, let $S(v, v^k, b, b')$ be the set of dimensions $r \neq k$ for which $P(v, v^k, b, b', r)$ is a live path. Because $p < 1 - \sqrt[4]{.5}$, there is a chance $(1-p)^4 = s > \frac{1}{2}$ that any given path $P(v, v^k, b, b', r)$ is live. Note that the paths $P(v, v^k, b, b', r)$ $(r \neq k)$ are node-disjoint for a fixed choice of v, v^k, b and b'. Thus the probability that any one of them is live is independent of the other paths.

Lemma 3.17. *With high probability, for all quadruples* (v, v^k, b, b'), $|S(v, v^k, b, b')| > \eta_p n$ *for some constant* η_p.

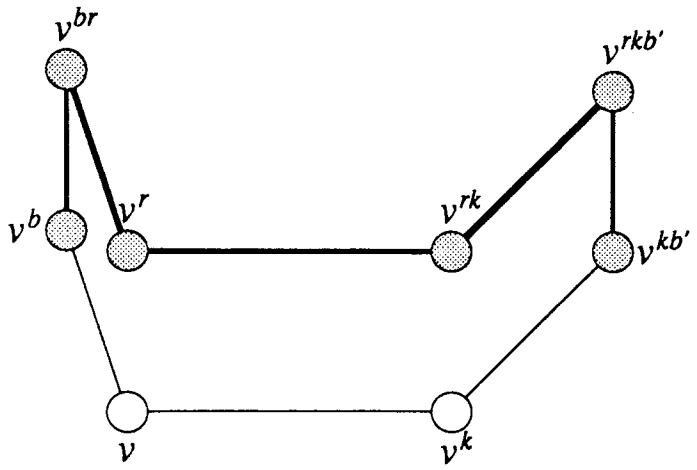

Figure 3-10: A choice of live path.

Proof. Same as lemma 3.15, except that there are $N \log^3 N$ different quadruples. ■

With high probability, we know that all pairs of neighbors have many paths from which to choose. What remains is for them to decide in a systematic but local fashion how to choose from among these paths without congesting any edge too much. In the rest of this section, we explore a way to choose paths in this manner.

Take a node v simulated by its neighbor v^b and consider the set $E_{v,b}$ of edges $\{(v^{br}, v^r)\}$. There are $2n^2$ nodes w (all of the form $w = v^{rt}$ or $w = v^{brt}$) which (like v) might potentially use one of the edges in the set as a second edge along a path. Any node which actually does must be simulated by its neighbor across dimension b. The next lemma bounds the number of such nodes.

Lemma 3.18. *For sufficiently large δ_p and with high probability, of the $2n^2$ nodes at distance 0 or 2 from either v or v^b, no more than $\delta_p n$ of them are simulated by neighbors across dimension b.*

Proof. As noted before, each node has a probability no more than $1/\alpha_p n$ of borrowing across any given dimension, regardless of the choices made by other nodes. The probability that many nodes choose across the same dimension is no more than

$$\binom{2n^2}{\delta_p n}\left(\frac{1}{\alpha_p n}\right)^{\delta_p n} \leq \left(\frac{2e}{\alpha_p \delta_p}\right)^{\delta_p n}$$

Of course, the actual probabilities depend on the particular $\delta_p n$-size subset we consider and on the relative order in which the nodes of the subset successfully found neighbors to simulate them. Then any node's probabilities are conditioned upon other nodes' previous choices. No matter how these choices are made, however, the stated probabilities are upper bounds on the actual probabilities since when each node chooses it always has at least $\alpha_p n$ choices.

For sufficiently large δ_p, this is smaller than an inverse polynomial in N. ■

Each of the at most $\delta_p n$ nodes (except for v and v^b) can use at most two edges in the set $E_{v,b}$ as a second edge along some path. To use an edge as a second edge, such a node would have to be a neighbor of one of the nodes incident to the edge. If w is of the form $w = v^{rt}$, then w is adjacent to v^r and v^t and no other node incident to an edge in $E_{v,b}$. Similar reasoning applies to nodes w which satisfy $w = v^{brt}$. Trivially, each of v and v^b can use no more than n edges of $E_{v,b}$ as a second edge along some path. If we sum over all edges in $E_{v,b}$ the number of nodes which can use each edge as a second edge counting according to multiplicity, the total will be no more than $(2\delta_p + 2)n$. Therefore no more than $\eta_p n/4$ of these edges will have more than $\gamma_p = 4(2\delta_p + 2)/\eta_p$ of those $\delta_p n$ nodes potentially using them as second edges. Let $S'(v,b) = \{r|$ more than γ_p nodes can send a path through the edge $(v^{br}, v^r)\}$. Then $|S'(v,b)| \leq \eta_p n/4$.

Let $T(v, v^k, b, b') = S(v, v^k, b, b') - S'(v,b) - S'(v^k, b')$. Then for each adjacent pair of nodes v and v^k, $|T(v, v^k, b, b')| > \eta_p n/2$. The sets $T(v, v^k, b, b')$ will be crucial for our reasoning. The probability that a pair successfully choose a path between them is lower bounded by the probability that they successfully choose the path from $T(v, v^k, b, b')$.

Note that among the edges in all the paths represented by the sets $T(v, v^k, b, b')$, there are now only a logarithmic number of quadruples (w, w^j, c, c') which might potentially congest any given edge. We've already limited the number of paths for which the edge is the second edge along the path. If the edge is the first edge along the path, then one of the edge's endpoints is the simulating node. Each endpoint simulates only a constant number of nodes, and each simulated node contributes

exactly n paths. If the edge is the third edge along the path, then the path is simulating an edge at Hamming distance one from the edge considered. There are exactly n edges of this type. The cases in which the edge is the fourth or fifth edge along the path are identical to the first two cases. Thus each edge can be potentially congested by no more than $\mu_p n = (4A_p + 2\gamma_p + 1)n$ paths.

We can now describe algorithm 3.5, which assigns paths to simulate edges. During algorithm 3.5, each edge will decide whether or not to accept some path routed through it. Because the other edges in the path simultaneously decide whether or not to accept the path, it is possible that some might accept it while others reject it. If this happens, we assume that an accepting edge counts the path as contributing to its load anyway. Call an edge *saturated* if it has accepted exactly B_p paths routed through it. Otherwise, call it *unsaturated.* Order the pairs (v, v^k) lexicographically. As before, in any round an edge accepts an arbitrary set of pairs which try to route through it until it reaches its capacity.

for $i = 1$ to $s'_p n$
  for each unassigned adjacent pair of nodes (v, v^k)
    (v, v^k) pick a path between them uniformly
  each unsaturated edge agrees to as many paths routed through as it can
  without exceeding its capacity, deciding conflicts arbitrarily
  all excess pairs remain unassigned

Figure 3-11: Algorithm 3.5.

Parallelling what we did before, we will present algorithm 3.6, a sequential version of algorithm 3.5. We will show that this modified algorithm terminates having assigned paths between every pair of nodes simulating neighbors, with high probability. Maintaining the parallel with what we proved earlier in this section, we will then show that the two algorithms perform indistinguishably, with high probability. At any time when the pair (v, v^k) attempt to choose a path between them during algorithm 3.6, let $U(v, v^k, b, b')$ be the subset of $T(v, v^k, b, b')$ consisting of dimensions r for which all of the edges along $P(v, v^k, b, b', r)$ are unsaturated. Define the ded-

ication of a path containing a saturated edge in a fashion similar to the dedication of saturated neighbors before. We dedicate paths to the pair (v, v^k) whenever $\beta_p n$ choices for a simulating path do not exist.

```
for i = 1 to s'_p n
    for all unassigned pairs (v, v^k) in arbitrary order
        if |U(v, v^k, b, b')| < β_p n
            dedicate enough r ∈ T(v, v^k, b, b')
        (v, v^k) pick a path between them uniformly
        if the chosen path is unsaturated or dedicated
            (v, v^k) is assigned to the path
        else (v, v^k) remains unassigned
```

Figure 3-12: Algorithm 3.6.

Lemma 3.19. *For a suitably large choice of the constant s'_p, with high probability all pairs of nodes searching for an assignment to a path have been assigned one after $s'_p n$ steps of algorithm 3.6.*

Proof. Each pair is successfully assigned with probability at least β_p during any step. The rest of the proof is identical to that of lemma 3.14. ■

We now show that with high probability algorithm 3.6 never adds dedicated paths with saturated edges to any $U(v, v^k, b, b')$. Thus with high probability algorithms 3.5 and 3.6 behave identically. This proves that algorithm 3.5 assigns all necessary paths with high probability.

Lemma 3.20. *With high probability no set $U(v, v^k, b, b')$ ever has cardinality less than $\beta_p n$ at the beginning of some step of algorithm 3.6, given $\beta_p = \eta_p/4$.*

Proof. There are at most $\mu_p n$ pairs which have a non-zero probability of congesting a given edge on some path represented by an $r \in T(v, v^k, b, b')$. Thus at most $5\mu_p n^2$ pairs have non-zero probability of congesting any of those edges, counting according to multiplicity. For a path to leave $U(v, v^k, b, b')$ one of its edges must become saturated. For $(\eta_p/2 - \beta_p)n = \beta_p n$ paths to become unavailable, $B_p \beta_p n$ pairs must choose a path crossing an edge on some path represented by an $r \in T(v, v^k, b, b')$.

The probability that a pair chooses any particular path is at most $1/\beta_p n$, no matter what other choices are made. Thus if there are q_{w,w^j} paths that a particular pair (w, w^j) might choose which contain an edge on some path in $T(v, v^k, b, b')$, then the probability that (w, w^j) chooses such a path is at most $q_{w,w^j}/\beta_p n$, and $\sum_{w,w^j} q_{w,w^j} \leq 5\mu_p n^2$.

By a moment generating function argument similar to those in lemmas 2.1 and 2.7 and in theorem 2.13, the probability that more than $\beta_p n$ paths become unavailable is therefore no more than $O(N^{-k})$ for arbitrary k. ■

With high probability $O(n)$ steps are sufficient to select all paths. Since we have guaranteed that the paths have constant congestion, this proves the following theorem.

Theorem 3.21. *For each $p < 1 - \sqrt[4]{.5}$ (about .16) there exists an ϵ_p and an η_p such that with probability $1 - N^{-c_{15}}$, at least $\epsilon_p n$ neighbors of every node are live and $|S(v, v^k, b, b')| \geq \eta_p n$ for all quadruples (v, v^k, b, b'). Given these facts hold, there is an $O(\log N)$ step algorithm which with high probability finds an embedded fully functioning N-node cube in $H_n - F$ with constant load, dilation and congestion. The paths which simulate the edges of the cube only use live nodes.*

3.6 Implementing the Constant Delay Embedding

As given so far, the algorithms of the previous section are far from implementable. Each node needs to know information about which nodes have decided to simulate which other nodes, which paths it may route through, whether or not certain tentative assignments have been finalized, and so forth. In this section we will show how such information might be exchanged in polylogarithmic time per step. This implies that the embedding of the previous section is obtainable in polylogarithmic time.

Focus on any particular node v. Because v might be faulty, one of its neighbors must choose a simulating node for it. Arbitrarily, we will use the lexicographically smallest labelled live neighbor to simulate v during the course of algorithm 3.3. First, the neighbors must agree on which one of them is the lowest. During any step of

algorithm 3.3, that neighbor of v must inform all the other neighbors which one of them v selected during that step. Both of these operations are trivial once we understand how a node's neighbors can communicate even with faults.

Each node v^i broadcasts to v's other neighbors by first broadcasting to all of its neighbors. Then each node v^{ij} passes the information to its unique other neighbor which is also a neighbor of v, the node v^j. A picture of this type of broadcasting appears in figure 3-13.

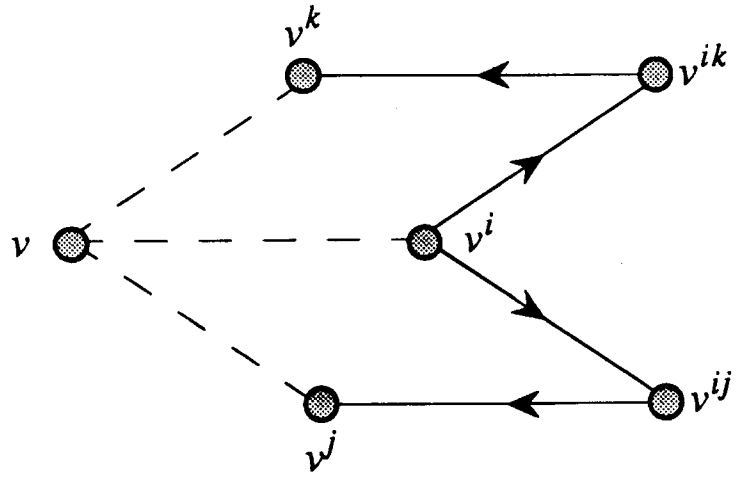

Figure 3-13: Broadcasting to other neighbors.

We only care if the message gets through to the other neighbors of v which are live. The broadcast we have described sends the messages through a set of intermediary nodes, several of which are likely to be faulty. Thus if each node broadcasts just once, we might expect that several nodes will not receive the information they need. We remedy this problem by allowing each node to broadcast its information and then repeating the broadcast twice more. With probability $1 - N^{-c_{16}}$, every neighbor of v is informed of the activity of all of v's other neighbors. We prove this scheme works by showing that with probability $1 - N^{-c_{16}}$, for every node v of the hypercube, every live pair v^i and v^j of v's neighbors are connected by a live path consisting of two or three broadcasts.

Lemma 3.22. *With probability* $1 - N^{-c_{16}}$, *between every pair of live neighbors*

v^i, v^j *of every node* v *there is a live path of the form* $(v^i, v^{ik}, v^k, v^{kj}, v^j)$ *or of the form* $(v^i, v^{ik}, v^k, v^{kl}, v^l, v^{lj}, v^j)$.

Proof. Consider a node v and a neighbor v^i. The neighbor v^i successfully broadcasts to another neighbor v^k exactly when both v^k and the intermediary node v^{ik} are both live. Since the probability of failure is no more than $1 - \sqrt[4]{.5}$, the probability that one or both of these nodes are dead is no more than $1 - \sqrt[2]{.5}$. Further, none of these pairs $\{v^k, v^{ik}\}$ share a common node. Thus there is a $\rho > 0$ and a $c_{17} > 0$ such that with probability $1 - N^{-c_{17}}$ ρn of the pairs will be live, for all neighbors v^i of v.

Next take two disjoint sets S_1 and S_2 each containing ρn neighbors of a given node v and consider the set $T(S_1, S_2) = \{v^{kl} | v^k \in S_1, v^l \in S_2$ and v^{kl} is live$\}$. Since there are $\rho^2 n^2$ pairs of nodes v^k, v^l which satisfy the first two requirements and each pair has a constant probability that it satisfies the last requirement (independent of other nodes), with high probability the set $T(S_1, S_2)$ is nonempty. There are no more than N^2 ways to choose the sets S_1 and S_2 for any given node v. Thus with high probability the set $T(S_1, S_2)$ is nonempty for all choices of S_1 and S_2. Since there are only N choices for v, with high probability for each node v and each choice of S_1 and S_2, the set $T(\S_1, S_2)$ is nonempty.

With probability $1 - N^{-c_{16}}$ (for any $0 < c_{16} < c_{17}$), the conclusions of the first two paragraphs hold. For a given node v, let $V_1 = \{v^k | v^{ik}$ and v^k are both live$\}$ and $V_2 = \{v^k | v^{jk}$ and v^k are both live$\}$. Then if $|V_1 \cap V_2| \neq 0$, a the path of length four connects v^i and v^j. If $|V_1 \cap V_2| = 0$ then v^i and v^j are connected by a path of length six. ■

Before algorithm 3.5 can route the simulating paths, each node must know which nodes simulate the neighbors of the nodes it simulates. At the end of algorithm 3.3 each live node knows which nodes simulate each of its neighbors. At least ρn neighbors of every node are live. For every pair of neighbors v and v^k, we only need some live neighbor of v to communicate with some live neighbor of v^k. Each neighbor v^i of v attempts to route along the path $v^i, v^{ij}, v^{ijk}, v^{jk}$ to each neighbor v^{jk} of v^k. We are only interested in the $(\rho n)^2$ paths which begin and end at live nodes. Each of these

paths intersects at most one other and with high probability one of the $(\rho n)^2/2$ node-disjoint paths will be nonfaulty. Since there are only a polynomial number of pairs of neighbors, each with only a polynomial number of possible sets of live neighbors, this communication will be possible with high probability. A series of three broadcasts by all nodes will accomplish the task in $O(\log^3 N)$ steps.

Finally, we will describe how to implement a slight variant of algorithm 3.5. Assume that v has an even number of 1's in its bit-vector representation. To avoid confusion, to find a simulating path for the edge (v, v^k), only the node v^b simulating v will actually choose a path. Say that during a step of the algorithm, instead of choosing a random dimension in $U(v, v^k, b, b')$, v^b chooses a random dimension from $\{1, 2, \ldots, n\}$. Then we know (1) the probability that v^b chooses any particular $r \in U(v, v^k, b, b')$ does not increase, (2) all sets $U(v, v^k, b, b')$ have cardinality $\beta_p \log N$ with at least the same probability as in algorithm 3.5 and (3) each node v^b has probability at least β_p of choosing an $r \in U(v, v^k, b, b')$.

During any step of this modified algorithm, all of the nodes that have chosen an $r \in U(v, v^k, b, b')$ succeed with at least the probability stated in the analysis of algorithm 3.5. All other nodes may or may not find an unsaturated path and may or may not encounter too much congestion. With high probability, if we run the modified algorithm $2/\beta_p$ times as long as algorithm 3.5, each node v^b will choose an $r \in U(v, v^k, b, b')$ at least as many times as it did in algorithm 3.5. Thus, even if nodes never find simulating paths except when they choose an $r \in U(v, v^k, b, b')$, all nodes will find the necessary paths at least as successfully as before.

Each node that chooses a path attempts to route a message describing the path along the path. Any node along the path can send messages back if it detects too much congestion along one of its edges. If the message comes back, the even node knows that it was unsuccessful. Otherwise, both the even node and the odd node which is the message's destination know which path to use in the future.

3.7 Extensions and Remarks

As mentioned in the introduction, edge faults are easily handled once node faults are understood. Say each edge fails with probability p_e, each node fails with probability p_n and the failure of any component is independent of the failure of other components. Then all results still follow with little change. Specifically, as long as $p_n + p_e - p_n p_e < 1 - \sqrt[5]{.5}$ (about .13), the algorithms of sections 3.5 and 3.6 work with high probability. The only addition to our reasoning is that when one node tries to communicate with a neighbor node, it is unsuccessful not only if the neighbor is faulty but also if the link between them has failed.

This work extends to the case in which p is small; that is, if $p < N^\alpha$ for $0 > \alpha > -1$. In this case, faults are so far between that the results of the second section can be strengthened. The deterministic algorithm 3.1 achieves a constant delay embedding with high probability. This result follows directly from the following fact.

Lemma 3.23. *If faults occur with probability p for small p then with high probability no sphere of radius 14 contains more than a constant number of faults.*

Proof. Say $p < N^\alpha$. Then the probability that m nodes out of any given n^{14} nodes are faulty is no more than

$$\binom{n^{14}}{m} N^{\alpha m} < N^{\alpha m} n^{14m} N$$

There are at most N such spheres to consider. If $m > -1/\alpha$, the total probability that some sphere contains m faults is an inverse polynomial whose exponent can be diminished by increasing m. ■

Each simulating node only needs to distribute its connections among the dilation 7 fault-free paths discovered in section 3.4.

Last, a word about the practicality of the results of this chapter. We have made little attempt to optimize constants since we need large constants to obtain the full breadth of our results. However, in practice the full strength of these theorems will probably be unnecessary. We cannot expect half the processors in a network to fail as a normal occurrance. We are optimistic that when the number of faults is moderate,

our techniques will work quite well either on their own or as a basis for some heuristic approach.

Chapter 4

Embedding Trees Dynamically

4.1 Introduction

Achieving high performance on a parallel computer requires the satisfaction of two potentially conflicting requirements. First, the computational load posed by the program should be evenly shared among all processors (load balancing). Second, processes communicating frequently should be placed on processors that are close (communication locality).

This problem has been studied abstractly as the problem of embedding a process graph G in a processor graph H ([BCHLR], [BCLR], [BI], [C], [GHR], [HJ], [KLMRR]). The vertices of G are processes comprising the parallel program, with edges representing communication between processes. The vertices of H are processors, and the edges represent communication channels. For many computations, it is possible to predict G before execution. In such cases it is useful to map the vertices of G into those of H so as to minimize load, dilation and congestion.

This chapter focuses on embedding arbitrary binary trees into the butterfly and hypercube networks. Trees arise naturally in many computations: divide-and-conquer algorithms, branch-and-bound search ([KZ]), functional expression evaluation, and image understanding (quad/oct trees). In [BCLR], Bhatt et al. showed that every N-node binary tree could be embedded in an N-processor hypercube such that each processor received a single tree node, and the maximum dilation was $O(1)$. Embed-

ding trees into butterfly networks is harder, because the butterfly is much sparser than the hypercube. In [BCHLR], Bhatt et al. showed how to embed the complete binary tree with N nodes in a butterfly network with N processors with constant dilation and load. The problem of embedding arbitrary trees into butterfly networks was left open.

Tree structured computations are often dynamic. As the computation progresses, the tree may grow or shrink, in a manner which may be impossible to predict beforehand. In [BC], Bhatt and Cai propose a dynamic version of the embedding problem. They consider a process graph which is a binary tree that can grow during execution. At each step any node of the tree that does not have two children can request to spawn a child. The dynamic embedding problem is harder than the static one since newly spawned children must be allocated to processors incrementally, without making assumptions about how the tree will grow in the future. Further, the placement decision must itself be implemented within the network in a distributed manner without accessing global information. The paradigm proposed by Bhatt and Cai disallows process migration; i.e. once a process is placed on a particular processor, it cannot be moved subsequently. Obviously, allowing migration can potentially give better load balancing/dilation but can also be extremely expensive in practice.

Bhatt and Cai present ([BC]) a randomized algorithm for dynamically growing trees with M vertices on an N processor binary hypercube. Each child process is placed no farther than a distance $O(\log \log N)$ from its parent. Further, with high probability (independent of the tree shape) the algorithm only assigns $O(M/N + 1)$ vertices to each processor. The congestion of the embedding was not determined but is probably on the order of $\log N$.

4.1.1 Summary of Results

We consider the problem of growing trees on butterfly and hypercube networks. Our framework is identical to that of Bhatt and Cai ([BC]), although our growth algorithms are substantially simpler and have provably better performance. We begin by describing a level-by-level strategy for embedding a binary tree in a butterfly. Mod-

ifications to this scheme form the basis of all our embedding algorithms. The first modification we introduce is the use of random *flip bits*, which randomize the locations of tree nodes within a level of the butterfly. Analysis of the behavior of these flip bits is sufficient to prove our first result.

Theorem 4.1. *An arbitrary binary tree T with M vertices can be dynamically grown on an N processor hypercube with dilation 1 such that with high probability the maximum load per processor is $O(M/N + \log N)$.*

Note that this is optimal to within a constant factor whenever the tree T is large (i.e., $M \geq N \log N$). For these large trees, it gives an optimal $O(M/N)$ load as in [BC] while improving dilation from $O(\log \log N)$ to 1. Next we present another modification of the scheme involving *level balancing* — in effect, we stretch certain paths within the tree so that the number of tree nodes assigned to any level of the butterfly is balanced. This modification leads to our next result, this time for a butterfly.

Theorem 4.10. *An arbitrary binary tree T with M vertices can be dynamically grown on an N processor butterfly network with dilation 2 such that with high probability the maximum load per processor is at most $O(M/N + \log N)$*

Again, this is optimal to within a constant factor when $M \geq N \log N$. This result is a substantial improvement over previous work since not even good static embeddings of arbitrary binary trees were known. Finally, we take advantage of an embedding of the butterfly into the hypercube which embeds entire levels of the butterfly to subcubes of the hypercube in order to develop a scheme for local redistribution of load within levels. This leads to an embedding algorithm for the hypercube which simultaneously optimizes maximum load and dilation. In addition, the congestion of the embedding is optimal if $M = O(N)$.

Theorem 4.14. *An arbitrary tree T with M vertices can be grown on a N processor hypercube with constant dilation such that with high probability the maximum load is $O(M/N + 1)$ and the congestion is $O(M/N + 1)$.*

It should be noted that although our theorems are phrased in terms of trees

which only grow, these embedding algorithms are also effective for dynamic trees which can both grow and shrink at their leaves. Consider a binary tree T which grows and shrinks. At each stage in the tree's evolution, the probability space of possible embeddings of the current form of the tree T' is equivalent to the space of embeddings which would have occured had we simply grown the tree T' using the same algorithm. Therefore the same results hold for each step in the tree's evolution (assuming, of course, that the total number of steps in the tree's evolution is bounded by a polynomial in N).

We also prove a lower bound for deterministic embedding algorithms for hypercubes which shows that any deterministic algorithm which balances load must necessarily have dilation $\Omega(\sqrt{\log N})$. It follows that any embedding algorithm which simultaneously optimizes load and dilation (to within constant factors) must be randomized. This consequence also holds for the butterfly, since it is a subgraph of the hypercube.

Tom Leighton, Abhiram Ranade and Eric Schwabe coauthored all the work appearing in chapter four.

4.1.2 Overview

The basic embedding algorithm is presented in section 4.2 along with the introduction of flip bits and the proof of theorem 4.1. The level-balancing scheme is introduced and analyzed in section 4.3, along with a proof of theorem 4.10. Improvements to the hypercube embedding algorithm and proof of theorem 4.14 are given in section 4.4. Section 4.5 states and proves the lower bound for deterministic algorithms.

4.2 The Basic Growth Algorithm

4.2.1 Preliminary Scheme

We begin with a *level-by-level* strategy for growing a tree on an N-node butterfly network. For this chapter, we set n so that $N = n2^n$. That is, the N-node butterfly

has n levels.

In the cases where we are ultimately interested in an embedding in a hypercube, we will first embed the tree in a butterfly, and then consider some embedding of the butterfly in the hypercube. We place the root of the tree on processor $\bar{0}_0$ in the butterfly. This processor is connected to two processors in level 1, on which we place the children of the root. These processors are in turn connected to 4 level 2 processors, which will in turn receive the children of the root's children, and so on. This strategy enables us to grow any n level binary tree with dilation 1, and with at most one tree vertex per butterfly processor. Trees with greater height are wrapped around; i.e. level n vertices are placed in butterfly level 0, and so on. The set of tree vertices which are mapped to level i of the n level butterfly consists of those vertices in levels $i, i+n, i+2n \ldots$ and so on; we refer to this as the i^{th} *level set* of the tree. There are two issues we need to consider:

1. *Evenly distributing tree vertices within the processors in each level.* We would like the vertices belonging to level set i to be evenly distributed among the processors in the i^{th} level of the butterfly; i.e. to guarantee that no single processor in level i receives too many vertices.

2. *Evenly distributing tree vertices among different butterfly levels.* For example, when mapping a complete binary tree of height h, level $h - 1 \bmod n$ of the butterfly would receive the leaves of the tree, or about half the total number of vertices. Ideally, we would like the vertices to be divided evenly among all the levels of the butterfly.

We will defer our consideration of the second issue until section 4.3. First, a modification of the basic scheme helps us achieve balance within a level.

4.2.2 Flip Bits

A random *flip* bit is generated at each vertex of the tree to decide where its children will be spawned. Consider a vertex v of the tree that has been placed on some

processor p in level i of the butterfly. This node is connected to processors q and r in level $i+1 \bmod n$, which will receive the children of v. The flip bit chosen for vertex v decides whether the left child of v will be placed on q or on r. The right child is then placed on the other processor. Note of course that it is not necessary that v have two children – the bit only determines where the children will be placed if they are ever spawned.

In section 4.3 we will show that this ensures even distribution within each level. Intuitively, each vertex is effectively placed using a random path determined by the flip bits chosen along its ancestors. For now, this modified scheme is sufficient to prove theorem 4.1.

Theorem 4.1. *An arbitrary binary tree T with M vertices can be grown dynamically on an N processor hypercube with dilation 1 such that with high probability the maximum load per processor is $O(M/N + \log N)$.*

Theorem 4.1 follows directly from the following lemma.

Lemma 4.2. *An arbitrary tree T with M vertices can be grown in a butterfly network of N processors such that each column in the butterfly receives no more than $O(M/2^n + n)$ vertices with high probability.*

Suppose this lemma were true. Then by simulating the $N = n2^n$-node butterfly by a 2^n-node hypercube, where each node of the hypercube simulates an entire column of the butterfly, we have an embedding algorithm for the hypercube which achieves dilation 1 and load $O(M/N + \log N)$ with high probability. Thus this lemma is sufficient to prove theorem 4.1.

The general idea behind the proof of lemma 4.2 is that a large number of vertices will be placed in the same column in the butterfly only if the flip bits on the paths leading to these vertices are chosen in a specific (unlikely) manner.

A *stagnant path* p is a maximal path $v(1), v(2), \ldots, v(l)$ in T with $v(1)$ towards the root such that all $v(i)$ are placed in the same column v of the butterfly. Let the *leader* of p be the n^{th} ancestor of $v(1)$, and the *trace* of p be the set of $n+l-1$ vertices between the leader (inclusive) and $v(l)$ (exclusive). If $v(1)$ is in the first n levels of

the tree, then the leader of the path is defined to be the root of the tree.

Notice that there is a unique path in the butterfly from the leader of a stagnant path p to vertex $v(1)$. Thus, given the column in which the leader lies, and the column in which the path p lies, we can completely determine the flip bits chosen along the trace of the path. The next observation is that the traces of distinct stagnant paths mapped to the same column are distinct; i.e. the information gained from one trace is different from that obtained in the other.

Lemma 4.3. *Let p and p' be two distinct stagnant paths placed in the same column of the butterfly. Then their traces are vertex disjoint in the tree.*

Proof. Contrary to the lemma, suppose the lowest point in the tree at which the traces intersect is vertex u. At vertex u, the two traces are mapped to the same column of the butterfly. Likewise, the two stagnant paths are mapped to the same column. The two children of u are mapped to different columns of the butterfly, however, and therefore the traces must reconverge in some butterfly column between the children of u and the beginnings of the two stagnant paths. However, the two paths cannot meet again in any column until they have traversed all n levels of the butterfly. Since the two stagnant paths are at a distance less than n from u, the traces cannot reconverge in the butterfly before reaching them, and we have a contradiction. ■

Lemma 4.4. *For any column v of the butterfly, there is at most one stagnant path mapped to v such that $v(1)$ is in the first n levels of the tree.*

Proof. This lemma follows immediately from lemma 4.3 by noting that any two such paths will have the same leader (the root of the tree). ■

Proof. (of Lemma 4.2) We shall count the number of different settings of the flip bits that give rise to some column having at least $C = k(M/2^n + n)$ tree vertices. This can be done as follows:

1. Choose the column: 2^n choices.

2. Choose the number of stagnant paths: C choices.

3. Choose the endpoint of each path: $\binom{M}{C_0}$, where C_0 is the number of stagnant paths. Define $\beta = C/C_0$.

4. Choose the length of the paths: $\binom{C+C_0}{C_0}$ choices.

5. Choose the flip bits at all vertices in T except those in the C_0 traces. The total number of flip bits is M, and the length of the jth trace is $n + l_j - 1$, except for the possible case when one stagnant path has v_1 in the first n levels of the tree, in which case the length of its trace is $l_j - 1$. Thus the total number of bits this step fixes is: $M - \sum(n + l_j - 1) + n = M - (C_0(n-1) + C) + n$. Thus the total number of choices is $2^{M-(C_0(n-1)+C)+n}$.

First we claim that the above choices completely determine all the flip bits. To see this, consider the trace with its leader belonging to the smallest level in T, of all traces. Clearly, the last step of the above procedure fixes the position of the leader. This fixes all the bits in the trace, since the endpoint and the length of the trace are known. The bits for the other traces are similarly determined.

The total number of ways of choosing all the bits is 2^M. Thus the probability that some column gets more than C vertices is at most

$$\begin{aligned} & 2^{2n} C \binom{M}{C_0} \binom{C+C_0}{C_0} 2^{M-(C_0(n-1)+C)} / 2^M \\ \leq\ & 2^{2n} C \left(\frac{M(C+C_0)e^2}{C_0^2}\right)^{C_0} 2^{-(C_0(n-1)+C)} \\ \leq\ & 2^{2n} C \left(\frac{2e^2 M(C+C_0)}{C_0^2 2^n 2^\beta}\right)^{C_0} \\ \leq\ & 2^{2n} C \left(\frac{2e^2}{k} \frac{\beta(\beta+1)}{2^\beta}\right)^{C_0} \end{aligned}$$

To go from the first line to the second we have used the inequality $\binom{n}{r} \leq (ne/r)^r$. Choosing $k > 10e^2$, and noting that $\beta(\beta+1) \leq 5(2^{\beta/2})$, we can simplify the above expression to:

$$\begin{aligned} & 2^{2n} C (\tfrac{1}{2^{\beta/2}})^{C_0} \\ \leq\ & 2^{2n} C 2^{-C/2} \\ \leq\ & 2^{-C/4} \\ \leq\ & 2^{-kn/4} \end{aligned}$$

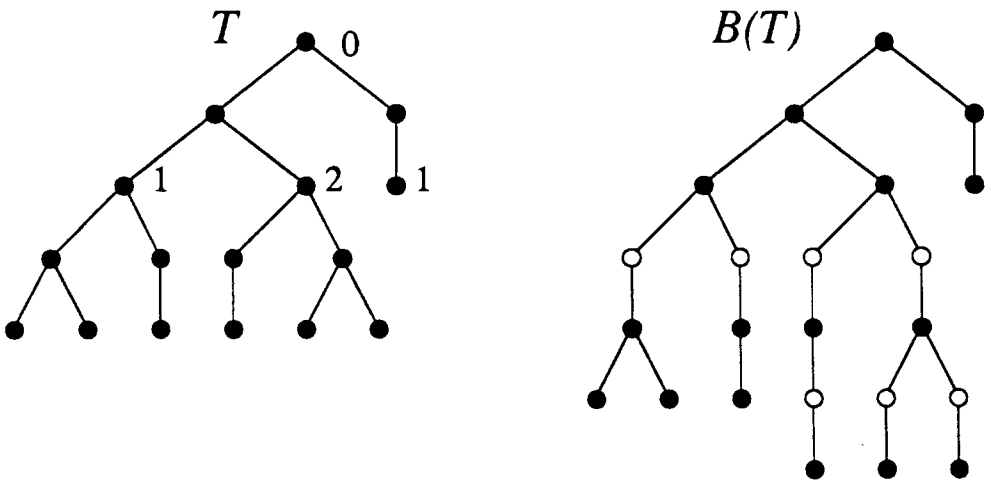

Figure 4-1: Level balancing a tree, n=6. The numerical labels indicate the stretch counts chosen at those nodes. White nodes indicate dummy vertices.

$$\leq \quad N^{-k/8}$$

■

4.3 Embedding in the Butterfly

In this section we introduce a modification to the embedding algorithm which insures that with high probability the nodes of the binary tree are distributed evenly among the levels of the butterfly. We then prove that the flip bits described in the previous section are sufficient to distribute the tree nodes evenly within each level.

4.3.1 A Level-Balancing Transformation

We transform the tree T being grown by selectively inserting dummy vertices into some of its edges during the growth. Even if some level originally has a disproportionately large number of vertices, the newly introduced vertices help to even the distribution of the tree vertices among the levels.

The n-way level balancing transformation is as follows. Define a vertex of T to

be *distinguished* if it lies in level $i = 0 \pmod{n/3}$.[1] For each distinguished vertex v in T we pick a random number $S(v)$ between 0 and $n/3$ called the *stretch count.* We insert a single dummy vertex in each of the edges that connect v to its descendants in levels $i + 1$ through $i + S(v)$. Figure 4-1 illustrates the transformation. Note that this transformation can be applied as the tree grows. Each node only needs to know what level of the tree T it belongs to, and the stretch count generated at its nearest distinguished ancestor. This is sufficient information to decide whether or not a dummy vertex is inserted when a child is spawned.

The new tree $B(T)$ that results is grown on the butterfly using the procedure described in section 4.2.2. This gives a dilation 1 embedding for $B(T)$. This corresponds to a dilation 2 embedding of T, since some of the edges in T were replaced by two edges in $B(T)$.

4.3.2 Analysis of Tree Balancing

We show that the n-way level balancing transformation of section 4.3.1 is sufficient to evenly distribute the tree vertices among the levels in the butterfly. In particular, we show that for any tree T, no level set in $B(T)$ will contain a disproportionately large number of vertices. Since level i of the butterfly receives vertices from the i^{th} level-set of $B(T)$, this implies that tree vertices are uniformly distributed among the butterfly levels.

Lemma 4.5. *For an arbitrary tree T, the n-way level-balancing transformation gives a tree $B(T)$ such that the total number of vertices in the i^{th} level-set of $B(T)$ is at most $O(M/n + 2^n)$ with high probability.*

We will prove the following slightly modified (but equivalent) version. Define the i^{th} *level set triple* of a tree to be the set of vertices from level sets i, $i + n/3$ and $i + 2n/3$. Define a partition of T into 3 zones as follows (Figure 4-2). Zone 0 consists of vertices in levels kn through $kn + n/3 - 1$. Zone 1 consists of vertices in levels

[1] In what follows we may make references like "(mod x)" or "contribution of x messages" when x may not be integral. Rounding these quantities to integers does not affect the correctness of the proof. For ease of exposition, we shall not consider the issue.

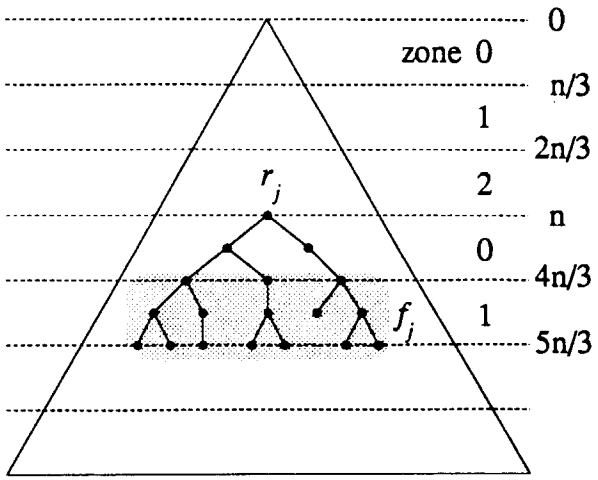

Figure 4-2: Subdivision into Zones, and a forest f_j.

$kn + n/3$ through $kn + 2n/3 - 1$. Zone 2 consists of vertices in levels $kn + 2n/3$ through $(k+1)n - 1$. Each zone consists of a number of trees of maximum height $n/3$. We will show that no level set triple of $B(T)$ will receive more than $O(M/n)$ vertices from any zone of T, with high probability. Lemma 4.5 follows because there are only 3 zones, and since the number of vertices in a level set triple upper bounds the number of vertices in a level set.

The key observation is that each zone can be partitioned into a set of forests $f_1, f_2, \ldots, f_\alpha$ that contribute independently to level set triple i, for any i. We illustrate the partitioning for zone 1. Each f_j consists of all trees from zone 1 between levels $kn + n/3$ and $kn + 2n/3 - 1$ that have a common ancestor r_j at level kn, for some fixed k. Other zones are partitioned similarly.

Lemma 4.6. *Let X_j denote the number of zone 1 vertices from a forest f_j placed in level set triple i of $B(T)$. Then all variables X_j are mutually independent, and $E(X_j) = 3M_j/n$, where M_j is the number of vertices in f_j.*

Proof. Let variable Y_j denote the level set triple into which the roots of the trees in f_j are placed. By definition, these roots are all placed in the level set triple given by the level set triple of r_j plus $S(r_j)$, mod $n/3$. Since the stretch counts of the r_j's are uniformly selected from $[0, n/3]$ and are mutually independent, it follows that the

Y_j's are also uniformly selected from $[0, n/3]$ and are mututally independent. Since X_j is completely determined by Y_j and the stretch counts chosen at the roots of trees in f_j, it follows that the X_j are mutually independent, and that $E(X_j) = 3M_j/n$. ■

Similarly, this lemma holds for any other zone of the tree T, except for the first section of zone 0, which contains the vertices in levels $0 \ldots n/3 - 1$. However, this segment of the tree contains at most $2^{n/3} - 1$ nodes, which will be mapped one-to-one to nodes of the butterfly.

Proof. (of Lemma 4.5) The X_j are independent random variables. Clearly, no X_j can contribute more than $2^{2n/3}$ vertices, since the forest is part of a tree of height no more than $2n/3$. The mean of each X_j is $3M_j/n$, where M_j is the number of vertices in f_j; therefore the mean of $X(= \sum X_j)$ is at most $\sum M_j \leq 3M/n$. We have by the independence of the X_j that for any t, $E[e^{tX}]$

$$\begin{aligned} &= \prod_i E[e^{tX_i}] \\ &= \prod_i \sum_\lambda Pr[X_i = \lambda] e^{t\lambda} \end{aligned}$$

As in lemma 2.7, the expectation is maximized when only the events $[X_i = 0]$ and $[X_i = 2^{2n/3}]$ have positive probability. Suppose there were some value x, not equal to 0 or $2^{2n/3}$, such that $Pr[X_i = x] = \delta > 0$. Then by the convexity of e^{tX_i}, changing $Pr[X_i = x]$ to 0 and setting $Pr[X_i = x - 1] = Pr[X_i = x + 1] = \delta/2$ would increase the expectation of e^{tX_i}. It follows that in order to maximize the expectation, the two endpoints of the interval must be the only events with positive probability. If we use Markov's inequality to put an upper bound on $Pr[X_i = 2^{2n/3}]$ then

$$\begin{aligned} E[e^{tX}] &\leq \prod_i \left(\left(1 - \frac{3M_i/n}{2^{2n/3}} \right) + \frac{3M_i/n}{2^{2n/3}} e^{t2^{\frac{2n}{3}}} \right) \\ &= \prod_i \left(1 + \frac{3M_i/n}{2^{2n/3}} (e^{t2^{2n/3}} - 1) \right) \\ &\leq \prod_i \exp \left(\frac{3M_i/n}{2^{2n/3}} (e^{t2^{2n/3}} - 1) \right) \\ &\leq \exp \left(\frac{3M/n}{2^{2n/3}} (e^{t2^{2n/3}} - 1) \right) \end{aligned}$$

Again using Markov's inequality, we obtain for any constant b, $Pr[X \geq 3bM/n]$

$$= Pr[e^{tX} \geq e^{3bMt/n}]$$

$$\leq \frac{\exp(\frac{3M}{n2^{2n/3}}(e^{t2^{2n/3}}-1))}{e^{3bMt/n}}$$

This quantity is minimized at $t = lnb/2^{2n/3}$. At this value of t, and as long as $M \geq n^2 2^{2n/3}$, this quantity is smaller than N^{-k} for some constant k which can be made as large as desired by choosing b sufficiently large. ■

4.3.3 Effectiveness of Flip Bits

We now show that, given the effectiveness of the level-balancing algorithm, the flip bits suffice to distribute the tree nodes within the levels of the butterfly.

Lemma 4.7. *Let W_i denote the total number of vertices in level set i in an arbitrary binary tree T. When T is grown on a butterfly with n levels, no processor from any level i receives more than $O(W_i/2^n + n)$ vertices with high probability, for all i.*

In other words, whenever $W_i > n2^n$, each of the 2^n processors in level i will receive roughly the same number of tree vertices.

The key to the proof is the observation that the vertices placed on a processor can be attributed to a large number of mutually independent sources. To see this, partition T into subtrees $T_1, T_2, \ldots$ where each subtree is rooted at some vertex in level $kn + i$ and consists of all the descendants of the that vertex between levels $kn + i + 1$ and $kn + i + n$ (figure 4-3).

Lemma 4.8. *At most one level n vertex from each subtree T_j will be placed on any processor p on level i of the butterfly. The probability of a vertex from T_j being placed on processor p is $w_j/2^n$, where w_j denotes the number of vertices in level n of tree T_j. Further the contributions of the different subtrees to p are mutually independent.*

Proof. Any tree T_j can have at most 2^n vertices at level n, and the growth algorithm guarantees that these will be placed on distinct processors within a single level. Thus we know that at most one vertex from a tree T_j will be placed on a given processor p in level i of the butterfly.

It follows from the above that the number of vertices from T_j placed on p is a random variable with value either 0 or 1. The probability that any given vertex from

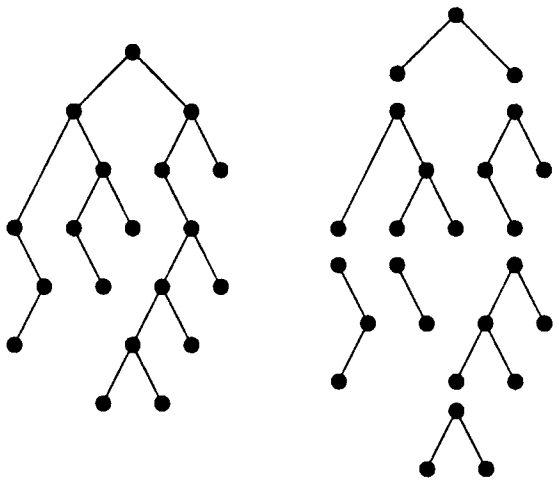

Figure 4-3: The tree T and its partition, $i = 1$, $n = 2$.

level n of T_j will be placed on p is $1/2^n$, so the expectation of this random variable is $w_j/2^n$. Since the value of the random variable can only be 0 or 1, $w_j/2^n$ must be the probability that it is 1. Thus the probability of a vertex from T_j being placed on p is $w_j/2^n$.

The independence between different subtrees follows because the flip bits in each subtree are picked independently. ■

To complete the proof of lemma 4.7, we need the following lemma, due to Hoeffding ([H]).

Lemma 4.9. *[Hoeffding] If we have L independent Bernoulli trials with respective probabilities $p_1, \ldots, p_L$, with $Lp = \sum p_i$, and $m \geq Lp+1$ is an integer, the probability of at least m successess is at most $B(m, L, p)$, where $B(m, L, p) \leq (Lpe/m)^m$.*

Proof. (of Lemma 4.7) The number of vertices placed at a processor is the sum of independent random variables corresponding to each tree T_j. The expected number of vertices is $\sum w_j/2^n = W_j/2^n$. The probability that some processor receives more than $k(n + W_j/2^n)$ vertices is at most (using lemma 4.9):

$$\left(\frac{k(n + W_j/2^n)}{eW_j/2^n}\right)^{-(kn+kW_j/2^n)} < (k/e)^{-kn}$$

Thus the probability that one of the 2^n processors in any of the n levels receives more

than $k(W_j/2^n + n)$ vertices is at most

$$n2^n(k/e)^{-kn} < N^{-k\log k/k_1}$$

for some constant k_1. ■

Theorem 4.10. *An arbitrary binary tree T with M vertices can be grown dynamically on an N processor butterfly network with dilation 2 such that with high probability the maximum load per processor is at most $O(M/N + \log N)$.*

Proof. By lemma 4.5, with high probability we have $W_i = O(M/n + 2^n)$ for all i, and by lemma 4.7 , with high probability no processor in level i will receive more than $O(n + W_i/2^n)$ vertices. Thus with high probability, fewer than $O(\log N + M/N)$ vertices are mapped to any processor. ■

4.4 An Improved Hypercube Embedding

The butterfly can be embedded in the hypercube with dilation 2 such that each level of the butterfly is a subcube of the hypercube. Therefore we can have the hypercube simulate any embedding algorithm for the butterfly, with a unique 2^n-node subcube simulating each level. We will take advantage of this by using a scheme which has each level (subcube) receiving only $O(M/n + 2^n)$ tree nodes, and developing a method for local distribution within these subcubes which will reduce the load on each indvidual processor while guaranteeing low congestion. We begin with some preliminaries.

4.4.1 Embedding the Butterfly and Star Covers

Let $G(x)$ be the *Grey code* value of the binary string x, defined by $G(x_{\log n} \ldots x_1) = x_{\log n}|x_{\log n} \oplus x_{\log n-1}| \ldots |x_2 \oplus x_1$. For any bit string x, $G(x)$ and $G((x+1) \bmod n)$ differ in exactly one bit position. For an integer i, let $bin(i)$ be the binary representation of i. The embedding which maps butterfly processor v_l to node $G(bin(l))|bin(v)$ of the hypercube has dilation 2 and maps each level of the butterfly to a distinct 2^n-node subcube of the hypercube. Also note that within each level l, if v and v^k differ in

exactly one bit, then there is a hypercube edge between the embedded locations of the nodes v_l and v_l^k.

For any node x of a 2^n-node hypercube, we define the *full star centered at* x to be the set of nodes consisting of x along with the n nodes adjacent to x. The existence of perfect one-error-correcting codes implies that when $n = 2^m - 1$ for some integer m there exists a collection of $2^n/n + 1$ full stars such that every node of the hypercube belongs to precisely one star in the collection.

Suppose n is not of this form. Consider the largest n' such that $n' \leq n$ and n' is of the form $n' = 2^m - 1$; then $n' \geq n/2$. We can partition the hypercube into subcubes of $2^{n'}$ nodes, and cover each of these with full stars. This *star cover* perfectly covers the nodes of the 2^n-node hypercube. Each star in the star cover consists of a node x and some subset of $\Theta(n)$ (in this case at least $\frac{n}{2}$) of its neighbors.

Choose a star cover for a 2^n-node hypercube, and duplicate this cover in each subcube of the $N(= n2^n)$-node hypercube which corresponds to a level of the butterfly. This collection of stars yields a star cover of the N-node hypercube; call it C.

4.4.2 Modifying the Embedding Algorithm

Our discussion of the hypercube algorithm has two parts:

1. We describe a modified algorithm for embedding on the butterfly which, when simulated on a hypercube, maps at most $O(M/2^n + n)$ tree nodes to any star in the cover C, with high probability.

2. We show how to deterministically redistribute the load within a star of the hypercube among its nodes in such a way that each node receives $O(M/N + 1)$ tree nodes, the dilation remains constant and the resulting congestion is $O(M/N + 1)$.

We begin by showing how to modify the butterfly embedding algorithm given in the previous section so that when it is simulated on the hypercube, the amount of load assigned to any star in the cover C is balanced.

We will modify our embedding algorithm as follows. Use the embedding algorithm from the previous section, but where previously we placed the children of a tree node $v \in B(T)$ which was embedded in level l into level $l+1$, choosing their locations by a random flip bit, we will now place the first child of v into level $l+2$, using a *pair* of flip bits to determine its position within the level, and placing the second child (if it exists) at the location in that level determined by complementing both flip bits. It is clear that this will increase dilation by a factor of two.

Since we are embedding the level-balanced tree $B(T)$, we know that, with high probability, each level-set of the tree contains $O(M/n + 2^n)$ nodes. As in lemma 4.6, we observe that the vertices placed in a single star come from many independent sources.

Partition $B(T)$ into subtrees $T_1, T_2 \ldots$ in such a way that the root of each subtree is embedded at level $l+2$ in the butterfly (or level $l+1$ if n is odd) and each subtree contains the descendants of its root down to the nodes embedded at level l in the butterfly.

Lemma 4.11. *Consider an arbitrary star S in C, contained in level l of the butterfly. Then at most two vertices from each subtree can be placed on processors in S. Furthermore, the contributions of each subtree to S are mutually independent.*

Proof. Any subtree can have at most $2^{n/2}$ vertices placed in level l of the butterfly, and these will necessarily be placed at distinct locations within the level. Suppose that three vertices from the same subtree were mapped to the star S. Since the flip bits are chosen in pairs, any pair of these vertices must be mapped to locations which differ in an even number of bits; since they are all mapped to the same star, any pair of them must differ in exactly two bit positions. Consider the paths to each of these three vertices from their lowest common ancestor; call this vertex x. Clearly, two of the vertices must be descendants of one child of x, and one must be a descendant of the other. The vertex (call it y) which is the lone descendant of one of the children of x now differs from both of the other two vertices in two bit positions which are not corrected elsewhere in the tree. However, at some point the paths of the other two vertices diverge (since they are placed on different processors in level l), and y's

path cannot duplicate the flip bits on both paths simultaneously. Therefore y differs from one of the other two vertices in at least four bit positions, contradicting the supposition that all three vertices were in the same star in level l. Therefore at most two vertices from the same subtree can be placed in the star S.

The independence between different subtrees follows from the fact that the flip bits are picked independently in each subtree. ■

Lemma 4.12. *We can embed an arbitrary binary tree T with M nodes into an N-node hypercube such that, with high probability, no star in the cover C receives more than $O(M/2^n + n)$ tree nodes.*

Proof. Consider an arbitrary star S in level l from the cover C. Let X_i be the number of tree nodes from subtree T_i which are assigned to processors in S. The X_i are independent random variables, each with maximum value 2 (from lemma 4.11) and mean $\Theta(m_i n/2^n)$, where m_i is the number of leaves of the subtree T_i. It follows that the mean of $X = \sum X_i$ is $\Theta(mn/2^n)$, where m is the number of tree nodes embedded into level i of the butterfly. But since we are balancing levels by embedding the tree $B(T)$, we have $m = O(M/n + 2^n)$, so that the mean of X is less than $c_{18}(M/2^n + n)$ for some constant $c_{18} > 0$. By the same argument as in the proof of lemma 4.5, we can bound the expectation of the random variable e^{tX} by

$$E[e^{tX}] \leq \exp\left(\frac{c_{18}}{2}(M/2^n + n)(e^{2t} - 1)\right)$$

Again as in lemma 4.5, we obtain for any constant b,

$$\begin{aligned} & Pr[X \geq \tfrac{bc_{18}}{2}(M/2^n + n)] \\ = \; & Pr[e^{tX} \geq e^{(tbc_{18}/2)(M/2^n+n)}] \\ \leq \; & \frac{\exp((c_{18}/2)(M/2^n+n)(e^{2t}-1))}{e^{tbc_{18}/2(M/2^n+n)}} \end{aligned}$$

This value is minimized at $t = lnb/2$, at which point this quantity is smaller than N^{-k} for k which can be made as large as desired by choosing b sufficiently large. ■

4.4.3 Redistributing Load Within Stars

With high probability, each star in the cover has at most $O(M/2^n + n)$ tree nodes assigned to its $\Theta(n)$ nodes. We would like to redistribute the $O(M/2^n + n)$ load on each star evenly among the $\Theta(n)$ nodes of the star, using the hypercube edges connecting butterfly nodes within the same level, so that two conditions hold:

1. Each node gets at most $O(M/N + 1)$ load.
2. We can choose paths of constant length between the redistributed locations of adjacent tree nodes so that the congestion on any hypercube edge is at most $O(M/N + 1)$.

If these two conditions can be achieved by a redistribution scheme which runs dynamically as the tree is embedded then, with high probability, the embedding algorithm achieves load $O(M/N + 1)$, dilation $O(1)$, and congestion $O(M/N + 1)$ — simultaneously optimizing load and dilation to within constant factors. In addition, the congestion will be optimal if $M = O(N)$.

Place an $O(M/N + 1)$ upper limit (with appropriate choice of constant depending on the constant in lemma 4.12 and the number of elements in each star) on the number of tree nodes which can be assigned to a single node. All additional load is sent to some other node in the star which has room. It is clear that we have sufficient capacity over each star to handle the load, and that we will still have constant dilation. In addition we will have maximum load $O(M/N+1)$ at each node of the hypercube. Note that this is not allowing process migration—each tree node is redistributed *before* it is embedded into the hypercube. Once the node's redistributed location is determined, it is embedded there permanently.

Suppose we redistribute one tree node from node v^i to node v^j in the star centered at v in the hypercube (load coming from or going to the center is redistributed directly). This load is passed along the path $v^i \rightarrow v^{ij} \rightarrow v^j$ rather than through the center of the star.

Lemma 4.13. *If all load being redistributed among points of stars is follows paths of the form $v^i \rightarrow v^{ij} \rightarrow v^j$ rather than paths through the centers of stars, then the resulting congestion due to this redistribution is $O(M/N+1)$.*

Proof. For each star in the cover, consider the corresponding *extended star*, which consists of the star centered at v, plus all vertices v^{ij} such that both v^i and v^j are in the star. The edges in the extended star consist precisely of those paths along which load can be redistributed in the star centered at v. The redistribution within that star can add at most congestion $O(M/N+1)$ to any of the edges in the extended star. All that remains is to observe that any edge in the hypercube is in at most two extended stars. Thus the total congestion it receives from redistribution is $O(M/N+1)$. ■

Let l be the level of the butterfly to which u is mapped; then v is mapped to level $l+2$. Furthermore, their positions within their respective butterfly levels differ in at most two bit positions (before redistribution). We consider here the case where both u and v are both initially mapped and redistributed to some point of a star rather than the center. When one or both of them is mapped to the center of a star, the argument is even simpler.

Let x and y be the centers of the stars to which u and v, respectively, are mapped. Let p and q be the dimensions within the star to which u is mapped and redistributed, and likewise r and s for v. Let f_1,f_2 be the flip bits selected when v is embedded as a child of u. We then define the path from u, which is redistributed to x^q in level l, and v, which is redistributed to y^s in level $l+2$, as follows (this procedure is illustrated in figure 4-4):

1. Move from level l to level $l+1$ to level $l+2$ along the edges determined by the flip bits f_1,f_2.

2. Flip the bits in positions p, then q, in effect undoing the redistribution of u which was performed in level l. We are now at y^r, the original location of v before redistribution.

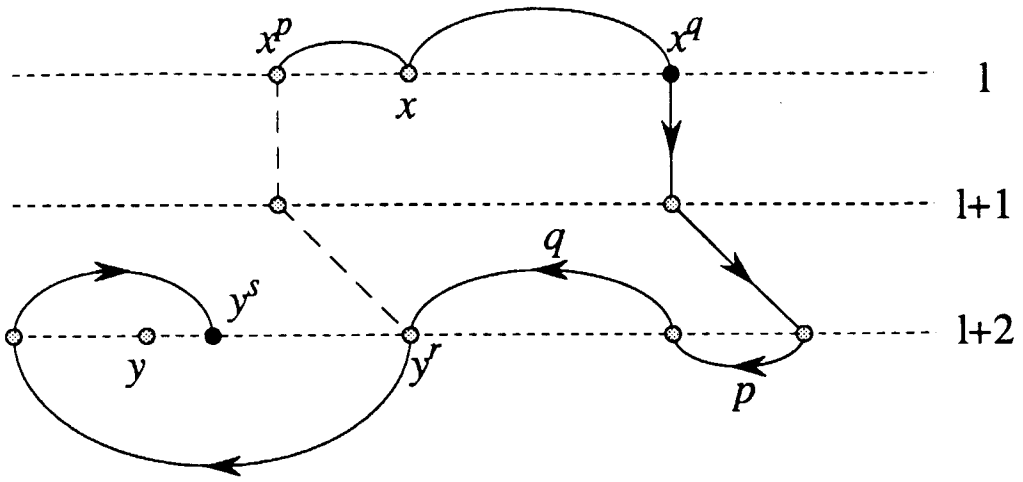

Figure 4-4: The path chosen between redistributed node locations. The dashed lines indicate the path determined by the flip bits, before redistribution. The first pair of directed edges also show this choice of flip bits. The second pair undoes the redistribution at level l. The last pair balances the load at level $l+2$.

3. Flip the bits in positions s, then r. This takes us to y^s, the redistributed location of v in its star in level $l+2$.

In order to show that the congestion is $O(M/N+1)$ in this case, it suffices to show two things. First, that the congestion along each edge of the butterfly is $O(M/N+1)$. Second, that the congestion along each hypercube edge connecting nodes within a butterfly level is $O(M/N+1)$. From these two facts it follows that the total congestion is $O(M/N+1)$.

Consider an arbitrary butterfly edge. There are at most two nodes of the butterfly which, when choosing the paths to their descendants, can use that edge. Since after redistribution each of these nodes has load $O(M/N+1)$, the congestion along the edge being considered can also be at most $O(M/N+1)$.

The congestion on hypercube edges connecting butterfly nodes within a level has two sources: (1) the redistribution of the nodes embedded to that level, and (2) undoing the redistribution of the parents of the nodes embedded to that level.

It follows directly from lemma 4.13 that the total congestion from the first source does not exceed $O(M/N+1)$. We can break up the congestion derived from the

second source into four sets, according to the flip bits chosen along the paths from the parents of the nodes in the level we are considering. Each fixed setting of flip bits determines a bijective map of the nodes, and therefore the jump edges, from two levels above to the current level. The congestion on any edge from undoing the redistribution of parents equals the congestion on its preimage from the original redistribution. The congestion in each set is therefore $O(M/N+1)$, and so the total congestion derived from undoing the redistributions is also $O(M/N+1)$. It follows that the entire congestion on any edge is $O(M/N+1)$.

Theorem 4.14. *An arbitrary tree T with M vertices can be grown on a N processor hypercube with constant dilation such that with high probability the maximum load is $O(M/N+1)$ and the congestion is $O(M/N+1)$.*

4.5 A Lower Bound for Deterministic Algorithms

In this section, we prove that any deterministic algorithm for dynamically embedding an M-node tree in an N-node hypercube ($M \geq N$) which maintains maximum load $a\frac{M}{N}$ must have not only maximum but *average* dilation $\Omega(\sqrt{\log N}/a^2)$. It follows that any deterministic embedding algorithm which achieves $O(M/N+1)$ load must necessarily result in embeddings with dilation $\Omega(\sqrt{\log N})$ for some trees. Thus any embedding algorithm which simultaneously optimizes maximum load and dilation (to within constant factors) must be randomized.

Theorem 4.15. *Any deterministic algorithm for dynamically embedding trees in an N-node hypercube which achieves load aM/N for a tree with M ($\geq N$) nodes must have average edge length $\Omega(\sqrt{\log N}/a^2)$.*

Proof. Let aM/N be the load maintained by the embedding algorithm when embedding an M-node tree. Define the *size* of a node in the hypercube to be the number of 1's in the n-bit string associated with the node. Partition the hypercube into $6a$ blocks, each block corresponding to some range of node sizes and containing $N/6a$ nodes. Since there are at most $O(N/\sqrt{\log N})$ nodes of any size, each block must

contain at least $\Omega(\sqrt{\log N}/a)$ sizes. This means that any two nodes which are in non-adjacent blocks are at distance $\Omega(\sqrt{\log N}/a)$ from each other.

Choose an arbitrary $M \geq N$, and grow a path of $M/2$ nodes, starting at the root. At this point, some block must contain $M/12a$ tree nodes; choose such a block. We will continue growing the tree from the $M/12a$ nodes in the chosen block. Grow paths from each of these tree nodes simultaneously, stopping each path's growth when it reaches a hypercube node which is neither in the chosen block nor in a block adjacent to it. The total number of nodes in the chosen block and adjacent blocks is at most $N/2a$; since the algorithm maintains load aM/N, this set of nodes contains at most $(aM/N)(N/2a) = M/2$ tree nodes. It follows that the total length of the $M/12a$ paths grown is at most $M/2$. This verifies that the tree being considered has at most M nodes.

Now we can calculate the average edge length. Since each of the $M/12a$ paths connects a node in the chosen block to a node in some non-adjacent block, the total edge length in these paths is at least $(M/12a) \times \Omega(\sqrt{\log N}/a) = \Omega(M\sqrt{\log N}/a^2)$. Since the entire tree contains at most M edges, it follows that the average edge length of the embedding is $\Omega(\sqrt{\log N}/a^2)$. ■

4.6 Remarks

The embedding in section 4.4 achieves dilation at most 12. One edge of T corresponds to at most two edges of $B(T)$, each of which corresponds to two butterfly edges. In the embedding of the butterfly into the hypercube each butterfly edge corresponds to two edges of the hypercube. The redistribution algorithm adds at most four edges to the resulting path for a total of 12 hypercube edges. By combining the techniques of section 4.4 with those of section 4.2, we can reduce this to 6 or 7 with no increase in load or congestion.

It is also likely that we can improve the bound on congestion to $O(M/N \log N + 1)$ for hypercube embeddings by combining the techniques in section 4.4 with those of section 4.2. We suspect that this bound is tight for all on-line algorithms, but we

can prove a bound of $\Omega(M/N \log N + 1)$ only for deterministic on-line algorithms. Any M-node binary tree can be embedded off-line in an N-node hypercube with load $O(M/N + 1)$ and constant dilation and congestion.

Although we have not worked out the details, we suspect that our embedding algorithms also work for trees that can shrink from the top as well as grow and shrink from the bottom, and that they can be made to work for arbitrary trees of small degree. We also expect that our techniques will prove useful for finding embeddings in other networks, such as the shuffle-exchange graph.

Bibliography

[ALN] W. Aiello, T. Leighton, and M. Newman, "Routing on the Hypercube in $O(\log N)$ Bit Steps," *typescript*, 1989.

[BS] B. Becker and H.U. Simon, "How Robust is the n-Cube?," *Proc.* 27^{th} *Ann. IEEE Symp. on Foundations of Computer Science,* Oct. 1986, pp. 283 – 291.

[BC] S. Bhatt and J.-Y. Cai, "Take a Walk, Grow a Tree," *Proc.* 29^{th} *Ann. IEEE Symp. on Foundations of Computer Science,* Oct. 1988, pp. 469 – 478.

[BCHLR] S. N. Bhatt, F. R. K. Chung, J. W. Hong, F. T. Leighton, and A. L. Rosenberg, "Optimal Simulations by Butterfly Networks," *Proc.* 20^{th} *Ann. ACM Symp. on the Theory of Computing,* May 1988, pp. 192 – 204.

[BCLR] S. Bhatt, F. Chung, T. Leighton, and A. Rosenberg, "Optimal Simulation of Tree Machines," *Proc.* 27^{th} *Ann. IEEE Symp. on Foundations of Computer Science,* Oct. 1986, pp. 274 – 282.

[BH] A. Borodin and J.E. Hopcroft, "Routing, Merging and Sorting on Parallel Models of Computation," *Proc.* 14^{th} *Ann. ACM Symp. on the Theory of Computing,* May 1982, pp. 338 – 344.

[BI] S. N. Bhatt and I. Ipsen, "How to Embed Trees in Hypercubes," *Technical Report 443,* Yale University, 1985.

[C] M. Y. Chan. "Dilation-2 Embeddings of Grids into Hypercubes," *Technical Report UTDCS 1-88,* The University of Texas at Dallas, 1988.

[DHSS] D. Dolev, J. Halpern, B. Simons, and R. Strong "A New Look at Fault Tolerant Network Routing," *Proc.* 16^{th} *Ann. ACM Symp. on the Theory of Computing,* Apr. 1984, pp. 526 – 535.

[GHR] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg, "Optimal Embeddings of the FFT Graph in the Hypercube," *typescript,* University of Massachusetts, 1987.

[Gi] Giladi, *typescript*, 1989.

[Gr] J. Greene, "Configuration of VLSI Arrays in the Presence of Defects," Ph.D. dissertation, Stanford University, Stanford, CA, 1983.

[GG] J. Greene and A. Gamal, "Area and Delay Penalties for Restructurable VLSI Arrays," *JACM,* Vol. 31, No. 4, Oct 1984, 694 – 717.

[H] W. Hoeffding, "On the Distribution of the Number of Successes in Independent Trials," *Annals of Mathematical Statistics,* Vol 27, 1956, pp. 713 – 721.

[HJ] C. T. Ho and S. L. Johnsson, "Embedding Generalized Pyramids in Hypercubes," *Technical Report,* Yale University, 1988.

[HLN1] J. Hastad, T. Leighton and M. Newman, "Reconfiguring a Hypercube in the Presence of Faults," *Proc.* 19^{th} *Ann. ACM Symp. on the Theory of Computing,* May 1987, pp. 274 – 284.

[HLN2] J. Hastad, T. Leighton and M. Newman, "Fast Computation with Faulty Hypercubes," *Proc. of the* 21^{st} *Ann. ACM Symp. on the Theory of Computing,* May 1989, pp. 251 – 263.

[KLMRR] R. Koch, T. Leighton, B. Maggs, S. Rao and A. Rosenberg, "Work-Preserving Emulations of Fixed-Connection Networks," *Proc.* 21^{st} *Ann. ACM Symp. on the Theory of Computing,* May 1989, pp. 227 – 240.

[KZ] R.M. Karp and Y. Zhang, "A Randomized Parallel Branch-and-bound Procedure," *Proc.* 20^{th} *Ann. ACM Symp. on the Theory of Computing,* May, 1988, pp. 290 – 300.

[LL1] F.T. Leighton and C.E. Leiserson, "A Survey of Algorithms for Integrating Wafer-Scale Systolic Arrays," *Wafer Scale Integration,* edited by G. Saucier and J. Trilhe, IFIP, 1986, pp. 177 – 195.

[LL2] F.T. Leighton and C.E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Transactions on Computers,* Vol. C-34, No. 5, May 1985, pp. 448 – 461.

[P] N. Pippenger, "Parallel Communication with Limited Buffers," *Proc.* 25^{th} *Ann. Symp. on the Foundations of Computer Science*, Oct. 1984, pp. 127 – 136.

[Rab] M. O. Rabin, "Efficient Dispersal of Information For Security, Load Balancing and Fault Tolerance," JACM, to appear.

[Ran] A. Ranade, "How to Emulate Shared Memory," *Proc.* 28^{th} *Ann. Symp. on the Foundations of Computer Science*, Oct. 1987, pp. 185 – 194.

[LNRS] T. Leighton, M. Newman, A. Ranade and E. Schwabe, "Dynamic Tree Embeddings in Butterflies and Hypercubes," *Proc. First Ann. ACM Symp. on Parallel Algorithms and Architectures*, June 1989, pp. 224 – 234.

[VB] L. G. Valiant and G. J. Brebner, "Universal Schemes For Parallel Computation," *Proc.* 13^{th} *Ann. ACM Symp. on the Theory of Computing*, May 1981, pp. 263 – 277.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR 500	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-87-K-825
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS (*City, State, and ZIP Code*) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (*City, State, and ZIP Code*) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (*Include Security Classification*) Randomness and Robustness in Hypercube Computation	
12. PERSONAL AUTHOR(S) Mark Joseph Newman	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (*Year, Month, Day*) April 1991 15. PAGE COUNT 107
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB-GROUP	18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

In this thesis we explore means by which hypercubes can compute despite faulty processors and links. We also study techniques which enable hypercubes to simulate dynamically changing networks and data structures.

In chapter two, we investigate strategies for routing permutations on faulty hypercubes. We assume that each node or edge in the hypercube fails with constant probability and that failures are independent of one another. We describe a routing algorithm which successfully routes messages between working processors in O(log N) steps on an N-node faulty hypercube with high probability.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Carol Nicolora	22b. TELEPHONE (*Include Area Code*) (617) 253-5894 22c. OFFICE SYMBOL

☆U.S. Government Printing Office: 1985—507-047

19 a.

In chapter three, we develop techniques for reconfiguring hypercubes in the presence of faults. Again assuming constant probabilities of failure and the independence of faults, we show that a faulty hypercube can simulate a fault-free hypercube of the same size with only constant delay. We exhibit both deterministic and randomized algorithms for hypercube reconfiguration.

In chapter four, we turn our attention to the embedding of dynamically growing data structures in the hypercube. Among several results, we show that an arbitrarily growing binary tree with a maximum of M nodes can be embedded in an N-node hypercube with load O(M/N + log N) and dilation 2 with high probability.