Yale University
Department of Computer Science

Full Utilization of Communication Resources

David Saks Greenberg

YALEU/DCS/TR-860
June 1991

This work was supported in part by National Science Foundation grants MIP-8601885, CCR-8807426, by NSF/Darpa grant CCR-8908285, and by AFOSR grant 89-0382

Full Utilization of Communication Resources

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

David Saks Greenberg
December, 1991

Copyright © 1992 by David Saks Greenberg
ALL RIGHTS RESERVED

Acknowledgments

I might never have gotten through the frustrating early years of this work without the support of Josh Cohen Benaloh and David Wittenberg. Over the succeeding years the encouragement and comradery of Rob Bjornson, Mike Factor, Scott Fertig, Ching-Tien Ho, Pangfeng Liu, Ruben Michel, Nicholas Reingold, and Da-Wei Wang were invaluable.

It is impossible to measure the importance of one's advisor to a thesis. Sandeep Bhatt's insistence on precise and clear writing have undoubtedly made this thesis easier to understand. The remaining obfuscations are the result of my failure to rise to his standards. His technical guidance is also, of course, greatly appreciated.

Arny Rosenberg, in his usual understated way, suggested several avenues for exploration which were extremely fruitful. Charles Leiserson's comments at the 1989 SPAA were helpful in getting me past an early stumbling block. Jeff Westbrook's careful reading identified many sections requiring expansion and clarification. Dana Angluin, Marina Chen, Ilse Ipsen, and Lennart Johnsson supplied ideas and support along the way.

I'd also like to thank Betsy Hearns for her infinite patience with my administrative incompetence. Her help in copying and mailing papers and in unraveling the Yale bureaucracy made my time at Yale immeasurably more comfortable.

4.6 Disjoint Paths 37
4.7 Eigenvalues of the Hypercube 37

5 Efficient Embeddings 38
5.1 Pin limitations 38
5.2 Multiple-path Embeddings of Grids 40
5.2.1 Multiple-path embeddings of cycles 40
5.2.2 Embedding Cycles with Load 1 40
5.2.3 Load 2 Embeddings which Fully Utilize the Hypercube Links 44
5.2.4 Bounds on Width and Cost 45
5.2.5 Multiple-path embeddings of grids 46
5.3 Multiple-copy Cube-Connected-Cycles 47
5.3.1 Terminology 48
5.3.2 Embeddings of the CCC Network 48
5.3.3 The Multiple-copy embedding 49
5.3.4 Extensions 54
5.4 A General Technique 54
5.4.1 Complete Binary Trees 56
5.4.2 Arbitrary binary trees 57
5.5 Large-copy embeddings 58

6 Beyond Hypercubes 59
6.1 Cross-Product Decomposition 59
6.2 Multiple-copy Decompositions 66
6.3 Neighbor-Distinguishing Labellings 67
6.4 From Multiple-copies to Multiple-paths 67
6.5 k-dimension cubes 68

7 Wire lengths 70
7.1 Length-Weighted Design 71
7.2 Length-weighted Layouts for Trees 72
7.3 Length-weighted Layouts for Hypercubes 72

8 Conclusions 74
8.1 Beyond Network Comparisons 75
8.2 Open Questions 75
8.3 Final thought 78

Chapter 1

Introduction

The simulation of the air flow past the blades of a gas turbine required over four days on a Cray X-MP[34, 65]. Simulating the flow around the whole engine or entire airplane is well beyond the capacity of current computers. Replacing costly scale models with computer simulations will require an increase in the speed of computers by several orders of magnitude. However, the nanosecond switching times and submicron feature sizes of today's integrated circuits are approaching physical limits inherent in the speed of light and the size of molecules.

The ability to solve enormous problems, like the simulation of flow around an airplane, will require computers to consist of millions of independent, high-speed processors. A crucial component of these computers will be an interconnection network capable of channeling the torrents of data produced by these processors to their proper destinations. In an inadequate network communication bottlenecks will negate the advantage of having millions of processors. Any network which meets the communication demands of these massively parallel systems must allow as much of the data as possible to remain local to the processors and must make efficient use of the available physical resources.

In order to illustrate the need for both locality and efficient use of resources we look at a proposed machine for the mid 1990s, the Mosaic C machine currently being designed at California Institute of Technology[76, 77, 78]. This computer is designed to have 16,000 processors; each processor will be on its own chip along with a 64KByte local memory and a router capable of transmitting 80MByte off the chip per second. The processors will execute 14 million scalar instructions per second. Since each instruction consumes four bytes each processor will require 56MByte of data per second.

At first glance it appears that the off-chip data channels are fast enough to allow data to reside on or off the chip. However, the nominal speed of the channels is a poor measure of off-chip access time. Off-chip accesses require time to build packets with proper chip addresses, to multi-plex 16 bit operands onto narrower channels (8 or 4 bit in the Mosaic), to traverse the physical wires between chips,

to route through intermediate chips, and to wait when channels are assigned to other packets. It will be important to orchestrate data so that the communication resources are not overwhelmed.

Two types of locality are required to ease the communication bottleneck. First, most memory accesses must be to local memory. Johnsson[44] estimates that data access off chip for current multi-computers is two orders of magnitude slower than access on chip. Accesses to external boards are another order of magnitude slower. The basic operations of the computation must be partitioned across the processors so that the operands often reside in the memory of the processor executing the operation; the results of each operation should often be used by the processor executing it and operations which use the same data item should reside, as often as possible, on the same processor.

Second, when memory accesses are not to local memory they should be to the memory of chips which are nearby in the network. The load which a message adds to the network is proportional to the length of the path which it travels in the network. If each message travels through 100 chips (less than the average distance between chips in a mesh of 16,000 chips) then the average load on each network channel is 100 times greater than if each message went between two neighboring chips.

The stress on the network will only increase for machines which are more powerful than the Mosaic. The Mosaic processors execute one scalar instruction per chip per cycle; if technology allows a vector floating-point instruction per chip per cycle then the data requirements will increase. The current Mosaic aims at 16,000 processors but its design is extensible to a two million processor version[77]; the potential network distances and bottlenecks will be magnified.

Communication patterns

Fortunately many algorithms have natural data locality; the problem can be broken into subtasks so that each task requires data from only a few others. We take as an example the calculation of the air flow past a jet engine. The equations for the motion of the air cannot be simply solved algebraically. Instead, a finite difference equation is created over a lattice of points surrounding the engine. The equations relate the value at each point to the value of the surrounding points. Solutions are obtained by iteratively adjusting the value at each point, based on its previous values and the values of the neighboring points. (Figure 1.1 shows the standard *five point stencil* in which the value at each point depends on the previous values of its four grid neighbors.)

Two features of these calculations are important for parallel computing: the iterative nature and the dependence only on neighbors. The iterative nature of the problem means that if a grid point is assigned to a single processor then its successive values will always be in the local memory. The dependence on

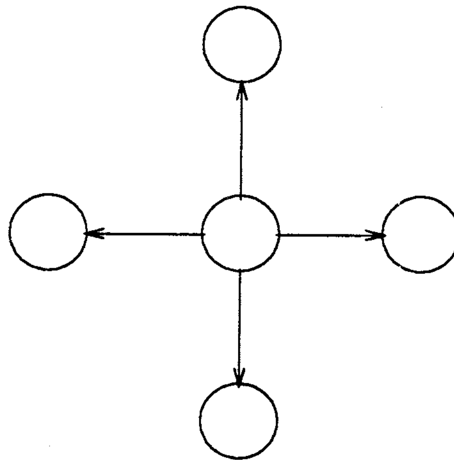


Figure 1.1: A five point stencil

neighbors means that if blocks of grid points are assigned to a single processor then for those points in the interior of the block the values of the neighbors will also be in local memory.

For example, suppose each block contained one hundred thousand points and each point were described by ten variables. Of the five million variables required to update these values only about twelve thousand would not be in the local memory; the remaining values would be in the four logical neighbors. A two order of magnitude reduction in communication results from utilizing the problem structure.

In order to capture the structure of an algorithm we define its *communication pattern.* First, the basic repetitive tasks of the algorithm, such as the computation of the next value of a grid point, are identified. A graph is then formed with the basic tasks serving as graph vertices. An edge is added between two vertices if the result of some operation in one task is used by an operation in the next iteration of the other task. The resulting communication graph gives a convenient representation of the algorithm's data dependencies.

We have already seen that one common communication graph is the grid. A one dimensional grid, or cycle, is the natural pattern for pipelined algorithms such as the Jet Propulsion Laboratory's Radar processing program[80]. Numerous fluid flow algorithms[8] use one, two, and three dimensional grids. Examples of problems using two dimensional grids include Reif and Storer's video compression algorithm[70] and the Perkin-Elmer group's program for displaying three dimensional graphics[60]. Components of the weather forecast systems of the National Weather Service [85] naturally form three dimensional grids.

Algorithms which follow the divide and conquer paradigm, such as the graph algorithms of Nayar, Prabhu, and Wright[57] or which evaluate strategy games (be they chess or economics) have communication patterns which form trees. The target detection program from Perkin-Elmer[60] uses the FFT communication pattern. Other algorithms use hexagonal meshes and pyramids. Problems which have simple communication patterns abound throughout computer science.

Whatever the communication pattern, it will be important to map it carefully onto the network. The existence of structure allows the possibility of mapping neighboring tasks to neighboring processors. If locality is not maintained an opportunity to significantly reduce the communication overhead may be squandered.

Interconnection Networks

If every pair of processors were connected by a fast link then all neighboring tasks would automatically be mapped to neighboring processors. A complete network connecting each of 16,000 processor-chips to every other is, however, impossible. Each chip has only at most a couple of hundred pins which can be used for data. Thus even a slow, single-pin link could not be made to all other processors. Since in many architectures five to ten or more pins are needed for each link, at most a few tens of bidirectional links will be possible. The network must be relatively sparse.

Several different networks have been proposed for parallel computers. The mesh interconnection has been used for the MPP[9], Mosaic C[30], and the J-Machine[59]. The butterfly is used for the BBN Butterfly[41] while Browning[22, 23] proposed a tree based machine. The Caltech Cosmic Cube[75], the TMC Connection Machine[36], the Intel iPSC[42], and the nCube all use a hypercube interconnect. New machines are being designed and the preceding list is per force only a representative sample.

The natural network for the two-dimensional air-flow example is a mesh; the communication pattern exactly matches the network. For each pair of communicating processors there is a direct link. Data can stream off the chips, across the links, and on to their destination chips.

On the other hand, suppose the air flow algorithm were executed on a machine with an interconnection network based on the complete binary tree. It is easy to show that at least one hundred of the processors on the left side of the tree will have a neighbor on the right side of the tree. All the data between processors on one side and their neighbors on the other must flow through the root of the tree. The resulting bottleneck on the links adjacent to the root will slow the computation down by two orders of magnitude compared to running on a mesh based machine.

The air flow problem mapped naturally to a mesh connected network. But

suppose our problem has a divide-and-conquer solution. When we attempt to run the algorithm on a sixteen-thousand-processor, mesh interconnection network many messages will have to be forwarded through one hundred or more intermediate processors[19]. Once again we will have delays due to the network.

Different algorithms seem to need different interconnection networks. But we do not want to build a different machine for each problem or communication pattern. We would like to have one network which could handle the communication for many different patterns.

Graph Embeddings

In order to compare networks as candidates for a general purpose computer we need to ask how well a particular network can service a given communication pattern. *Graph embeddings* provide a way of abstractly asking how suitable a network is for a communication pattern. The graph embedding framework reduces both the network and the communication pattern to abstract graphs. A mapping of one graph to the other corresponds to a way of assigning processes to processors and routing communication.

If a good embedding can be found then there exists a way of mapping the algorithm to the given network so that communicating processes are mapped to logically near processors. The communication delay due to the network is minimized since messages need not travel far. There have been many good embeddings found of common communication patterns into common networks. Meshes, trees, pyramids, FFTs, CCCs, and Butterflies have been shown to map easily to hypercubes[14, 32, 40, 82]. (In Chapter 4 we describe an embedding of the FFT, CCC, and Butterfly into the hypercube.) Trees have been shown to be embeddable in FFTs[13]. FFTs, CCCs, and Butterflies each easily can be embedded into each other.

Showing that a network is not good for a communication pattern is more difficult. We have already mentioned that there are no good embeddings of trees in meshes or vice versa. It has also been shown that meshes and mesh-like networks cannot be embedded in FFTs[13]. The lack of a good embedding certainly makes servicing the communication pattern difficult. It has been argued[47] that fast communication may still, in principle, be possible. By making copies of data (a practice limited by scarce memory) and allowing the mapping to change over time, the congestion inevitable in a direct embedding can sometimes by avoided. For example, the mesh communication pattern can be serviced on a FFT interconnect[47].

Nonetheless, the existence of a good embedding is a good indicator of whether or not the network can efficiently simulate the communication pattern.

The Boolean Hypercube

The consideration of graph embeddings as a measure of network flexibility has led to the boolean hypercube being considered a good general purpose network. In fact, several commercial machines have been based on this network[36, 42]. Besides allowing the embeddings of several common communication patterns (mesh, tree, and FFT) the hypercube has a nice recursive structure. A shortest path between any two processors is easily determined. Fast sorting and permutation algorithms for the hypercube have been developed[67, 86]. It is also possible to reconfigure a faulty hypercube to efficiently simulate the original unfaulty hypercube[35].

The hypercube does have some obvious disadvantages though. The degree of the hypercube nodes grows with the size of the network. Thus communication nodes designed for one size are not extensible to larger sizes. If a node is designed for a d-dimension hypercube then it cannot be used to build a hypercube with more than d dimensions and has wasted hardware when used for a hypercube with fewer than d dimensions.

Hypercubes are also hard to build. Planar wirings of a hypercube require area proportional to the square of the number of nodes and many long wires will always be necessary. As the desired size of the hypercube increases the complexity can become extreme. Currently all hypercube-based machines have at most twelve dimensions and some designers have argued that sixteen dimensions is about the largest size possible[11].

Physical Constraints

Physical constraints affect the cost and efficiency of all networks. The amount of chip area required for wires and routing elements, the lengths of wires between chips, the number of pins available to carry data off chips, the power required by the chips, the necessity of cooling the chips, and the necessity of a consistent clock signal are just a few of the design issues which physics imposes on the network design.

These physical considerations have, in part, led to arguments for the abandonment of high-dimension networks such as hypercubes in favor of simpler two and three dimensional meshes[27, 74]. The meshes are extensible, can be wired in a small planar area, and require no long wires. By building a network which conforms to physical constraints the hope is that faster transmission over inter-chip links will compensate for the loss in locality.

One goal of this thesis is to help give a theoretically sound underpinning to decisions concerning potential trade-offs between optimizing locality and minimizing the effects of physics.

Perhaps the best studied physical resource is chip area. For example, Dally

and Seitz[27, 74] look at the specific question of comparing members of the graph families called k-ary n-cubes. At one extreme is the two-dimensional mesh, at the other the boolean hypercube. They examine the question of mapping the network to a fixed area chip. The number of network edges which must cross the chip bisection grows with the network dimension. To accommodate a larger number of wires either the channels carrying the wires must be made physically thinner or the wires must multiplex across channels. It will not be possible to transmit as much data per time unit over edges of the higher dimension network.

They use the latency of an average message as a joint measure of locality and wire efficiency. Higher dimension means shorter, slower average paths. Since the delay due to slower data transmission increases more quickly than the savings due to shorter path lengths they conclude that lower dimension meshes are preferable.

Pin limitations

While acknowledging the importance of chip area we still wish to consider other resources. In particular, since large machines will consist of many chips the rate at which data can flow between chips is important. All data leaving a chip must use just a few (currently at most 200 to 300) pins. These pins are a precious resource which must be efficiently used.

This is not to say that efficient use of chip area is not important. Rather we ask how well we can optimize inter-chip communication if we ignore chip area issues. We hope to show ways in which the pins can be used more efficiently.

While the study of pin limitations is the basis of most of the results in this thesis we will also consider wire length; another important off-chip resource.

Resource Limited Embeddings

We have set ourself the goal of simultaneously maintaining locality and making efficient use of pin connections. We will start by extending the concept of graph embedding. The graph embedding model captures the structure of the network but not its physical costs. We will rectify this shortcoming by not treating all network edges equally. The physical constraints of network construction will provide us with limitations on edge capacities.

In our primary example, pin limitations, the capacity of a network link is tied to the number of pins available for it. If all networks are built using chips containing the same number of pins then the networks having higher degree will necessarily have fewer pins assigned to each communication link. Fewer pins means fewer bits can be transmitted in each clock cycle. We will demand that networks having these lower capacity links make better overall use of their links. A perfect, subgraph embedding in a logarithmic degree network is not as fast

as a similar embedding in a constant degree network. To be considered equally good the high degree network must provide many short paths.

We will show that, although the links of the hypercube are allowed logarithmically less capacity than those of constant degree graphs, the hypercube can still efficiently simulate meshes, trees, and FFTs. Our main tools will be two new types of graph embeddings: one in which multiple paths are assigned to each guest link and one in which multiple copies of the guest are mapped at once. These embeddings allow us to use all of the hypercube links, even though the guest graph has many fewer links than the hypercube. By using all the links we use all the pins; efficient, high utilization of the pins results.

1.1 Plan of the Thesis

The remainder of the thesis is organized as follows. The next chapter makes explicit the assumptions made in the thesis. Chapter 3 gives formal definitions while Chapter 4 contains useful facts about hypercubes. In Chapter 5 embeddings are developed which make efficient use of pins. Chapters 6 and 7 contain two variations on the resource limitation theme; the former extends the pin limitations to other networks besides the hypercube and the latter looks at a way of mitigating the delay due to long wires. Chapter 8 presents some conclusions and open problems.

Chapter 2

Background

In the last chapter we argued that many large scientific computations require computers consisting of interconnected, independent processors. Furthermore, these computers require networks which allow efficient use of available physical resources while avoiding frequent data reference to logically distant processors. In this chapter we examine more closely the type of algorithms that can benefit from our techniques and consider the effects of various physical constraints.

Our model of computation has a structure similar to data flow models. An algorithm consists of independently executable *tasks*. Each task has some, typically small, number of inputs and outputs. Our tasks will be iterative in nature. That is, they will repeatedly accept a set of inputs, compute values as a function of these inputs, and output these values. Examples of iterative algorithms include those in which the tasks represent slices of time (eg. the simulation of an artificial retina), those in which the tasks represent successive refinements of an approximation (eg. a finite element solution to an integral equation), and those in which the tasks represent successive versions of a pipelined algorithm (eg. the variational analysis of parameters to a system).

2.1 A simple algorithm; a simple machine

As an example we consider a variant of Conway's Game of Life. Each task is a cell on a torus. On each time step each cell sends a bit to its eight neighbors, a one if it is *alive* or a zero if it is *dead*. Each cell then sums its inputs; if the result is less than three it changes its bit to zero, if greater than five its bit becomes a one, otherwise its bit is unchanged. Each task is identical, sends equal-length messages to each neighboring task, and repeats the same operations on each step.

It is straightforward to design a machine to implement the Game of Life. The 'processors' are simple circuits which take nine one-bit inputs (the last value in the processor and its eight neighbors) and outputs the new value for the

processor. If all the circuits use a global clock then all the values are computed simultaneously. On one cycle all processors can latch the neighboring values and on the next cycle (or possibly several cycles) the processors can all compute the new value. Latching of neighboring values can recommence on the next cycle. The processors are synchronous, receive neighboring values in a single cycle, and repeat the same operations each round.

Both the algorithm and the machine are homogeneous. Since all the operations occur in lockstep there is no need for complicated communication schemes. However, real applications and real machines vary from these simple machines in many ways. In the remainder of this chapter we will examine ways in which real systems differ from this simple example.

2.2 Inhomogeneity

It is rare that all the tasks are exactly the same. The amount of computation required by each task may be different. Some tasks may require less information from their neighbors than others and thus the lengths of the messages may be different. We will assume throughout this thesis, however, that the tasks are the same length, as are the messages. This assumption is mostly a logical convenience since the asynchrony in the hardware, which we describe below, can mimic the effects of non-uniform tasks or messages.

An additional reason for assuming the tasks to be of equal size is that it helps to concentrate the attention on the communication network. When task or message size varies greatly then the overall computation speed may be limited by poor load balancing or long data paths, obscuring the limitations of the network.

The *execution* of identical tasks need not proceed in lockstep. Although some parallel computers, such as the TMC Connection Machine, employ a global clock, others allow each processor to run at its own rate. Thus the processors may complete the iterations at different rates. Due to the data dependencies no processor will be able to get too far ahead of its neighbors but they need not be synchronous.

Asynchronous execution of the processors presents a problem for the network. If messages are sent to processors which are not ready to receive them then the network may become clogged with undeliverable messages. We do not wish to be concerned with such side effects of processor speed. Thus we will place a logical buffer at each end of each communication link. A processor can proceed at its own speed, limited only by the availability of inputs.

Our model allows the processors to compute asynchronously while the network functions independently in a synchronous manner. At the beginning of each *communication step* each processor can inject a packet into the network, during the step each communication link can forward one packet, and at the end of the

step one arriving packet can be consumed by each processor. Our challenge is to move packets from source to destination as fast as possible. This will require choosing good routes for the packets; no packet should travel over many more communication links than necessitated by a shortest path and no packet should be delayed too often by having to wait for another packet to use its preferred communication link.

In order to better orchestrate the routing of messages and avoid message collision, the network will sometimes proceed in phases. Each phase will consist of some small number of communication steps. New messages will only be accepted from the processor buffers on the first step of each phase. Thus a message may be delayed by up to the length of the phase (eg. if it arrives on the second step.) However, in return for the delay in entering the network, the message will be given a stronger guarantee on its maximum delay in the network. Any message entering at the beginning of a phase will be delivered to its destination by the end of the phase.

Our networks are designed for the best case algorithms; ones in which the communication occurs in regular, iterative phases. Minor variation in task size or processor speed is smoothed away in order to allow optimal use of the network.

2.3 VLSI Area Cost

The inherent degree of synchrony in the system is important to the design of communication networks but does not directly impart an advantage to one network topology over another. Physical restrictions such as limited area, wire length, or number of pins do favor some networks over others. Some interconnection patterns are more expensive in terms of these resources than other patterns.

Perhaps the most studied physical cost is the area required to lay out a circuit or network. Large area layouts have several disadvantages. If the area is above some threshold (the maximum size of a VLSI chip) then additional costs will be incurred in order to divide the layout up into manageable size pieces (see Section 2.4). Even if the layout area is below this threshold, larger area layouts will be more costly and more prone to manufacturing defect. In addition, large layout area often also means some wires are long (see Section 2.5).

Thompson[83] defined a, now standard, model for determining area costs. He discovered that for many problems a lower bound could be shown for the product of the area of any circuit solving the problem and the square of its running time (AT^2). Thompson[83], Yao[89, 90], and Vuillemin[88] used bounds on the number of bits which must cross the mid-point of a circuit to achieve AT^2 lower bounds for boolean-valued functions and transitive functions. Networks which can solve these problems inherit their AT^2 bounds.

Rather than ask what is the minimum area to solve a problem we can ask

how much area is required for a particular network. The obvious lower bound on area is N, the number of processors. Some networks such as meshes and trees[55] can be constructed with this ideal layout area. Others such as shuffle-exchanges, FFTs, and hypercubes[51, 84, 88] require larger areas.

Another approach is to start with a fixed chip area and ask what is the best network which can be built with this area. Following Leiserson[54] *area-universal* networks have been investigated by several researchers[10, 18, 33]. The time to simulate any communication pattern on an area-universal network is not much greater than the time to simulate the same pattern on any other network of equal area.

Rather than look for a general network, Seitz[74] compares the performance of different networks when each is implemented on a fixed area chip with a fixed feature size. There are a limited number of channels which can be etched across the bisection. Networks requiring more wires to cross the bisection will necessarily be able to assign fewer channels per wire. For example, the N-node mesh can be assigned N times as many channels per wire as the N-node hypercube. Theoretically, the mesh can send N times as many bits as the hypercube over each wire in a single time step.

Layout area does not tell us everything about communication though. A good layout area does not guarantee that all other factors will also be good. In addition, the actual cost of VLSI chip area has been constantly decreasing. An inefficiency in layout area may not cost as much as an inefficiency elsewhere. In this thesis we examine networks which are so large that even an area N layout will not fit on a single chip; in fact we will assume that each processor is on its own chip. In such cases the costs of communicating off-chip can dominate the costs of area inefficiency.

2.4 Packaging Constraints

Once a multi-computer becomes large enough that it must be implemented on several chips, and even on multiple boards, a new cost becomes apparent. The number of pins over which data can leave the chip is severely limited. Current designs such as the C Mosaic[78] and J machine[59], which are optimized for communication power, are able to use only about one hundred pins off each chip for data. The scarcity of pins limits the chips to at most six bidirectional links, each eight bits wide. For a twelve dimension hypercube the links would have to be only half as wide. The resulting trade-off between number of links and width of links will be a central theme in this thesis.

At the board level the data squeeze is even more evident. The J machine has sixty-four processors per board, each requiring fifteen pins per port. The routing of the approximately one thousand wires to connectors at the edge of

the board is only barely possible. The designers of the J machine decided to use new elastomeric connectors which allow connections to be made directly from the middle of one board to the middle of the board above or below it in a stack. These connectors also significantly reduce the lengths of the wires. Even with this new technology the limited number of interchip connections remains a critical bottleneck.

Noakes and Dally note several other packaging issues for the J Machine[59]. An important consideration is cooling. The proposed four thousand processors each dissipates 1.5W and the external memories contribute another .45W per processor. All this heat required a careful spacing of the boards to allow sufficient cooled air to remove the excess heat.

Another problem is the delivery of power and ground signals. They must be routed among the processors so as not to couple with the data signals. The clock signal for a synchronous system is an even greater problem. Noakes and Dally employ an elaborate system of delay lines under the active control of a host to keep the clock signal aligned.

Cooling and power/ground wiring are common to all networks so they probably do not give differential advantage to one network over another. The necessity of clock alignment is an added problem for synchronous networks but will be ignored in this thesis.

The limit on the number of pins leaving a chip, however, imposes a clear tradeoff in network design. The more links there are between processors the fewer pins can be used for each link and thus the slower the link. This basic tradeoff will be an essential part of the results in Chapter 5.

2.5 Wire Length

Both attempts to limit layout area and packaging decisions are linked with the desire to keep wire length short. Long on-chip wires will necessitate large layout area while a requirement for short wires complicates packaging. In addition, long wires can slow the system due to the delay in transmission across them.

Wire length, the distance a signal must travel between processors, affects messages independently of the other message traffic. The time required to transmit a bit along a wire depends on the wire's length. Ideally all wires would be short; all communicating processors would be near to each other. Unfortunately this is often impossible in practice. The layout of a complete binary tree on N nodes requires some wires be of length at least $\sqrt{N}/\log N$[56]. Many other networks such as the shuffle exchange, the FFT, and the hypercube contain trees as subgraphs and thus also must have long wires. As with area, two-dimensional meshes can achieve the lower bound of unit wire length.

The cost of long wires is not simple to model. Various physical models lead

to different dependencies[20, 26, 69]. At least three physical components must be considered: speed of light, capacitive effects, and resistive effects. If all are combined the wire delay varies with at least the square of distance. However, hardware techniques such as repeaters and exponential horns can significantly reduce the delay. Thus wire delay is commonly taken to vary logarithmically in its length (mostly capacitive delay) or linearly in its length (mostly resistive or speed of light). Sometimes the constraints on cycle time due to other factors is great enough that variations in wire delay can be ignored and the wire delay is considered to be constant.

A complicating factor in the modeling of wire delay is the possibility of using the wire as a transmission line and piping multiple signals along the line. In this case the significant factor becomes the minimum distance between signals rather than wire length.

In Chapter 7 we will consider one possible way to mitigate the effects of wire length; more pins are used for the longer wires. In the chapters preceding that we will, however, ignore the effects of wire lengths and assume that switching costs out-weigh wire delays.

2.6 Costs of Routing Complexity

Quantifying the costs of one routing scheme versus another is difficult. There are many factors involved including at least: switch size, amount of decision logic, queue and other memory size, and message length overhead. Decisions on the type of routing scheme will affect the delay experienced by messages and the amount of area required for the network components.

As the number of edges incident to each node increases, the ability to route the inputs on the incoming edges to an arbitrary permutation of the outgoing edges requires a larger switching network. If a full cross-bar switch is used the area required can be significant; if a smaller switching network is used then significant delays may occur. In this thesis we will assume that the on-chip delay to switch signals between incoming and outgoing edges is not significant. Obviously this gives a small preferential advantage to higher degree networks such as hypercubes.

Some routing schemes (eg. Fluent[67], Adaptive Sorters [48, 58]) make decisions on where an incoming message will be routed based on the contents of the message. The more complicated the decision the larger the logic required to implement it. In addition, if queueing is required then space must be allocated for the queues. The time spent waiting in queues may be substantial. Several researchers [46, 66] have examined the question of minimizing queue size. Certainly small size queues are preferable but even a small queue may have significant associated cost.

All of these routing problems are undoubtedly important. We will, whenever

possible, endeavor to make sure that our embeddings require as little queueing and switching as possible. We will not, however, explicitly include these issues in our models.

2.7 Working Assumptions

In this section we summarize some key assumptions made in the first two chapters.

One Chip per Node Each processor, along with its memory and switching/routing circuitry, is assumed to fit on a single chip. This assumption allows us to concentrate on inter-node communication. All communication is reduced to the routing of message packets between nodes through a sparse network.

Pin Limitations The number of bits in a packet sent across a communication link is inversely proportional to the degree of the origin node. If each chip has $2W$ pins then a degree-d node can send and receive W/d-bit packets along each link in one communication step. By fixing the number of pins per chip the relation between packet size and number of incident links gives us our primary example of the effects of resource limitations on network comparisons.

Wire Length Throughout Chapters 4, 5, and 6 we assume that one bit of information can be transmitted along each wire in a time step, independent of the wire's length. In Chapter 7 we consider a model in which the time to transmit a bit varies with the length of the wire and look at one way to mitigate the effects of wire length delay.

Communication Synchrony We assume that inter-node communication occurs in discrete phases. At the beginning of each phase each processor can inject a message into the network. The use of synchronous *communication steps* allows us to decouple the network from the asynchronous processors and thus concentrate on the network.

Need for Data Locality Throughout this thesis our emphasis will be on algorithms with natural, iterative parallelism. We assume that it is important for the network to capture the inherent data locality of the algorithms. The ability of the network to maintain locality reduces the load on the network and thus enables larger, more communication intensive versions of these problems to be solved.

Chapter 3

Definitions

3.1 Graph Embeddings

Both the interconnection networks of parallel computers and the communication patterns of parallel algorithms can be represented as graphs. The processors and processes become vertices; while communication links and pairs of communicating processes become edges. An embedding of the communication graph into the network graph captures the difficulties of assigning processes to processors and of routing messages. All the complicating physical details are removed.

The physical details may, however, be important to network design. An important part of this thesis will be showing how to add some physical issues back into the model. But we must first define the standard graph embedding.

> An *embedding* of a guest graph, $G = (V, E)$, into a host graph, $H = (W, F)$ consists of: a one-to-one vertex map $\eta : V \rightarrow W$ and an edge map μ which assigns each edge $(u, v) \in E$ to a path in H from $\eta(u)$ to $\eta(v)$.
>
> When $|V| > |W|$ we allow many-to-one mappings, but restrict the mapping so that each host vertex is the image of no more than $\lceil |V|/|W| \rceil$ guest vertices.

An embedding is an attempt to fit one graph into another. If the guest graph is a communication pattern and the host graph is an interconnection network then an embedding tells how to assign the processes to processors and how to route the messages. A good embedding should result in low communication overhead. Similarly, if both graphs are interconnection networks then an embedding tells how to simulate one network on the other; guest processors are mapped to the host processor which will simulate them and communication links are mapped to routes over which corresponding messages will be sent. When the guest graph contains more vertices then the host graph then the mapping must be many-to-one. However, to avoid trivial solutions and to ensure that the computation load

is balanced, we insist that no host vertex is the image of more than its fair share of guest vertices.

Measures of Embedding Quality

As a guide toward determining which embeddings are good we use three simple measures.

> The *dilation* of an edge $e \in E$ is the length of the path $\mu(e)$, and the dilation of an embedding is the maximum dilation of any edge in G.
>
> The *congestion* of an edge $f \in F$ equals the number of edges in G whose images contain f. The congestion of an embedding is the maximum congestion of any edge in H.
>
> The *load* of a processor is the number of processes mapped to it. The load of an embedding is the maximum load of any processor.

Dilation, congestion, and load are detailed measures of embeddings. Often we prefer a single measure of quality – in terms more closely related to networks and communication patterns. Rather than edges of a guest graph we consider all the messages delivered in a single communication phase; instead of host graph edges we consider the ability to transmit a standard sized message (called a packet) in a single communication time-step. Our single measure tells us how many host communication time-steps are required to complete one communication phase. We define this measure formally as *packet cost*.

> The *one-packet cost* of an embedding of G into H is the minimum number of time steps necessary for H to simulate the transmission of one packet along each edge of G. It is assumed that at most one packet can cross each edge of H in one time step.

Simple bounds on the packet cost in terms of dilation and congestion (assuming the load to be one) are easy to derive. In a dilation-d, congestion-c embedding at least d steps are required for those packets which must traverse an edge dilated to length d and at least c steps are necessary for the c packets to cross the edges with congestion c. On the other hand c communication steps suffice for every packet to move forward at least one edge; so, in cd steps every packet can be delivered to its destination. Thus $\max\{c, d\} \leq$ packet cost $\leq cd$.

In order to keep packet cost low it is necessary to arrange the timing of the packets so that no packet is delayed frequently at congested edges. Leighton, Maggs, and Rao [52] showed that the routing can always be orchestrated so that the one-packet cost is at most a constant factor times the sum of the dilation

and the congestion. Unfortunately, their result does not always yield a method of determining the optimal routing plan. Thus, we will sometimes want to explicitly define the timing of the packets over their routes, thereby establishing low packet cost.

Sometimes our embeddings will specify several paths for each guest edge, or there may be the possibility of pipelining several packets along a single path. In these cases the one-packet cost does not adequately define the goodness of the embedding. Imagine that the guest edges require the communication of long messages, each containing p packets. Each host edge can still only transmit one packet per time step. The relevant measure becomes the number of time steps necessary to deliver all the packets for all the messages of one communication phase. Clearly p times the one-packet cost will suffice; but often we can do much better.

> The ***p*-packet cost** of an embedding of G into H is the minimum number of time steps necessary for H to simulate the transmission of p packets along each edge of G.

Previous Graph Embeddings

The study of graph embeddings has is roots in several areas: the investigation of graph/subgraph isomorphisms, the search for good VLSI layouts, and the construction of data structures (see [72] for more details). Although our focus is on the use of graph embeddings to map a communication pattern into a network the techniques and results of these earlier works are still relevant. An important tool for graph embeddings is the graph separator; the planar separator theorem of Lipton and Tarjan led to embeddings of planar graphs in complete binary trees and several researchers have made extensive use of separators to produce graph embeddings (eg. [17, 18, 19, 51]).

In Figure 3.1 we list a few of the previous graph embedding results concerning networks. Several of these embeddings will be used as tools within the thesis. The ability to ***square*** a mesh[4, 29, 49, 53] (ie. map a rectangular mesh to an equal area square mesh) was originally studied as an aid to VLSI design. Often it is easy to design a circuit as a rectangle but better to fabricate square chips; a mapping of the rectangular mesh to the square mesh provides a way of automatically converting the design to a chip layout. Our interest in squaring meshes derives from the necessity of equalizing resource use.

A second set of embeddings which will be useful to us are the embeddings of arbitrary trees into complete binary trees and of complete binary trees into FFTs[13, 15]. We will first find efficient embeddings of FFT-like graphs (FFTs, butterflies, and CCCs) and then compose them with these earlier embeddings in order to produce efficient embeddings of arbitrary trees.

GUEST	HOST	References
mesh	square mesh	[4, 29, 49]
tree	complete binary tree	[15]
complete binary tree	FFT	[13]
mesh	hypercube	[25, 71]
tree	hypercube	[14, 16]
FFT,CCC	hypercube	[32]
pyramid	hypercube	[39, 82]

Figure 3.1: Graph Embeddings

Multiple–path embeddings

Standard embeddings, which assign a single path to each guest edge, may not capture the entire ability of one graph to service the communication inherent in another graph. The host graph may, in fact, be able to accommodate multiple paths for each guest edge – without much greater dilation or congestion than mappings which provide just a single path per edge.

> A *width–w* embedding of G into H is a one-to-one embedding in which each edge of G is mapped to w edge-disjoint paths in H. The congestion of an edge $f \in H$ equals the number of edges in G whose image paths contain f. The congestion of the embedding equals the maximum congestion among all edges in H. (The definitions of dilation and load are unchanged.)

Multiple–copy embeddings

Rather than allowing multiple paths per guest edge we may choose to embed multiple copies of the guest graph. Multiple copies will inevitably increase the load; but, it may be possible to route the paths of all the copies so that the overall congestion is not much greater than an embedding of a single copy.

> A *k–copy embedding* of G into H is a collection of k one-to-one embeddings of G into H. Since each embedding is one-to-one, each node of H can host up to k nodes, one from each copy of G. The congestion of an edge $f \in H$ in a k–copy embedding equals the sum, over all embeddings, of the congestion on f. The *edge-congestion* of a k–copy embedding is the maximum congestion on any edge in H.

3.2 Common Networks

Throughout the remainder of this thesis we will often refer to several common interconnection networks. In this section we formally define each network and give a few well-known facts about each. The facts should serve to give some idea of why the networks have become popular as well as to aid us in our later proofs.

The networks which we describe will be the the mesh/torus (sometimes called grid), the tree, and the hypercube, along with its constant degree derivatives (the Fast Fourier Transform Network (FFT), the Cube-Connected-Cycles (CCC), and the Butterfly).

3.2.1 Mesh networks

Perhaps the simplest network is the grid or mesh. Processors are arranged in rows and columns with each processor connected to its four nearest neighbors (sometimes denoted as North, East, South, and West). Formally the mesh is defined as follows:

> The vertices of the $\ell \times w$ *mesh* form an array $\langle i,j \rangle$ where $0 \leq i < \ell, 0 \leq j < w$. The edges are divided into two sets: the rows R and columns C. The sets R and C are defined as follows:
>
> $$\begin{aligned} R &= \{(\langle i,j \rangle, \langle i+1,j \rangle) \mid 0 \leq i < \ell - 1, 0 \leq j < w\} \\ C &= \{(\langle i,j \rangle, \langle i,j+1 \rangle) \mid 0 \leq i < \ell, 0 \leq j < w-1\} \end{aligned}$$
>
> For a wrapped mesh or *torus* the sets R and C are defined as:
>
> $$\begin{aligned} R &= \{(\langle i,j \rangle, \langle i+1 \pmod{\ell}, j \rangle) \mid 0 \leq i < \ell, 0 \leq j < w\} \\ C &= \{(\langle i,j \rangle, \langle i,j+1 \pmod{w} \rangle) \mid 0 \leq i < \ell, 0 \leq j < w\} \end{aligned}$$
>
> The vertices of a *k-dimensional,* $L_1 \times L_2 \times \ldots \times L_k$ *mesh* are k-tuples with ith component $\in [0, L_i)$. There are k edge sets, the ith consists of edges between pairs of vertices differing by one in the ith component (modulo L_i for tori).

The layout and wiring of a two-dimensional mesh is trivial. The *bisection width*, the number of edges which must be cut to disconnect the network into two approximately equal pieces, of an N vertex square mesh is $\sqrt{N}$. The *diameter*, the longest distance in the graph between two vertices, is $2\sqrt{N}-1$. The relatively

large diameter means that unless locality is maintained some processors may have to send messages over long paths.

The mesh is the natural communication pattern for many algorithms, including some image processing and discrete method approximations of partial differential equations. Its simple structure has led to the development of many algorithms tailored to its pattern (eg. matrix algorithms and sorting algorithms[58, 73]).

Some algorithms require slight variations to the mesh. For example some partial differential equation solvers require the addition of diagonal edges (Northeast, etc.) Some sorting and matrix algorithms require *wrapped* edges connecting vertices at opposite sides; thereby forming a torus. The addition of these extra edges does not make the layout or wiring of the mesh significantly more difficult. If no new wires are added it is still possible to quickly simulate these new networks on the mesh.

Computational problems in physics are often three dimensional; in fact, relativistic equations require four dimensions and some quantum dynamic systems use six. Some work has been done on building three dimensional mesh interconnects in three physical dimensions[59]. When the dimension of the mesh interconnect exceeds the physical dimension of construction then the layout and wiring advantages are lost.

The study of mesh-based algorithms has a rich history. Leiserson[53], Aleliunas and Rosenberg[4], Atallah and Kosaraju[49], and Ellis[29] have examined maps of rectangular meshes into square meshes. Chan[24, 25], Bettayeb, Miller, and Sudburough[12], Ho and Johnson[38], and Stout and Wagar[82] have looked at mapping meshes into hypercubes.

3.2.2 Trees

Another simple network is the tree.

> A *degree-d tree* has a vertex set consisting of a prefix closed set of strings over the alphabet $\{1, 2, \ldots, d\}$ (including the empty string). There are edges between vertex w and all vertices of the form $wx, x \in \{1, 2, \ldots, d\}$.
>
> A height-h *complete binary tree* is the special case where $d = 2$ and the vertex set includes all strings of length less than h. All strings of length i are said to be on *level* i. There are 2^i vertices in level i and $2^h - 1$ vertices in the tree.

The tree is also simple to lay out on the grid, requiring area only linear in the number of nodes. However, some N-vertex trees require wires of length $\Omega(\sqrt{N}/\log N)$. Most trees of interest to this thesis have diameter $O(\log N)$ and

all such trees require long wires. All trees have bisection width at most $\log N$ while complete binary trees have bisection width one. As a network, trees never require communications to travel very far but tend to suffer from bottleneck congestion near their root.

Many variants of trees have been proposed to alleviate the congestion at the root including fat-trees[33, 54] and X-trees[28]. We will not discuss these in this thesis.

Many algorithms, such as those based on divide-and-conquer or on mini-max games, use trees as their communication structure. These algorithms are often most efficient when the trees have limited depth.

3.2.3 Hypercube-based networks

While the mesh and tree are easy to lay out on the plane, the former has high diameter and the latter has low bisection width. The networks based on the binary hypercube combine low diameter with high bisection width. Each of these networks has diameter $O(\log N)$, matching that of the tree. The hypercube has edge bisection-width $\Theta(N)$ while the FFT, Butterfly, and Cube-connected-cycles each have edge bisection width $\Theta(N/\log N)$.

Boolean Hypercubes

The boolean hypercube has been used as the network for several commercial parallel machines [36, 42]. Mathematically it is very simple and its extensive symmetry gives it many useful properties. Unfortunately its relatively high degree and the large area and wire length required to lay it out on the plane are clear disadvantages compared to the mesh and tree. Nonetheless the hypercube will be our most common host network. Many of its useful properties are discussed in Chapter 4.

> The *k-dimensional hypercube*, Q_k, consists of 2^k nodes with distinct k-bit addresses. There is a directed edge (u, v) if and only if the addresses of u and v differ in exactly one bit position. (See Figure 3.2) An edge between two nodes that differ in the ith bit is said to lie in the ith dimension.*

The FFT network

The high degree of the hypercube can be avoided by expanding each hypercube node into a chain of $\log N + 1$ subnodes. If the subnodes are connected in a chain

*We define the hypercube as a directed graph; thus each communication link is modeled as a directed edge.

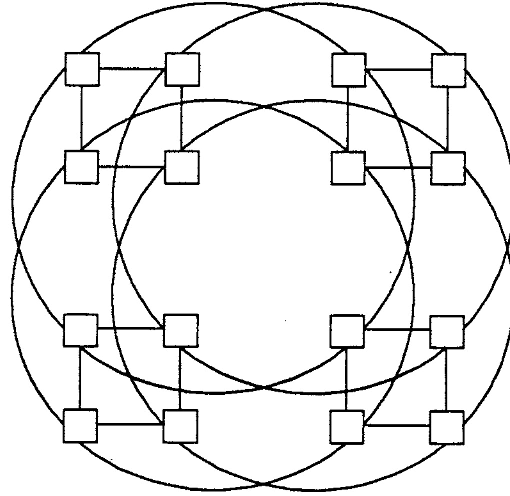


Figure 3.2: The 4-dimensional Hypercube

and the dimension-i edges of the hypercube nodes are assigned to edges between the ith and $i+1$st subnode in the corresponding chains then the FFT is formed. The FFT gets its name from the fact that it is also the communication pattern of the Fast Fourier Transform algorithm. This pattern is related to the more general convolution pattern used in signal processing algorithms.

The nodes of the n-stage FFT network, $F(n)$, are divided into $(n+1)$ *levels* and 2^n columns. (See Figure 3.3) Each vertex is addressed $\langle l, c\rangle$ where $0 \leq l \leq n$ and $0 \leq c < 2^n$. The edges of the FFT are divided into two sets: the *straight-edges* S, and the *cross-edges* C. The sets S and C are defined as follows:

$$
\begin{aligned}
S &= \left\{(\langle l, c\rangle, \langle l+1, c\rangle) \mid 0 \leq l < n, 0 \leq c < 2^n\right\} \\
C &= \left\{(\langle l, c\rangle, \langle l+1, c \oplus 2^l\rangle) \mid 0 \leq l < n, 0 \leq c < 2^n\right\}
\end{aligned}
$$

We call the edges between nodes at levels l and $l+1$ *level-l* edges. The operator $\oplus$ denotes bitwise exclusive-or of its two arguments. From node $\langle l, c\rangle$ the level-l straight-edge simply increments l, while the level-l cross-edge complements the lth bit of c in addition to incrementing l.

Fact 1 For every pair of columns c_1 and c_2, there exists a unique path of length n connecting nodes $\langle 0, c_1\rangle$ and $\langle n, c_2\rangle$. The lth edge in the path is the level-l

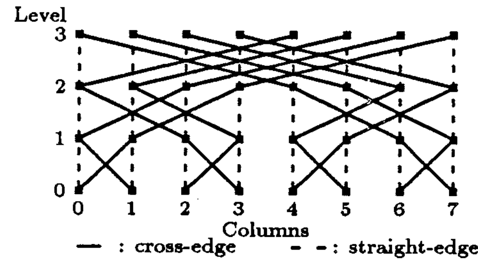


Figure 3.3: The 3-Stage FFT

cross-edge if c_1 and c_2 differ in bit l and the level-l straight-edge otherwise. (Bit 0 is the lowest order bit).

The Cube-connected-cycles Network

If each hypercube node is expanded into a length-($\log N$) cycle and the dimension-i edge is replaced by an edge between the ith copy of the corresponding nodes then the CCC is formed[63]. The CCC has fewer nodes than the FFT and undirected degree 3 rather than 4. They can, however, be easily simulated on each other.

Like the FFT the vertices of the CCC are divided into levels. The n-stage CCC network has n *levels* and 2^n columns. (See Figure 3.4) Each vertex is addressed $\langle l, c\rangle$ where $0 \leq l < n$ and $0 \leq c < 2^n$. The edges of the CCC are divided into two sets: the *straight-edges* S, and the *cross-edges* C. The sets S and C are defined as follows:

$$\begin{aligned} S &= \left\{(\langle l, c\rangle, \langle l+1 \pmod n, c\rangle) \mid 0 \leq l < n, 0 \leq c < 2^n\right\} \\ C &= \left\{(\langle l, c\rangle, \langle l, c \oplus 2^l\rangle) \mid 0 \leq l < n, 0 \leq c < 2^n\right\} \end{aligned}$$

We call the edges between two nodes at level l or a node at level l and one a level $l+1$ *level-l* edges. The operator $\oplus$ denotes bitwise exclusive-or of its two arguments. From node $\langle l, c\rangle$ the level-l straight-edge simply increments l, while the level-l cross-edge complements the lth bit of c.

The Butterfly network

The butterfly network has edges similar to the FFT and subnode cycles like those of the CCC. It has served as the network for several research parallel machines[41].

The nodes of the n-stage Butterfly network, $B(n)$, are divided into n *levels* and 2^n columns. (See Figure 3.5) Each vertex is addressed $\langle l, c\rangle$ where $0 \leq l < n$ and $0 \leq c < 2^n$. The edges of the Butterfly are divided into two sets: the *straight-edges* S, and the *cross-edges* C. The sets S and C are defined as follows:

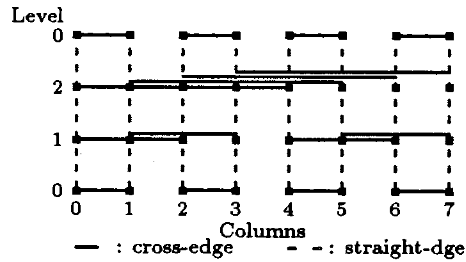


Figure 3.4: The 3-level CCC (level 0 shown at top and bottom)

$$\begin{aligned} S &= \left\{(\langle l,c\rangle,\langle l+1 \pmod{n},c\rangle) \mid 0 \le l < n, 0 \le c < 2^n\right\} \\ C &= \left\{(\langle l,c\rangle,\langle l+1 \pmod{n},c\oplus 2^l\rangle) \mid 0 \le l < n, 0 \le c < 2^n\right\} \end{aligned}$$

We call the edges between nodes at levels l and $l+1$ *level-l* edges. The operator $\oplus$ denotes bitwise exclusive-or of its two arguments. From node $\langle l,c\rangle$ the level-l straight-edge simply increments l, while the level-l cross-edge complements the lth bit of c in addition to incrementing l.

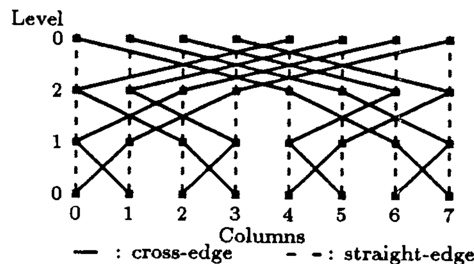


Figure 3.5: The 3-Stage Butterfly (level 0 shown at top and bottom)

3.3 Cross-Products

Once we have graphs with interesting properties it is often nice to combine them to form new graphs. One particularly useful technique for combining graphs is the cross-product. In fact many of the graphs already described can be viewed as cross-products of simpler graphs. There are many slightly different definitions of the cross-product. The one used in this thesis is the following:

> The *cross-product* of two graphs $G = (V,E)$ and $H = (W,F)$ is denoted $G \times H$ and consists of vertex set $V \times W = \{\langle v,w\rangle \mid v \in V, w \in W\}$

and edge set $\{(\langle v, w_1\rangle, \langle v, w_2\rangle) \mid v \in V, (w_1, w_2) \in F\} \cup \{(\langle v_1, w\rangle, \langle v_2, w\rangle) \mid (v_1, v_2) \in E, w \in W\}$. By analogy with multiplication, the graphs G and H are referred to as *factors* of $G \times H$.

Fact 2

The cross-product of the length-L path (resp. cycle) and the length-W path (resp. cycle) is the $L \times W$ grid (resp. torus).

The cross-product $Q_n \times Q_m$ is equal to Q_{n+m}.

The complete binary tree of height $2h-1$ is a subgraph of the cross-product of the height-h complete binary tree with itself.

Chapter 4

Properties of Hypercubes

As mentioned in the previous chapter the boolean hypercube is a very flexible network; it is capable of hosting a variety of networks via efficient embeddings. In this chapter we collect some general properties of the hypercube which are useful to our later results and also for hypercube algorithms in general. Some of these properties are old enough to be considered folklore, others are new to this thesis.

4.1 Graycodes

A classic problem is to find an ordering of consecutive binary numbers such that successive values differ in exactly one bit. The problem of creating such one-bit-change-at-a-time sequences is isomorphic to the problem of finding Hamiltonian paths and cycles in the binary hypercube. Each change of a single bit corresponds to traversing an edge of the hypercube. Including each of the binary numbers is equivalent to visiting each hypercube node. Solutions to the problem have come to be called *gray codes*, so named because Gray discovered one of the first such sequences[71].

There are, in fact, many possible gray code sequences for any hypercube. The study of how many different codes exist for each size hypercube (under various definitions of 'different') has a long history but is outside the scope of this thesis. We will concentrate on two particular gray codes: one old and frequently used called the Binary Reflected Gray Code (BRGC for short); and one new, with special properties, which we call the Even Distribution Gray Code.

We start with some terminology:

> A *gray coded* list is any list for which each two successive words differ in a single bit. If the first and last words also differ in a single bit then the list is a gray coded cycle. For example, $000, 001, 011, 111$ is gray coded list.

Any gray coded list of the 2^n n-bit binary words is called a *gray code* on n bits. For example, 000, 001, 011, 010, 110, 111, 101, 100 is a gray code on three bits.

The sequence of bit positions changed between each successive pair of words in a gray code is called a *gray code transition sequence.* It is simple to convert a gray code specified by a list of its words into a transition sequence along with the first code word and *vice versa.* When no first code word is given with a transition sequence then the first code word is assumed to be the all zeros word (of length equal to the ceiling of the log of the length of the transition sequence). The gray code transition sequence for the previous example is $0, 1, 0, 2, 0, 1, 0$. For initial code word 001 the gray code becomes 001, 000, 010, 011, 111, 110, 100, 101.

We denote the ith element of the n-bit transition sequence $(1 \leq i < 2^n)$ by $G_n(i)$. Note that although there are 2^n code words in a gray code there are only $2^n - 1$ elements of a transition sequence. The bit changed to go from the last code word to the first is uniquely determined by the previous elements.

Given a transition sequence G_n denote the implied gray code as H_n where $H_n(0) = 0^n$, and for $0 \leq i < 2^n$, $H_n(i+1)$ equals $H_n(i)$ with bit $G_n(i)$ complemented.

4.1.1 Operations on Gray Codes

Given one gray code it is easy to produce others via simple reflections and rotations of the hypercube. Each reflection is equivalent to choosing a node to be labeled 0^d and a rotation is just a naming of the d dimensions. A Hamiltonian cycle in the hypercube must remain a Hamiltonian cycle regardless of which node is labeled 0^d and what names are given to the dimensions. Equivalently, a gray code must remain a gray code after any bitwise-exclusive-or of all code words with a fixed code word and/or any permutation of the bit positions of all code words with a fixed permutation.

Lemma 1 *For all $0 \leq w < 2^n$ and all permutations $\pi \in S_n$ the sequence produced by applying a fixed reflection $\oplus_w$ or rotation π to each code word in a gray code produces a new gray code.*

The fact that transition sequences do not explicitly refer to the length of the code words makes them useful in stating recursive specifications of gray codes. We next give recursive definitions of two gray codes.

4.1.2 Binary Reflected Gray Code

The binary reflected gray code sequence derives its name from the fact that the second half of the code is identical to the first half in reverse (or reflected) order with the high order bit complemented. The transition sequence is defined recursively as follows:

$$\begin{aligned} G_1 &= 0 \\ G_n &= G_{n-1} \circ n-1 \circ G_{n-1} \\ \text{where } \circ &= \text{string concatenation.} \end{aligned}$$

Thus the three bit transition sequence is $G_3 = 0102010$ and the three bit code beginning with 000 is 000, 001, 011, 010, 110, 111, 101, 100.

4.1.3 Even Distribution Gray Code

When the gray code actually corresponds to a cycle within the hypercube it is sometimes desirable for the number of edges used in each dimension to be approximately the same. Vickers and Silverman[87] give additional reasons for finding such even distribution codes (which they call muddy). Exactly equal distribution is possible only when the number of dimensions is a power of two. Vickers and Silverman used a heuristic search to discover optimally evenly distributed codes for up to eight dimensions. We present a new, explicit method of producing codes in which no dimension is used more than three times as often as the average.

The idea behind these gray codes is to build up from copies of a smaller gray code just as in the BRGC but to permute the dimensions used in the second copy so as to equalize the distribution. The exact permutation used will be defined by the following sequence of *shuffle-like* permutations, σ_i. Define $f(i) = 2^i + i$, We denote the identity permutation as I and specify all other permutations as products of transpositions.

$\sigma_0 = I$,

$\forall i \geq 0 \; \sigma_{f(i)} = I$,

$\forall i \geq 0 \; \sigma_{f(i)-1} = \prod_{a=0}^{2^{i-1}} (a, a + 2^{i-1})$,

$\forall i > 0, \; 1 \leq k \leq i, \; 0 \leq j < 2^{k-1}$

$$\sigma_{f(i)+2^i-2^k+j} = \prod_{a=0}^{2^{i-k}-1} (2^i + j2^{i-k+1} + a, 2^i + j2^{i-k+1} + a + 2^{i-k}).$$

The first 12 values of σ are I; I; (0,1); I; (2,3); (0,2)(1,3); I; (4,5); (6,7); (4,6)(5,7); (0,4)(1,5)(2,6)(3,7); I.

The Even Distributed Gray Code is then defined as:

$$\begin{aligned} E_1 &= 0 \\ E_n &= E_{n-1} \circ n-1 \circ \sigma_{n-1}(E_{n-1}) \\ \text{where } \circ &= \text{string concatenation} \\ \text{and } \sigma & \text{ is applied to each word of its argument code} \end{aligned}$$

Lemma 2 *The even distributed gray code transition sequence is a valid gray code sequence.*

Proof: The proof is inductive. Clearly E_1 is a gray code transition sequence. In the recursive construction of E_n both E_{n-1} and $\sigma_{n-1}(E_{n-1})$ are transition sequences by the induction hypothesis and Lemma 1 so they each form gray codes within $(n-1)$-dimensional subcubes. The element $n-1$ connects the two $(n-1)$-dimensional gray codes into a single n-dimensional gray codes.

Implicitly the use of gray code transition sequences uses Lemma 1 to ensure that no matter what vertex is reached after traversing E_{n-1} followed by $n-1$ the sequence $\sigma_{n-1}(E_{n-1})$ will form a gray code within a $(n-1)$-dimensional subcube.

Lemma 3 *In the n bit even distributed gray code transition sequence no bit occurs more often then $3 \cdot 2^n/n$ times.*

Proof: A careful bookkeeping shows that when $n = 2^i + i \mid i \geq 0$ that all dimensions less than 2^i are used $2^{2^i} - 1$ times and all dimensions greater than or equal to 2^i are used a total of 2^i times. This yields a maximum dimension use that asymptotically approaches the average $2^n/n$ as i approaches infinity.

When n is not of this form the dimension use can become slightly more skewed but a simple inductive proof shows that the maximum use of a dimension is at most three times the average use.

4.1.4 Properties of Gray Codes

Fact 3 In the binary reflected gray code bit $b > 0$ is used 2^b times and for any two levels at which b is used there exists some bit $b' < b$ which is used an odd number of times between these two levels.

Fact 4 Every contiguous subsequence of any gray code contains at least one element an odd number of times.

4.2 Moments

The moment function assigns a $\lceil \log n \rceil$–bit label to each node of Q_n so that all the neighbors of each node have distinct labels. This simple property underlies all the multiple–path embeddings presented in this thesis. In the following, $\mathrm{b}(x)$, $0 \leq x < n$ denotes the $\lceil \log n \rceil$–bit binary representation of the number x. Also, $\bigoplus$ denotes the *bitwise* xor of $\lceil \log n \rceil$-bit numbers.

Definition 1
The moment *of an n-bit number* $v = v_{n-1}v_{n-2}\dots v_0$ *is defined by* $M(0) = \mathrm{b}(0)$ *and* $M(v) = \bigoplus_{i|v_i=1} \mathrm{b}(i)$.

Lemma 4 *Each hypercube neighbor of a given node, u, has a distinct moment.*

Proof: Let v and w be the neighbors of u in dimension i and j. Then $M(v) = M(u) \oplus \mathrm{b}(i) \neq M(u) \oplus \mathrm{b}(j) = M(w)$. ∎

The moment of a node can take on any non-negative integer value less than $2^{\lceil \log n \rceil}$. Consequently when the number of dimensions is a power of two then there are exactly n possible values of the moment and in all cases there are fewer than $2n$ values of the moment.

4.3 Simple Hypercube Embeddings

In this section we present three simple embeddings (these embeddings also appear in [32].) They further illustrate the idea of an embedding and extend our understanding of the flexibility of the hypercube. In the next chapter we will extend the CCC embedding to produce a multiple-copy embedding.

4.3.1 FFT

In order to show that the FFT is a subgraph of the hypercube we describe an embedding which maps the vertices of the FFT onto the nodes of the hypercube. The n-level FFT has $(n+1)2^n$ vertices so our hypercube will have $n + \lceil \log(n+1) \rceil$ dimensions. For convenience, let $\beta = \lceil \log(n+1) \rceil$.

The Embedding

We first label each edge of the FFT by a single dimension of the hypercube. Every level-ℓ cross-edge is labeled $G_\beta(\ell+1)$, the $\ell+1$st bit in a β-bit gray code. Every level-ℓ straight-edge is labeled $\ell+\beta$. Thus, the labels on cross-edges are disjoint from the labels on straight-edges.

The origin vertex $s = \langle 0,0 \rangle$ of the FFT is mapped to the hypercube node $\phi(s) = 0$. The remaining FFT nodes are assigned hypercube addresses in the following way. To compute the hypercube address $\phi(v)$ of vertex v in the FFT, pick any path from the origin s to v. The ith bit of $\phi(v)$ is 1 if and only if dimension i appears as the label of an odd number of edges along the path. For example, if we choose the path from s to $\langle 1,4 \rangle$ in the 4-stage FFT to be the path which traverses edges labeled with dimensions $0,1,3,1,4,0,3,2,0$ then the hypercube address of v would be $\phi(v) = 10101_2$; positions 0, 2, and 4 occur an odd number of times.

At this point we need to establish three properties of the embedding:

ϕ is well-defined, i.e., the address $\phi(v)$ is independent of the path chosen from the origin to v.

ϕ is injective, i.e., each node in the FFT is assigned a distinct hypercube address.

ϕ is dilation 1, i.e., if edge (u,v) is in the FFT then $(\phi(u),\phi(v))$ is an edge of the hypercube.

Lemma 5 (ϕ is well-defined)
For each vertex $v = \langle l,c \rangle$ in the FFT and any two paths P and Q from $s = \langle 0,0 \rangle$ to v in the FFT, the hypercube address assigned to v using P is the same as the address assigned using Q.

Proof: We start with three facts about cycles in FFTs.

For every cycle O in the FFT the number of level-l edges along O is even. Since O is a cycle every edge from one level to the next higher must be matched by an edge coming back down.

For every cycle O in the FFT the number of level-l cross-edges along O is even. Each level-l cross-edge leads to a vertex whose column address is the same as the previous vertex except that the lth bit is complemented. Thus an even number of complements at each level are necessary to return to the first vertex.

For every cycle O in the FFT the number of level-l straight-edges along O is even. The total number of level-l edges is even as is the number of level-l cross-edges so the number of level-l straight-edges must also be even.

Now consider the hypercube dimension labels associated with a cycle in the FFT. Since all level-l edges of each type are assigned the same label each hypercube label must appear an even number of times.

Next consider the cycle which starts at s, follows P to v, and then follows Q back to s. Since each hypercube dimension appears an even number of times

the parity of the number of appearances along P must be the same as the parity along Q for every dimension. Therefore the address assigned to v is the same whether P or Q is used. ∎

Lemma 6 (ϕ is injective)
For every pair of vertices $u = \langle l_1, c_1 \rangle, v = \langle l_2, c_2 \rangle$ in the FFT,
$u \neq v \Rightarrow \phi(u) \neq \phi(v)$.

Proof: We assume $\phi(u) = \phi(v)$ and derive a contradiction. W.l.o.g. assume $l_1 \geq l_2$. Let $u' = (n, c_1)$ and $v' = (0, c_2)$. Consider the path in the FFT which starts at u, traverses straight-edges up to u', follows the unique path to v', and then traverses straight-edges up to v. Let v'' and u'' be, respectively, the nodes at level l_1 and l_2 along the path between v' and u'. (See Figure 4.1) Since u and v are mapped to the same node in the hypercube the path must cross every hypercube dimension an even number of times.

Now consider all the level-i edges in the path. For $i \geq l_1$ or $i < l_2$ there are two level-i edges in the path, one of which is known to be a straight-edge. However, the hypercube dimension assigned to a level-i straight-edge is only assigned to level-i straight-edges. Thus the only other edge which could cross this hypercube dimension is the other level-i edge which must also be a straight-edge. Therefore the paths from u to u' and u' to u'' must be the same making $u = u''$. Similarly $v = v''$.

For any remaining levels there is only one path edge on the level. If this edge were a straight-edge then it would be the only edge crossing its hypercube dimension. Thus all the edges on these levels must be cross-edges. However, no subsequence of dimensions from a gray code crosses every dimension an even number of times (see Fact 4). Thus the path does not exist and we have a contradiction. ∎

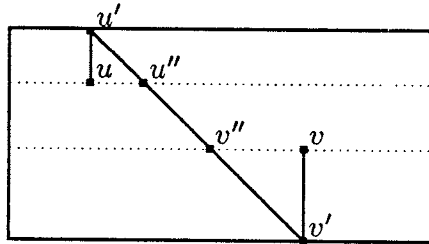


Figure 4.1: A Path from u to v

Lemma 7 (ϕ is dilation 1)
For every pair of vertices $u = \langle l_1, c_1 \rangle, v = \langle l_2, c_2 \rangle$ in the FFT,
(u, v) is an edge of the FFT $\Rightarrow (\phi(u), \phi(v))$ is an edge of the hypercube.

Proof: Every FFT edge was assigned a single hypercube dimension. ∎

Lemmas 5, 6, and 7 establish:

Theorem 1 *Each FFT is a subgraph of the smallest hypercube containing at least as many nodes as the FFT.*

4.3.2 Butterfly Graph

The butterfly graph is quite similar to the FFT except that the first and last levels of the FFT are identified as one in the Butterfly. This means that there are fewer nodes in the Butterfly than in the FFT for the same number of inputs. Thus our embedding of the Butterfly will sometimes have a smaller hypercube with which to work.

Theorem 2 *Every Butterfly graph is embeddable in a hypercube with optimal expansion and optimal dilation. Thus, for each m, $B(m)$ is embeddable in $Q_{\overline{d}}$ with dilation 1 if m is even and dilation 2 if m is odd, where $\overline{d} =_{\text{def}} m + \lceil \log m \rceil$.*

The Embedding

Our embedding will make use of the following characterization of the Butterfly. $B(m)$ consists of two copies of $F(m-1)$ along with edges between each output and its corresponding input in *both* copies of $F(m-1)$. This characterization can be compared to that of $F(m)$ as consisting of two copies of $F(m-1)$ and two sets of 2^{n-1} new nodes along with edges between each output in each copy of $F(m-1)$ and its corresponding node in *both* sets of new nodes. The new nodes become the new outputs for the FFT.

We first reserve the highest dimension, $\overline{d}$, as special. This partitions our hypercube into two subcubes of $\overline{d} - 1$ dimensions each. Embed the first copy of $F(m-1)$ into the subcube in which bit $\overline{d}$ is zero using the simple embedding of Section 4.3.1. Let α be the address of the hypercube node to which the first output of this copy of $F(m-1)$ is mapped. Next embed the second copy of $F(m-1)$ into the second subcube (on which bit $\overline{d}$ is one) using the same simple embedding except that the origin node $\langle 0, 0 \rangle$ is mapped to the neighbor across dimension $\overline{d}$ of α instead of to hypercube node 0.

The proof that this embedding is valid divides into two parts: those properties inherited from the embeddings of $F(m-1)$ and the verification of the dilation of

the additional edges between outputs of the $F(m-1)$ and their corresponding inputs in both copies. The well-definedness of the embedding is inherited from the embeddings of the copies of $F(m-1)$ and the fact that they are embedded in disjoint subcubes. The dilation of all the edges other than the ones between outputs and inputs is also inherited from the $F(m-1)$ embeddings. Thus only the dilation of these additional edges must be verified.

The edges between an output and its corresponding input in the same copy of $F(m-1)$ follows directly from the fact that the dimensions used for the straight-edges form a length m or $m+1$ cycle. When m is even the cycle is length m, we have used $m-1$ edges along the cycle and thus there exists a single edge to complete the cycle connecting the output back to the input. When m is odd the cycle is length $m+1$ and thus we require two edges to complete the cycle and the dilation of the output to input edge is two.

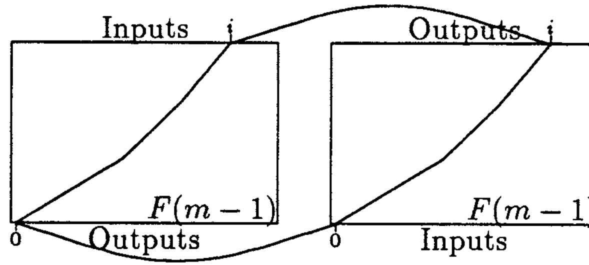


Figure 4.2: A cycle in the Butterfly

The edges between an output and its corresponding input in the other copy of $F(m-1)$ are all dilation 1 and in fact all cross dimension $\overline{d}$. We start with output 0 of the first copy of $F(m-1)$. Its image is, by the definition of the embedding, adjacent to the image of input 0 of the second copy across dimension $\overline{d}$. Now consider the output i of the second copy along with the following cycle in the hypercube: start at the image of output 0, follow the image of the unique length $m-1$ path in $F(m-1)$ to input i, next cross to the image of output i of the second copy, continue along the image of the unique path in $F(m-1)$ to input 0 of this copy, and lastly complete the cycle by crossing to the image of output 0. (See Figure 4.2) We already know that the last step, going from input 0 of the second copy to output 0 of the first copy, can use exactly dimension $\overline{d}$. The key point is that the images of the two unique length $m-1$ paths must cross exactly the same hypercube dimension since they must either both follow a cross-edge or both follow a straight-edge on corresponding levels of their respective FFTs and the same hypercube dimensions are being used for all cross-edges (respectively straight-edges) in *both* FFT embeddings. Thus, since this is a cycle in the hypercube the remaining step, going from the image of input i in the first copy to the image of output i in the second, must also cross only dimension

$\overline{d}$.

Now that we know that output 0 of the second copy is adjacent across dimension $\overline{d}$ to input 0 of the first copy we can use a symmetric argument to argue that the edges from output i of the first copy to input i of the second are also dilation 1.

Note that dilation 2 is optimal for m odd since in this case $B(m)$ has odd length cycles and no hypercube has odd length cycles.

4.4 The Cube-Connected Cycles

The CCC can be embedded in a manner similar to the embeddings of the FFT and Butterfly. There are, however, simpler embeddings of the CCC graph. Since the CCC is a subgraph of the cross-product of a hypercube and a ring the CCC can be embedded in the hypercube by decomposing the $(m + \lceil \log m \rceil)$-dimension hypercube into the cross-product of a m-dimension hypercube with a $\lceil \log m \rceil$-dimension hypercube and embedding the hypercube factor of the CCC in the m-dimension hypercube using the identity map and the ring factor in the other hypercube using a gray code.

Theorem 3 *Every Cube-connected-cycles graph is embeddable in a hypercube with optimal expansion and optimal dilation. Thus, for each m, $CCC(m)$ is embeddable in $Q_{\overline{d}}$ with dilation 1 if m is even and dilation 2 if m is odd, where $\overline{d} =_{\text{def}} m + \lceil \log m \rceil$.*

We defer the proof of this theorem until the next chapter where we will prove the stronger result that m copies of the m-level CCC can be embedded in the hypercube with optimal expansion and dilation and overall congestion of only 2. Once again note that dilation 2 is optimal for m odd since in this case $CCC(m)$ has odd length cycles and no Hypercube has odd length cycles.

4.5 Multiple-Copy Embeddings of Cycles

Hypercubes can be decomposed into edge-disjoint Hamiltonian cycles (see [5] for a survey). In particular, Alspach, Bermond, and Sotteau [5] show that the edges of every (undirected) hypercube with $2n$ dimensions can be partitioned into n (undirected) Hamiltonian cycles. Furthermore, if the number of dimensions is $2n+1$, then the edges can be partitioned into n cycles and one perfect matching.

These results are easily extended to multiple–copy embeddings of directed cycles in directed hypercubes. For the n-dimensional cube, we orient each of the $\lfloor n/2 \rfloor$ undirected cycles in either direction to obtain the following lemma. When n is odd it is not, in general, possible to partition Q_n into n directed cycles.

Lemma 8 *For all even n, n copies of the 2^n–node directed cycle can be embedded into Q_n with dilation 1 and congestion 1.*
For all odd n, $n-1$ copies of the 2^n–node directed cycle can be embedded into Q_n with dilation 1 and congestion 1.

4.6 Disjoint Paths

In later chapters, embeddings are presented which supply multiple paths between guest images. The search for such multiple-path embeddings was in part motivated by the following simple facts from [31].

Fact 5 Between any two nodes in the n-dimensional hypercube there are n edge-disjoint paths.

Fact 6 Between any two sets of $p \leq n$ vertices in the n-dimensional hypercube there are p vertex-disjoint paths such that each vertex in each set is the endpoint of exactly one path.

4.7 Eigenvalues of the Hypercube

Any finite graph can be represented by an adjacency matrix. For d-regular graphs the adjacency matrix can be normalized to become the transition matrix of the Markov chain for a random walk on the graph. The magnitude of the second largest eigenvalue of the matrix can be used to bound the convergence time of the walk. Several researchers have used this property to produce routing algorithms on expander graphs[61, 79]. In this section we enumerate the eigenvalues of the hypercube. (See [50] for related results.)

Lemma 9 *For the d-dimension hypercube the eigenvalues are $\{d-2i \mid 0 \leq i \leq d\}$. The eigenvalue $d-2i$ has multiplicity $\binom{d}{i}$.*

Proof: Let M_k be the adjacency matrix for the k-dimension hypercube.

$$M_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \; M_{k+1} = \begin{pmatrix} M_k & I \\ I & M_k \end{pmatrix}$$

Clearly the eigenvalues of M_1 are 1 and -1 with eigenvectors $\binom{1}{1}$ and $\binom{1}{-1}$. The lemma then follows by the observation that for any eigenvalue c with eigenvector v of M_k there is an eigenvalue $c+1$ with eigenvector $\binom{v}{v}$ and an eigenvalue $c-1$ with eigenvector $\binom{v}{-v}$ of M_{k+1}. ∎

Chapter 5

Efficient Embeddings

We have already noted that, unlike trees and meshes, the degree of the hypercube network grows as the network becomes larger. The standard graph embedding model ignores this fact and thus has led to the hypercube being considered a good general-purpose network. There exist good (packet-cost one) embeddings of meshes, trees, and butterflies into hypercubes. It seems that rather than building meshes for PDE solvers, trees for divide-and-conquer, and butterflies for convolutions that a hypercube can be used to simulate all the others.

The consideration of physical construction constraints seems to contradict the hypercube's generality. In this chapter we look in particular at pin limitations. We will first show that pin limitations make most previous embeddings into the hypercube inadequate to allow efficient simulation by the hypercube. These earlier embeddings make poor use of the hypercube edges and thus poor use of the pins. Having established that standard embeddings are inefficient we present two types of extended embeddings which allow efficient use of the hypercube's edges. We demonstrate how these multiple-path and multiple-copy embeddings can be used to efficiently simulate meshes, trees, and CCCs on the hypercube; even when pin limitations are considered.

There are, as we described in Chapter 2, many other resource constraints besides pin limitations. The incorporation of physical limitations into graph embeddings has just begun. The hypercube is also not the only proposed general computer. In Chapter 6 we extend some of the ideas presented in this chapter to other networks. In the Chapter 7 we add some thoughts about wire length.

5.1 Pin limitations

Suppose each processor is on its own chip and that each chip has $2W$ pins used for data transmission in the interconnection network. A single bit of data can be transmitted across each bit in a time step. If the network has constant degree

(such as a mesh, tree, or butterfly) then each link can be assigned $O(W)$ pins. On the other hand the n-dimension hypercube can only assign $O(W/n)$ pins to each link. The constant-degree network can send a W-bit word across a link in a constant number of steps while the hypercube needs $O(n)$ steps.

As an illustration of the effect of pin limitations on network comparisons we compare a cycle to a hypercube. If we build the cycle we can use W pins to connect to each neighbor; W-bit words can be sent one place around the cycle in a single clock step. Can the hypercube do as well?

The standard graph embedding approach would be to use a binary reflected gray code embedding of the cycle in the hypercube. The cost of the gray code simulation is perfect; each cycle message need travel over only a single link. The problem is that the links are thinner; it takes n clock steps to transmit the W bit word. Apparently the hypercube network is a poor substitute for the cycle network.

A moments reflection reveals that the binary reflected gray code is using only one nth of the hypercube edges. Pins have been used to construct the idle edges but are not being used by the embedding. Any embedding which allows the hypercube to efficiently simulate the cycle will have to somehow use all the hypercube edges.

It is well-known that between any two nodes of the hypercube there are n edge-disjoint paths (See Fact 5). Perhaps the binary reflected gray code can be made efficient by using multiple paths between each two successive nodes on the cycle.

Unfortunately, the binary reflected gray code cannot employ the idle edges to speed transmission. To prove this assertion we refer to Figure 5.1 which shows the binary reflected gray code embedding of the 16-node cycle in the 4-dimension hypercube. The label on each cycle edge corresponds to the dimension of the hypercube edge which is the image of the cycle edge. There are 2^{n-1} cycle edges which use a hypercube edge in dimension 0. Any path taken by any message corresponding to one of these edges must cross at least one dimension-0 hypercube edge.

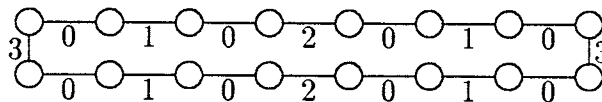


Figure 5.1: The binary reflected graycode embedding

If W bits are to be sent along each cycle edge then $W2^{n-1}$ bits must cross edges in dimension 0. The entire capacity of the 2^n edges in dimension 0 is only $W2^n/n$. Thus, regardless of multiple paths or clever routing schemes, $n/2$

clock steps will be required if the cycle is embedded into the hypercube using the binary reflected gray code.

Similar bottlenecks occur if attempts are made to create pin-efficient embeddings from other standard embeddings of constant degree graphs into hypercubes. Entirely new embeddings are necessary.

5.2 Multiple-path Embeddings of Grids

In this section we show that grids have efficient multiple-path embeddings in hypercubes. We present the technique for cycles; the extension to multi-dimensional grids is noted at the end of the section.

5.2.1 Multiple-path embeddings of cycles

We present two multiple-path embeddings of cycles. The first embedding maps the 2^n-node cycle into Q_n with load 1, width $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$-packet cost 4. Roughly speaking, half of all hypercube edges transmit a packet at each of the 4 steps. The second embedding attempts to keep all hypercube edges busy at each step; it maps the 2^{n+1}-node cycle into Q_n with load 2, width $\lfloor n/2 \rfloor$, and $\lceil n/2 \rceil$-packet cost 4. When n is a power of two then the $n/2$-packet cost of both embeddings can be reduced to 3.

A key idea in both multiple-path embeddings of cycles will be the conversion of the multiple-copy embedding of Lemma 8 into a multiple-path embedding. In Section 5.4 this idea will be generalized to allow the construction of additional multiple-path embeddings.

5.2.2 Embedding Cycles with Load 1

Theorem 4 *The length-2^n directed cycle can be embedded in Q_n with width $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$-packet cost 4. When n is a power of 2 the $n/2$-packet cost is 3.*

Proof: Suppose that $n = 4k + r$, $0 \le r < 4$, $k > 0$.* Thus for $r = 0, 1$ we have $\lfloor n/2 \rfloor = 2k$ and for $r = 2, 3$ we have $\lfloor n/2 \rfloor = 2k - 1$.

First we partition Q_n as the product $Q_{2k} \times Q_{2k+r}$. The product can be visualized as a grid with 2^{2k} rows and 2^{2k+r} columns. Each row is connected as Q_{2k+r} and each column is connected as Q_{2k}. The most significant $2k$ bits of the addresses in Q_n name a grid row while the least significant $2k + r$ bits name a grid column.

Furthermore we partition the columns into 2^r blocks by letting the least significant r bits of the column name be the name of a block and the most significant

*$k = 0$ is trivial

$2k$ bits be the name of a position within a block. Note that within each block each row and column forms a copy of Q_{2k}. Thus if each column is treated as a *coarse* node it has $2k$ neighboring columns within the block. (See Figure 5.2)

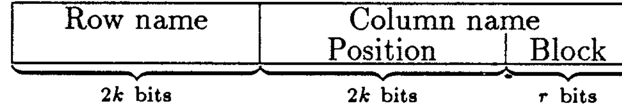


Figure 5.2: Dividing addresses into three fields

Next we number the k edge-disjoint undirected cycles and the $2k$ edge-disjoint directed cycles of Q_{2k} provided by Lemma 8 in Section 4.5 as follows. The undirected cycles are numbered arbitrarily. Then for $0 \leq i < k$ the ith undirected cycle oriented in one direction is numbered $2i$ and the ith undirected cycle oriented in the other direction is number $2i + 1$. For our embedding we will need $2^{\lceil \log 2k \rceil}$ cycles so when $2k$ is not a power of two we use some cycles twice. For $0 < j \leq 2^{\lceil \log 2k \rceil}$ our jth cycle is the directed edge-disjoint cycle numbered $j \bmod 2k$. Note that no cycle is used more than twice.

Now we can choose a *special* cycle within the subcube associated with each column. For column c at position x in block b we select the edge-disjoint directed cycle number $M(x)^\dagger$ as the special cycle. Observe that each node of Q_n lies in exactly one special cycle, and we have selected 2^{2k+r} special cycles.

We now use these special cycles, plus a few edges in the rows to form our length-2^n cycle, $\mathcal{C}$. (See Figure 5.3.) The cycle $\mathcal{C}$ consists of $2^{2k} - 1$ consecutive edges from each special cycle and 2^{2k+r} row edges; each row edge connects one column's special cycle to the special cycle in the next column. The order in which $\mathcal{C}$ visits columns is specified by the binary reflected gray code G_{2k+r} defined in Section 4.1.2.

Formally, the first vertex of $\mathcal{C}$ is the hypercube node at row 0 of column 0. The first $2^{2k} - 1$ edges of $\mathcal{C}$ follow the special cycle in column 0 (until the special cycle is about to return to row 0). The next edge of $\mathcal{C}$ is the row edge in the first dimension of G_{2k+r}. In the new column reached, and each successive column thereafter, $\mathcal{C}$ follows $2^{2k} - 1$ edges of the special cycle and then leaves via the row edge in the next dimension listed in G_{2k+r}. Upon returning to column 0 the cycle $\mathcal{C}$ is complete.

By construction $\mathcal{C}$ visits each of the 2^{2k+r} columns exactly once, in the order determined by G_{2k+r}. Since a visit by $\mathcal{C}$ to a column involves the traversal of a special cycle, each node is visited exactly once. It remains to show that when $\mathcal{C}$ returns to column 0 it is in row 0.

†Recall that $M(x)$ denotes the moment of x. (See Section 4.2)

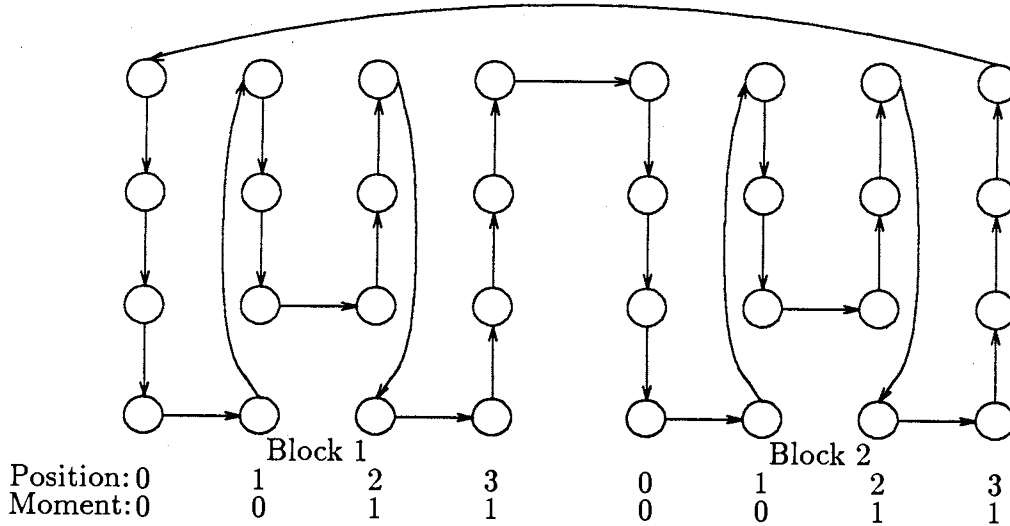


Figure 5.3: Forming the length-2^n cycle, $\mathcal{C}$, from column special cycles

Group the columns, starting with column 0 and in the order they were visited, into sets of four. Between columns within each set the dimensions specified by G_{2k+r} for row edges are always 0, 1, and 0. Thus within each set the moments of the first two columns are the same ($x \oplus 0 = x$) as are the moments of the last two. Furthermore the cycle associated with the moment of the first two columns is the reverse orientation of the cycle associated with the moment of the last two. (The names of the cycles were chosen so that names differing in the least significant bit corresponded to opposite orientations of the same undirected cycle.) The path taken by $\mathcal{C}$ in the first two columns is reversed in the next two columns thereby returning $\mathcal{C}$ to row 0. Since the number of columns is divisible by four, $\mathcal{C}$ must end in row 0 after visiting all the columns.

We are now ready to make the edges of $\mathcal{C}$ wide. We supplement each special edge with the $2k$ length-three paths which cross into neighboring columns, follow the projection of the edge in the neighboring column, and then cross back into the original column (See Figure 5.4).

Formally, we choose the first $2k$ edge-disjoint paths for column-edge (u, v) along a special cycle (u and v differ in dimension i, $2k+r \leq i < 4k+r$) as follows. The jth path, $0 \leq j < 2k$, from u to v is of the form $u, u \oplus 2^{r+j}, u \oplus 2^{r+j} \oplus 2^i, v$. In other words the jth path crosses into the column adjacent via the edge in dimension $r+j$ (one of the dimensions used within block row subcubes), crosses the ith dimension while remaining in this new column, and then crosses back to its original column via an edge in dimension $r+j$.

We add a $2k+1$st path of length one which goes directly from u to v via the

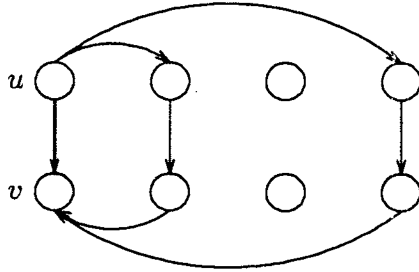


Figure 5.4: The length-three paths

edge in dimension i.

For the edges in the rows we similarly pick $2k+1$ edge-disjoint paths. The first $2k$ paths for row edge (u, v) (where the dimension i between u and v is now such that $0 \leq i < 2k+r$) are chosen so that path j, $0 \leq j < 2k$, is of the form $u, u \oplus 2^{2k+r+j}, u \oplus 2^{2k+r+j} \oplus 2^i, v$. Again we add a $2k+1$st path which goes directly across dimension i.

We claim that this multiple-path embedding of the special cycles has $\lceil n/2 \rceil$-packet cost 4. First observe that each first edge of the paths corresponding to a single edge of $\mathcal{C}$ is in a different dimension. Thus all first edges emanating from each node are disjoint. Similarly, the set of final edges is also disjoint. Next, we argue that the middle edges have congestion at most two.

Our main tool will be the following observation about the special cycles. Consider column c in block b, and its $2k$ neighboring columns within block b. By Lemma 4 each neighbor has a distinct moment. When n is a power of two each moment is assigned a distinct special cycle and in all cases at most two moments are assigned the same special cycle. Therefore when the special cycles of all the neighbors are projected onto column c their images are edge disjoint in the first case and have congestion at most two in all cases.

All the middle edges of paths for column edges are projections of special-cycle edges onto neighboring columns within a block. But, the projections of all special cycles onto any column have congestion at most two. Therefore each middle edge contends with at most one other.

The middle edges of paths for row edges are projections onto neighboring rows and thus disjoint from the column path's middle edges. Furthermore each row edge in $\mathcal{C}$, and all its projections, connects a unique pair of columns and thus its projections are disjoint from those of any other row edge in $\mathcal{C}$.

Thus a packet may be sent along all paths including the direct path on step

one, forwarded along all middle edges of the length-three paths on steps two and three, and arrive at their destination on step four. Furthermore an additional packet can be sent along the direct path on step four. Thus the embedding yields $(2k+2)$-packet cost 4 which is always better than that required by the theorem. (When n is a power of two the middle edges can be traversed in a single step and the cost is only 3.) ∎

Since this first embedding uses only about half the hypercube edges a natural question is whether the idle edges can also be put to use. We do not know how to increase the width in order to increase the efficiency. However, by increasing the load to two and thereby doubling the number of guest (cycle) edges the embedding in the next section uses nearly all the hypercube edges.

5.2.3 Load 2 Embeddings which Fully Utilize the Hypercube Links

Theorem 5 *The length-2^{n+1} directed cycle can be embedded in Q_n with width $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor$-packet cost 4. When n is a power of two the cost is 3.*

Proof: The most difficult case is $n \equiv 3 \pmod 4$. Thus we state the proof for the case $n = 4k+3$ and discuss the other values for n mod 4 at the end.

First we partition Q_{4k+3} into $Q_{2k+2} \times Q_{2k+1}$.

The product can be visualized as a 2^{2k+2} by 2^{2k+1} grid of nodes with each row connected as Q_{2k+1} and each column connected as Q_{2k+2}. The most significant $2k+2$ bits of the addresses in Q_n name a grid row, the next $2k+1$ bits name a grid column. The rows and columns each form a subcube.

As in Theorem 4 we will need a cycle for each possible moment of a node in either the row or column subcube. The number of special cycles needed will thus be $2^{\lceil \log(2k+1) \rceil}$ and $2^{\lceil \log(2k+2) \rceil}$ respectively. There are $2k$ disjoint cycles available in the row subcubes and $2k+2$ for the column subcubes. By using each cycle at most twice each moment can be assigned its own special cycle. For column c we choose the disjoint column cycle assigned to $M(c)$ while for row r we choose disjoint row cycle $M(r)$. Thus each node of Q_n is on two special cycles and the subgraph of Q_n induced by the special cycles spans Q_n and has in-degree and out-degree equal to 2 at every node.

By choosing the Eulerian tour of the induced spanning graph as our length-2^{n+1} cycle we need only show that each edge of the special cycles can be given width $2k+1$ with $\lfloor n/2 \rfloor$-packet cost 4.

Each special cycle is given width $2k+1$ by choosing $2k+1$ edge-disjoint paths in the following manner. If edge (u,v) is along a column special cycle (and thus u differs from v in dimension i, $2k+2 \leq i < 4k+3$) then the jth path, $0 \leq j < 2k+1$, from u to v is of the form $u, u \oplus 2^j, u \oplus 2^j \oplus 2^i, v$. On the

other hand if edge (u, v) is along a row cycle (and thus traverses dimension i, $0 \leq i < 2k + 2$) then the jth path, $0 \leq j < 2k + 1$, from u to v is of the form $u, u \oplus 2^{2+2k+j}, u \oplus 2^{2+2k+j} \oplus 2^i, v$. (Note that we could not add the direct path as a $2k + 2$st path because for columns edges the direct edge is used by row paths and vice versa.)

We claim that this width-$(2k+1)$ embedding of the special cycles has $(2k+1)$-packet cost 4. Observe that the set of first edges on all the length 3 paths for column edges are in dimensions less than $2k + 2$ while the set of first edges on paths for row edges are in dimensions greater than or equal to $2k + 2$. Thus the first edges emanating from each node, and therefore all the first edges, are disjoint. Similarly, the final edges are also disjoint. Next, we argue that the middle edges are disjoint.

All the middle edges of paths for column edges are projections of special-cycle edges onto neighboring columns. But, as we observed earlier, the projections of all special cycles onto any column or row have congestion at most two. Since the row edges are disjoint from the column edges all the middle edges together have congestion at most two.

Since the sets of first and last edges are each disjoint and the middle edges share their hypercube edge with at most one other middle edge, packets can be sent down each path in four steps and the $(2k + 1)$-packet cost is indeed 4.

For other values of n mod 4 the hypercube is similarly decomposed into as near equal pieces as possible. Note that when n is a power of two the cross-product decomposition yields two hypercubes whose dimension is a power of two. Thus no edge disjoint cycle need be used twice in order to assign cycles to moments and the cost becomes 3. ∎

5.2.4 Bounds on Width and Cost

We next show that the multiple-path embedding of Theorem 5 is the best possible when n is a power of two.

Lemma 10 *For $w > 2$, every width-w embedding requires dilation (and therefore p-packet cost, $p \geq w$) at least 3. There is no p-packet cost 3 embedding of the length-2^{n+1} cycle in Q_n with $p > \lfloor n/2 \rfloor$.*

Proof: In order to have $w > 2$ edge-disjoint paths between two distinct hypercube nodes, one of the paths must have length 3 or greater. Therefore, for all width-w embeddings, $w > 2$, the cost must be 3 or greater.

The number of edges traversed by all the paths in the embedding equals the sum, over all guest edges, of their dilations. To achieve cost 3, at least $w - 1$ edges must have dilation 3. Thus the sum of the dilations is greater than

$2^{n+1} \times (w-1) \times 3$. On the other hand the number of hypercube edges available during three steps is simply three times the number of directed hypercube edges or $3n2^n$. Thus in order for the number of edges used to be no greater than the number of edges available, we have that $6 \cdot 2^n(w-1) < 3n2^n$ which implies that $w \leq \lfloor n/2 \rfloor$. ∎

5.2.5 Multiple-path embeddings of grids

In Chapter 3 we mentioned that grids/tori are cross-products of paths/cycles and that hypercubes are cross-products of smaller hypercubes (see Fact 2). These two facts lead to a natural technique for extending embeddings of paths into embeddings of grids. Each axis of the grid is embedded in a hypercube via the cycle embedding and then the cross-product of the hypercubes inherits the embeddings of the axes and thus an embedding of the grid.

For example, when the k-dimensional grid with each side of length 2^a is embedded in Q_{ak} using the cross-product decomposition and the embedding of Theorem 4, we obtain a width-$\lfloor a/2 \rfloor$ embedding with $\lceil a/2 \rceil$-packet cost 4.

When the sides of the grid are equal, but not a power of 2, the cross-product embedding may use the hypercube nodes inefficiently. For example the 5 by 5 grid is the cross-product of two 5-node paths. Each 5-node path can be embedded in the 8-node hypercube and the cross-product of two 8-node hypercubes is the 64-node hypercube. Thus by embedding each axis into its own independent factor subcube we obtain an embedding of the 5 by 5 grid into the 64-node hypercube. The 25 grid nodes could, however, have fit in the 32-node hypercube. The ratio of the size of the hypercube used to the size of the smallest hypercube at least as large as the guest graph is often called the *expansion*. In the 5×5 grid example the expansion is 2; in general, cross-product decomposition can lead to expansion $k+1$ for k-axis grids.

Corollary 1 *The k-axis grid with all side lengths equal to L can be embedded in $Q_{k\lceil \log L \rceil}$ with expansion $k+1$, width $\lfloor \lceil \log L \rceil /2 \rfloor$ and $\lceil \lceil \log L \rceil /2 \rceil$-packet cost 4.*

Proof: Embed each axis in $Q_{\lceil \log L \rceil}$ via the embedding of Theorem 4 and use the cross-product decomposition. ∎

For width-1 embeddings it has been shown that using gray codes for the paths and applying the cross-product technique to k-axis grids causes expansion no greater than $k+1$. Chan [25] has shown that by abandoning the cross-product approach and increasing the dilation to $O(k)$ the expansion can be reduced to one.

Unfortunately Chan's techniques do not apply immediately to our multiple-path embeddings. Thus we have no current alternative to the cross-product technique.

When the sides of the grid are not equal (more precisely when the ceiling of the logarithm of their lengths are not equal) the multiple-path embeddings require additional work. We cannot just embed each axis in a hypercube large enough to hold it because the width of the embedding depends on the number of dimensions in the host hypercube. If one axis of a grid were embedded to a smaller hypercube than another axis then the width of edges on the first axis would be smaller than the width for edges on the other.

In order to compensate for the need to have all sides of the k-axis grid be equal we first *square* the grid; that is we map the k-axis grid with unequal sides onto a k-axis grid with equal sides. Aleliunas and Rosenberg [4] show that two-axis grids can be squared with constant dilation and expansion and Kosaraju and Atallah [49] extend this result to k-axis grids. Ellis[29] reduces the constants slightly.

Combining the grid squaring with Corollary 1 gives us the following:

Corollary 2 *The $L_1 \times L_2 \cdots \times L_k$ grid can be embedded in Q_{kL} with width $\lfloor \lceil \log L \rceil /2 \rfloor$, $O(1)$ expansion, and $\lceil \lceil \log L \rceil /2 \rceil$-packet cost $O(1)$, where $L = \lceil (\prod_{i=1}^{k} L_i)^{1/k} \rceil$.*

We leave to the reader the proof of Corollary 2 and to show that the embeddings of Theorem 5 can be used to create load-2^k embeddings of k-axis grids which more fully use the edges of the hypercube.

5.3 Multiple-copy Cube-Connected-Cycles

We now turn from grids to the cube-connected-cycle network (CCC). In this section we will assume that we are given a CCC and required to embed it in the smallest hypercube having at least as many nodes as the CCC. Since the m-level CCC has $m2^m$ nodes, optimal expansion is achieved by embedding into the $(m + \lceil \log m \rceil)$-dimension hypercube.

Our main result in this section, a multiple-copy embedding, is stated below. Note that since the directed cube-connected-cycle (straight-edges directed toward higher level) has degree 2 the edge-congestion is optimal. The dilation is also optimal since when n is odd the cube-connected-cycle has odd cycles and thus dilation 2 is necessary.

Theorem 6 *n copies of the $n2^n$-node directed cube-connected-cycles network can be embedded in $Q_{n+\lceil \log n \rceil}$ with edge-congestion two and with dilation one (two) when n is even (odd).*

Before we proceed with the proof we introduce a few definitions.

5.3.1 Terminology

Windows A *window* $W \subseteq \mathsf{Z}_k$ is an *ordered* subset of the dimensions of the hypercube Q_k. For any node v in Q_k, the *signature* $\sigma_W(v)$ is the concatenation of the address bits of v in the dimensions ordered by W. For example, the signature of node 01001 over the window $W = \{1,4,3\}$ is 110, the bits in positions 1, 4, and 3. (The bits are numbered left to right starting with 0.)

Longest Common Prefix For any sequence a we denote the prefix of length i by $\rho_i(a)$. When a is a sequence of bits then $\rho_i(a)$ denotes the integer value of the first i bits. For any two sequences a and b, we define $\lambda(a,b)$ to equal the length of the longest common prefix.

We recall that H_k is the Hamiltonian cycle formed by starting at hypercube node 0^k and crossing successively the dimensions listed in the binary reflected gray code sequence G_k. Also that $H_k(i)$ and $G_k(i)$ denote the ith elements of their respective sequences.

5.3.2 Embeddings of the CCC Network

In Chapter 4 we claimed that the n-level CCC could be embedded in the hypercube with congestion 1 and dilation 1 if n is even and dilation 2 otherwise. In this subsection we describe an abstract embedding which will be more easily convertible into a multiple-copy embedding.

To embed the n-stage CCC into the hypercube Q_{n+r} (where $r = \lceil \log n \rceil$),[‡] first partition the $n+r$ hypercube dimensions into two windows W and $\overline{W}$ where $|W| = r$, $|\overline{W}| = n$, and $W \cap \overline{W} = \emptyset$. Implicitly the partition into windows defines a cross-product decomposition of the hypercube.

The CCC vertex $\langle \ell, c \rangle$ is mapped to the hypercube node with signature $H_r(\ell)$ on window W and signature c on window $\overline{W}$. Each column is thereby mapped to its own subcube via a gray code and each row to its own subcube via the identity map. A nice feature of this mapping is that level-ℓ straight-edges are mapped to hypercube edges in dimension $G_r(\ell)$. Similarly, level-ℓ cross-edges are mapped to hypercube edges in dimension $\overline{W}(\ell)$. Optimal dilation and congestion are thus readily apparent.

Observe that the embedding is completely specified by the choice of a length-r window W, a disjoint length-n window $\overline{W}$, and a Hamiltonian cycle H_r. Recall that windows are ordered sequences, so the choice of W does not completely specify $\overline{W}$.

[‡]For sake of convenience, we assume in the remainder of this section that n is a power of 2. For other values of n, the congestion for multiple-copy embeddings is, at worst, doubled and some edges suffer dilation 2.

5.3.3 The Multiple-copy embedding

In order to embed n copies of the CCC network into Q_{n+r}, we specify, for each copy, two disjoint windows and a Hamiltonian cycle. In this subsection we show how to make these choices such that the overall edge-congestion is at most 2. Since each copy is embedded as described in the previous subsection, the dilation is 1 for each embedding.

To show that naive choices are insufficient, we consider two extremes. First, suppose that we choose the same partition of hypercube dimensions for all n copies; in all n embeddings the straight-edges are mapped to the same set of r dimensions. There are $n2^n$ straight-edges in each copy, $n^2 2^n$ in total, whereas there are only $rn2^n$ hypercube edges in the r dimensions. Consequently, the edge-congestion is at least n/r.

Next, suppose instead that each copy, i, $0 \le i < n/r$, uses a distinct set of dimensions for its length-r window W_i. The proof that the congestion is again n/r typifies the arguments throughout the rest of this section. We pick a dimension and, among all copies, look at all the CCC vertices which use this dimension for their cross-edge. If many of these CCC vertices are mapped onto the same hypercube node then the congestion due to cross-edges is high.

Consider one of the r dimensions, call it d, not contained in any W_i. In each copy, edges in dimension d are images of cross-edges. Furthermore, for a given copy i, all CCC vertices whose cross-edge is mapped to dimension d are at the same CCC level. It is easily established that all the hypercube images of CCC vertices on one level under embedding i have the same signature, call it s_i, on W_i.

Since the windows are disjoint there exists a hypercube node, v, such that $\forall i$, $\sigma_{W_i}(v) = s_i$. Each copy maps a CCC vertex to v and the cross-edges emanating from each of these n/r CCC vertices are all mapped to dimension d. The congestion on dimension-d edges is therefore n/r.

In the remainder of this section we present an n-copy embedding which avoids such congestion. In particular, we establish that, for our embedding, every hypercube edge is the image of at most one CCC cross-edge.

Overlapping windows From the preceding discussion we know that the windows must be chosen carefully to avoid high congestion. We construct the length-r windows for the n copies as follows: all windows contain dimension 1; half of the windows contain dimension 2 and the other half contain dimension 3; in general, of all the windows that contain dimension i, half of them also contain dimension $2i$ and the other half contain dimension $2i+1$.

Multiple embedding The length-r window and the length-n window for the kth copy are denoted $\mathcal{W}^k$ and $\overline{\mathcal{W}}^k$; the ith element of these are denoted $\mathcal{W}^k(i)$

and $\overline{\mathcal{W}}^k(i)$. Similarly, let $\mathcal{H}^k$ denote the Hamiltonian cycle for the kth copy, and let $\mathcal{H}^k(i)$ be the ith node on the cycle.

We define $\mathcal{W}^k$, $\overline{\mathcal{W}}^k$, and $\mathcal{H}^k$ $(0 \leq k, \ell < n, 0 < i < \log n)$ formally as follows. (Again let $\oplus\, b(k)$ denote bitwise xor with the $(\log n)$-bit binary representation of k.)

$$\begin{aligned}
\mathcal{W}^k(0) &= 1 \\
\mathcal{W}^k(i) &= 2^i + \rho_i(k) \\
\overline{\mathcal{W}}^k(\ell) &= \begin{cases} \ell & \text{if } \ell \notin \mathcal{W}^k \\ n + \lfloor \log \ell \rfloor & \text{if } \ell \in \mathcal{W}^k \end{cases} \\
\mathcal{H}^k(\ell) &= H_r(\ell) \oplus b(k)
\end{aligned}$$

To show that the above defines n embeddings, we observe that $\forall k$, $\mathcal{W}^k \cap \overline{\mathcal{W}}^k = \emptyset$ and $\mathcal{H}^k$ is a Hamiltonian cycle. We leave the reader to verify these and the following observations.

Observations

The following properties of the embeddings will be useful later.

1. $\forall k, \ell$ the kth embedding maps every CCC vertex at level-ℓ to a hypercube node whose signature on $\mathcal{W}^k$ equals $\mathcal{H}^k(\ell)$.

2. In the kth embedding, level-ℓ straight-edges are mapped to dimension-$\mathcal{W}^k(G_r(\ell))$ hypercube edges.

3. In the kth embedding, level-ℓ cross-edges are mapped to dimension-$\overline{\mathcal{W}}^k(\ell)$ hypercube edges.

We will also use the following properties of prefixes.

4. For any two embeddings k_1 and k_2: $\lambda(\mathcal{W}^{k_1}, \mathcal{W}^{k_2}) = \lambda(k_1, k_2) + 1$.

5. For any level ℓ and any two embeddings k_1 and k_2: $\lambda(\mathcal{H}^{k_1}(\ell), \mathcal{H}^{k_2}(\ell)) = \lambda(k_1, k_2)$.

6. For any two levels ℓ_1, ℓ_2 and r-dimension hypercube: $\lambda(H_r(\ell_1), H_r(\ell_2)) = \lambda(\ell_1, \ell_2)$.

As mentioned earlier a frequent technique in the proof will be to identify all the CCC vertices whose cross-edges (or straight-edges) are mapped to hypercube edges in a particular dimension. We show that the congestion on hypercube edges in this dimension is small by showing that no hypercube node can be the image of more than one or two of these CCC vertices.

Observations 2 and 3 are especially useful in identifying groups of CCC vertices whose images use the same hypercube dimension for cross-edges because they associate CCC levels with the hypercube dimension to which the CCC edges are mapped. For example, Observation 3 and the definition of $\overline{\mathcal{W}}^k$ together show that all cross-edges mapped to hypercube edges in dimension i, $0 \leq i < n$, are at level i. Thus we ask how many level-i CCC vertices can be mapped to a single hypercube node.

Lemma 11 *For any level i, $0 \leq i < n$, and any hypercube node v at most one of the n embeddings defined above maps a level-i CCC vertex to v.*

Proof: A dimension *separates* two sets of hypercube nodes if and only if the value of the hypercube address bit corresponding to this dimension equals 1 for all the nodes in one set and equals 0 for all the nodes in the other set. We show that for any two embeddings there exists a dimension which separates the set of hypercube images of level-i CCC vertices under one embedding from the set of images of level-i vertices under the other embedding. Thus there is no hypercube node to which two embeddings map level-i CCC vertices.

Given any two embeddings, k_1 and k_2, call the images of level-i CCC vertices under the two embeddings V_1 and V_2, respectively. Observation 1 shows that $\mathcal{H}^{k_1}(i)$ is the signature on $\mathcal{W}^{k_1}$ for all nodes in V_1 and that $\mathcal{H}^{k_2}(i)$ is the signature on $\mathcal{W}^{k_2}$ for all nodes in V_2. Observation 5 shows that $\mathcal{H}^{k_1}(i)$ and $\mathcal{H}^{k_2}(i)$ differ on their $\lambda(k_1, k_2) + 1$st bit. Furthermore Observation 4 shows that the hypercube dimension in position $\lambda(k_1, k_2) + 1$ is the same for both $\mathcal{W}^{k_1}$ and $\mathcal{W}^{k_2}$.

Thus this hypercube dimension in position $\lambda(k_1, k_2) + 1$ of both length-r windows separates V_1 and V_2 as desired. ∎

Lemma 11 treated CCC vertices whose cross-edges are mapped to a hypercube dimension less than n. We next examine all CCC vertices whose cross-edges are mapped to a dimension greater than n.

Lemma 12 *For any j, $0 \leq j < r$ and hypercube node v at most one of the n embeddings maps to v a CCC vertex whose cross-edge is mapped to a dimension-$(n + j)$ edge.*

Proof: As in the previous lemma let k_1 and k_2 be any two embeddings and V_1 be the set of hypercube nodes which are images of CCC vertices using dimension $n + j$ as cross-edges under embedding k_1 and V_2 be the images of CCC vertices

using dimension $n + j$ as cross-edges under embedding k_2. We will show that the hypercube dimension in position $\lambda(k_1, k_2) + 1$ of both length-r windows again separates V_1 from V_2.

Let the levels at which dimension $n + j$ is used for a cross-edge in the two embeddings be ℓ_1 and ℓ_2 respectively. When $j = 0$ and thus $\ell_1 = \ell_2 = 1$ we can apply Lemma 11.

When $j > 0$ we observe from the definition of the embeddings that $\ell_1 = 2^j + \rho_j(k_1)$ and $\ell_2 = 2^j + \rho_j(k_2)$. From these representations we see that ℓ_1 and ℓ_2 share $0^{r-j-1}1$ as their first $r - j$ bits and then $\lambda(k_1, k_2)$ more bits from the prefixes of k_1 and k_2. Thus $\lambda(\ell_1, \ell_2) = r - j + \lambda(k_1, k_2)$ and, since $r > j$, $\lambda(\ell_1, \ell_2) > \lambda(k_1, k_2)$.

Applying Observation 6 to the last equation yields $\lambda(H_r(\ell_1), H_r(\ell_2)) > \lambda(k_1, k_2)$. Using the definition of $\mathcal{H}^k$ to write $\mathcal{H}^{k_1} = H_r \oplus k_1$ and $\mathcal{H}^{k_2} = H_r \oplus k_2$ we can infer that $\lambda(\mathcal{H}^{k_1}(\ell_1), \mathcal{H}^{k_2}(\ell_2)) = \lambda(k_1, k_2)$. Since $\lambda(k_1, k_2)$ is the length of the longest common prefix $\mathcal{H}^{k_1}(\ell_1)$ and $\mathcal{H}^{k_2}(\ell_2)$ differ on their $\lambda(k_1, k_2) + 1$st bit.

Since by Observation 1, $\mathcal{H}^{k_1}(\ell_1)$ and $\mathcal{H}^{k_2}(\ell_2)$ are the signatures on the length-r windows of vertices in V_1 and V_2 respectively it follows that the hypercube dimension in position $\lambda(k_1, k_2) + 1$ of both length-r windows separates V_1 and V_2. ∎

Putting Lemma 11 and Lemma 12 together we show that the congestion on cross-edges is small.

Lemma 13 *The congestion due to cross-edges is at most 1, in dimension 1 the congestion is 0.*

Proof: The congestion in dimension 1 due to cross-edges is 0 since no cross-edges are mapped to dimension 1 edges. For dimension 0 or dimension d, $2 \leq d < n$, Lemma 11 guarantees that at most one CCC vertex using dimension d is mapped to any hypercube node. Similarly Lemma 12 guarantees that for $d > n$, at most one CCC vertex using dimension d is mapped to any hypercube node. The single CCC vertex mapped to a hypercube node using a given dimension for its cross-edge contributes congestion of one. ∎

We next turn our attention to straight-edges. From Observation 2 we know that embedding k maps level-ℓ straight-edges to hypercube dimension $\mathcal{W}^k(G_r(\ell))$. From the definition of G_r we can invert this relation to determine which levels are mapped to a given hypercube dimension. For example, dimension 1 is always the first element in $\mathcal{W}^k$ and thus always corresponds to the most significant bit in the gray code. Since the most significant bit is used only at $G_r(n/2 - 1)$ and $G_r(n - 1)$ we can conclude that only straight-edges at level $n/2 - 1$ and $n - 1$ are mapped to dimension-1 hypercube edges.

The remaining dimensions can be divided into r tiers: dimension i is in tier $t = \lfloor \log i \rfloor$. A tier t dimension will always correspond to bit t in the gray code (with msb = 0). A simple fact about reflected gray codes is that bit $t > 0$ is used 2^t times and that for any two levels at which t is used there exists some bit $t' < t$ which is used an odd number of times between these two levels (see Fact 3.)

Lemma 14 *For any dimension i, $1 < i < n$, and hypercube node v at most one CCC vertex which uses the dimension-i hypercube edge for its straight-edge is mapped to v. Furthermore no more than two CCC vertices which use dimension 1 for their straight-edge are mapped to v.*

Proof: Given two embeddings k_1 and k_2, define V_1 to be the set of images of all CCC vertices which use dimension i for straight-edges under embedding k_1 and V_2 be the images under embedding k_2. If i is in tier t and both V_1 and V_2 are nonempty then $\lambda(\mathcal{W}^{k_1}, \mathcal{W}^{k_2}) > t$. (From the definition of $\mathcal{W}^k$ if two embeddings have the same dimension in position t they must have the same dimension in all positions j, $0 \leq j \leq t$.)

Since i is in tier t the nodes in V_1 and V_2 can only be the images of CCC vertices in the 2^t levels at which tier-t dimensions are used for straight-edges. Partition the nodes of V_1 and V_2 into subsets depending on the CCC level of their preimages. Thus V_{1,ℓ_1} is the subset of V_1 containing images of level-ℓ_1 CCC vertices and V_{2,ℓ_2} is the subset of V_2 containing images of level-ℓ_2 CCC vertices. By separating each subset of V_1 from every subset of V_2 we will show that V_1 is disjoint from V_2.

Now let ℓ_1 and ℓ_2 be two levels at which tier-t dimensions are used for straight-edges. If $\ell_1 = \ell_2$ then by Lemma 11 V_{1,ℓ_1} is separated from V_{2,ℓ_2}. On the other hand if $\ell_1 \neq \ell_2$ then because some bit $t' < t$ is used an odd number of times between the two levels it follows that $\rho_t(H_r(\ell_1)) \neq \rho_t(H_r(\ell_2))$. Since $\lambda(k_1, k_2) > t$ we can use the definition of $\mathcal{H}^k$ to infer from the previous equation that $\rho_t(\mathcal{H}^{k_1}(\ell_1)) \neq \rho_t(\mathcal{H}^{k_2}(\ell_2))$. Let $j \leq t$ be one position where the two prefixes differ.

As in the previous lemmas we use Observation 1 to show that the signatures on the length-r windows of nodes in V_{1,ℓ_1} and V_{2,ℓ_2} are $\mathcal{H}^{k_1}(\ell_1)$ and $\mathcal{H}^{k_2}(\ell_2)$. Since $\lambda(\mathcal{W}^{k_1}, \mathcal{W}^{k_2}) > t$ the dimension in position j of both windows separates V_{1,ℓ_1} from V_{2,ℓ_2}.

It is also easily verified that dimension 1 is used for straight-edges at levels $n/2 - 1$ and $n - 1$ in each embedding. Thus by Lemma 11 at most one embedding for each of these two levels can map to v the CCC vertex whose straight-edge is mapped to a dimension-1 hypercube edge. ∎

To complete the proof of Theorem 6 we note that edges in dimension 1 are never used for cross-edges and are used at most twice for straight-edges while

edges in other dimensions are used at most once for straight-edges and once for cross-edges. Thus the overall congestion of two is established. ∎

5.3.4 Extensions

If an undirected version of the CCC is desired then messages must be sent along the straight-edges in both directions (rather than just toward higher levels). By a variant of Lemma 14 this additional traffic will contribute an additional congestion of at most two; increasing the total congestion to four.

A corollary of Theorem 6 is that every graph which is efficiently embeddable within a CCC network has efficient multiple-copy embeddings within the hypercube. It is easy to show that FFTs and Butterflies can be embedded in CCCs with dilation 2 and congestion 2. Thus they also have efficient multiple-copy embeddings into the hypercube.

In Section 5.4 we will be given Q_n and asked to embed the largest CCC having no more than 2^n nodes. When there exists m such that $n = m + \lceil \log m \rceil$ then the expansion will again be optimal. When no such m exists there will be an m with $n - 1 = m + \lceil \log m \rceil$. Thus a single CCC embedding will have expansion 2. However we can embed two CCCs, each in a separate copy of Q_{n-1}, so that each has optimal expansion within its subcube.

5.4 A General Technique

In this section we extend the techniques of Section 5.2 to a more general setting. In particular, starting with a $2^{\lceil \log n \rceil}$-copy[§] embedding of a directed graph G into Q_n the general technique produces a width-n embedding of 2^{n+1} copies of G into Q_{2n}. This transformation has the property that if the one-packet cost of the multiple-copy embedding is c, and δ is the maximum out-degree of any vertex of G, then the n-packet cost of the multiple-path embedding is $c + 2\delta$.

For example, in Section 5.2 we started with a multiple-copy embedding of the length-$2^{n/2}$ cycle and produced a width-n embedding of the length-2^{n+1} cycle. The cost of the multiple-copy embedding is at most 2, the out-degree of each vertex in a directed cycle is 1, and consequently the n-packet cost of the multiple-path embedding is at most 4.

At the end of this section we will apply the general technique to the multiple-copy embedding of the butterfly network. The resulting width-n graph has the property that it yields a width-n embedding of the complete binary tree.

[§] For simplicity we will assume n is a power of two hereafter.

We start with a generalization of the cross-product of two graphs to the cross-product of two sets of graphs. Let $\mathcal{R} = \{R_i \mid i \in \mathsf{Z}_N\}$ and $\mathcal{C} = \{C_i \mid i \in \mathsf{Z}_N\}$ be two sets of graphs, such that each R_i, C_i has vertex set Z_N.

The *cross-product of two sets* $\mathcal{R}$ and $\mathcal{C}$ is the graph (V, E) where $V = \mathsf{Z}_N \times \mathsf{Z}_N$, and the edge set E is defined to be

$$\begin{aligned} E &= \{(\langle i, j_1\rangle, \langle i, j_2\rangle) \mid i \in \mathsf{Z}_N, (j_1, j_2) \in R_i\} \cup \\ &\quad \{(\langle i_1, j\rangle, \langle i_2, j\rangle) \mid (i_1, i_2) \in C_j, j \in \mathsf{Z}_N\} \end{aligned}$$

One can visualize the vertices as arranged in an $N \times N$ grid. The edges in E connect rows and columns so that the subgraph induced by row i equals R_i and the subgraph induced by column j equals C_j.

We pause for one remark regarding our terminology. We say that graph $A = (\mathsf{Z}_N, E)$ ***equals*** graph $B = (\mathsf{Z}_N, F)$ if and only if $E = F$. That is, the graphs are equal if and only if they are isomorphic under the identity mapping on vertices. In general, isomorphic graphs need not be equal.

It is easy to see that if, for all i, R_i equals G and C_i equals H then the generalized cross-product equals the standard cross-product $G \times H$. Note that this is not necessarily true when "equals" is replaced by "is isomorphic to."

In our use of the generalized cross-product, the sets $\mathcal{R}$ and $\mathcal{C}$ will each contain isomorphic copies of the same graph. Before we proceed, we need one more definition. Let $G = (\mathsf{Z}_N, E)$ be a graph and $\phi : \mathsf{Z}_N \mapsto \mathsf{Z}_N$ be an automorphism on Z_N. Then the graph G_ϕ is defined as the graph with vertex set Z_N and edge set $\{(\phi(u), \phi(v)) \mid (u, v) \in E\}$.

Now, consider an n-copy embedding of some graph $G = (\mathsf{Z}_N, E)$ in Q_n. Number these copies 0 through $n - 1$. Each copy is an isomorphic image of G. In other words, the ith copy defines an automorphism ϕ_i of Z_N, such that $\phi_i(j)$ is the address of vertex j in the hypercube under the ith copy.

Having fixed ϕ_i, $0 \leq i < n$, let $R_i = C_i = G_{\phi_{M(i)}}$, where $M(i)$ is, of course, the moment of the number i. Finally, define the *induced cross-product* $X(G)$ to be the generalized cross-product of the sets $\mathcal{R}$ and $\mathcal{C}$.

Theorem 7 *Let G be a graph with maximum out degree δ, and for which there is an n-copy embedding in Q_n with one-packet cost c. Then there exists a width-n embedding of $X(G)$ into Q_{2n} with n-packet cost $c + 2\delta$.*

Proof: The vertex embedding follows directly from the definition of $X(G)$. First we divide Q_{2n} into the cross-product $Q_n \times Q_n$. Next, as in the proofs of Theorem 4 and 5 we view the cross-product as a grid with a copy of Q_n on each row and column. The rows are named by the most significant n bits of their hypercube addresses and the columns by the least significant n bits. Finally we embed R_i in row i and C_j in column j via the identity mapping.

Each edge of $X(G)$ embedded in a row is given width n by replacing it with the following n length-three paths. Suppose the edge is mapped to hypercube edge (x, y) in dimension d, $0 \leq d < n$. For $0 \leq k < n$ the kth path for (x, y) is $x, x \oplus 2^{n+k}, x \oplus 2^{n+k} \oplus 2^d, y$. Intuitively it crosses into a neighboring row, follows the projection of (x, y) in this neighboring row's subcube, and then crosses back into the original row. Similarly if an edge of $X(G)$ is embedded in a column it is mapped to some hypercube edge (u, v) in dimension $n + d$, $0 \leq d < n$. The kth path for (u, v) is $u, u \oplus 2^k, x \oplus 2^k \oplus 2^{n+d}, y$.

It remains to bound the cost of the embedding. Each hypercube node has at most δ edges from R_i in its row special image and uses each directed edge in the column dimensions once for each such edge. Thus, together, the first edges of all the paths use each directed hypercube edge at most δ times. The edges for the C_j place the same load on the row dimensions. Similarly the final edges of all the paths also use each directed hypercube edge at most δ times.

To complete the proof we must show that the cost of the middle edges is bounded by the cost of the n-copy embedding of G. Consider all the middle edges in a particular row. They are each the projection of an edge of $X(G)$ from a neighboring row. Lemma 4 of Section 4.2 and the construction of $X(G)$ guarantee that each neighboring row is a different automorph of G. Furthermore the automorphs of G were constructed so that their combined projections formed the n-copy embedding of G in Q_n. Thus together all the middle edges in this row form the n-copy embedding and can be simulated with cost c. Similarly the middle edges in each column also form an n-copy embedding.

Thus a simulation of the entire multiple-path embedding takes δ steps to simulate all first edges, c steps to simulate all middle edges, and δ steps to simulate all final edges. ∎

5.4.1 Complete Binary Trees

Theorem 8 *For all m and $n = m2^m$ the $(2^{2n} - 1)$-vertex complete binary tree can be embedded in Q_{2n} with width n, $O(1)$ n-packet cost, and $O(1)$ load.*

Proof: Section 5.3 shows that m copies of the butterfly can be embedded in Q_n with $O(1)$ cost. By repeating $n - m$ copies twice the n-copy embedding with $O(1)$ cost required by Theorem 7 is achieved. When Theorem 7 is applied using this multiple-copy embedding of the butterfly the result is a width-n embedding of the generalized cross-product of butterflies (call the product $\mathcal{X}$) in Q_{2n} with $O(1)$ n-packet cost.

We next show that the $2n$-level complete binary tree (CBT) can be embedded in $\mathcal{X}$ with $O(1)$ congestion, dilation, and load. Our main tool will be the fact that the M-node CBT can be embedded in the M-node butterfly with $O(1)$

congestion, dilation, and load [13]. The embedding is simplified by the fact that the embedding in [13] never maps two CBT leaves to the same butterfly node.

We start by embedding the top n levels of the CBT into R_0, the butterfly along the top row of $\mathcal{X}$. Each column of $\mathcal{X}$ receives at most one level-n CBT vertex. The tree is then extended by treating each level-n vertex as the root of a n-level CBT. These subtrees are each embedded in the butterfly corresponding to the column of the root. Since the leaf level of the row tree and the root level of the column trees is the same we have an embedding of the $(2n-1)$-level complete binary tree. We complete the embedding by giving each leaf of a column tree two children in its row's butterfly. One child is mapped to the butterfly neighbor along the cross-edge and the other along the straight-edge to the next higher butterfly level.

The mapping of the first n CBT levels has load equal to that of the embedding in [13] since it corresponds to a single CBT embedding. The next $n-1$ levels have the same load since each subtree is embedded in its own column butterfly. In addition each hypercube may have two CBT leaf nodes mapped to it so the overall load is 2 plus the load due to the embedding of [13]. The edges of $\mathcal{X}$ are each used at most once by the first $2n-1$ CBT levels and once by the last level. Thus the overall congestion on the edges of $\mathcal{X}$ is at most twice the congestion of the CBT to butterfly embedding of [13]. In sum, the n-packet cost and the load of the CBT to hypercube embedding are both $O(1)$. ∎

When a multiple-path embedding of the n-level CBT is desired, and n is not of the form specified by the theorem, a more complicated construction is necessary. For these cases the butterflies will not map bijectively into the factor hypercubes. Thus the embeddings will have larger expansion. In addition the simple approach of embedding first the top n levels and then the bottom n levels of the CBT may not work since the level-n nodes may not be mapped to nodes which are images of column butterfly nodes. While O(1) load and cost is still achievable we omit the proof from this thesis.

5.4.2 Arbitrary binary trees

In [15] it is shown that any (2^n-1)-node, constant-degree tree can be embedded in the n-level complete binary tree with $O(\log n)$ congestion and dilation. By composing this embedding with the multiple-path CBT embedding we achieve a width-n embedding of arbitrary constant degree trees into hypercubes with cost $O(\log n)$. All our previous embeddings had given us $O(n)$ speed-ups over standard embeddings while this embeddings yields $O(n/\log n)$ speed-up.

Recently Aiello and Leighton[2] have discovered an embedding of the cross-product of the n-node complete graph and the n-dimension hypercube into the

$(n + \log n)$-dimension hypercube. They use this embedding to produce $O(1)$ cost multiple-path embeddings of arbitrary trees into hypercubes.

5.5 Large-copy embeddings

When the guest graph is much larger than the hypercube there is often a third way to make efficient use of the hypercube. A single large copy, (containing $n2^n$ nodes), can be embedded into Q_n so that the $n2^n$ vertices are evenly balanced over the 2^n hypercube nodes and the $O(n2^n)$ guest edges are evenly divided among the $n2^n$ directed hypercube edges. We will call such an embedding a *large-copy* embedding. Johnsson and Ho have used large-copy embeddings of grids to speed matrix operations[43, 45].

Large-copy embeddings of cycles are easy to construct. A large cycle is embedded by traversing the edge-disjoint cycles of Lemma 8 in sequence. Each traversal of an edge-disjoint cycle in the hypercube embeds 2^n vertices of the large cycle and no two traversals use the same edge. Thus we get the following corollary:

Corollary 3 *For even n the $n2^{n-1}$-node undirected cycle and the $n2^n$-node directed cycle can each be embedded in Q_n with dilation 1 and congestion 1.*

Large-copy embeddings of CCCs, FFTs, and butterflies are also simple to derive. Standard constructions of each of these graphs expand each node of Q_n into an n-node cycle or path. The n edges associated with each hypercube node are divided among the n nodes which replace it. Thus the degree is reduced to a constant (three for the CCC and four for FFTs and butterflies). The new graph has n times as many nodes as the original hypercube.

When embedding the $n2^n$ node FFT-like graph into Q_n the construction above is simply reversed. Each n-node cycle or path is mapped to the hypercube node from which it was expanded. The edges of the FFT-like graph are spread out evenly among the hypercube edges and an efficient large-copy embedding results.

Lemma 15 *The $n2^n$-node CCC can be embedded in Q_n with dilation 1 and congestion 1 while the $(n+1)2^n$-node FFT and $n2^n$-node butterfly can be embedded with dilation 1 and congestion 2.*

The embeddings in [13] and [15] can again be applied to yield large-copy embeddings of trees from the large copy embeddings of FFTs.

Chapter 6

Beyond Hypercubes

In this chapter we apply our techniques to create multiple-path embeddings for host graphs other than the hypercube. We identify three properties of the hypercube which are important to our techniques: the decomposition into cross-products; the decomposition into Hamiltonian cycles, or more generally into multiple-copy embeddings; and the existence of the moment labelling. After examining each of these properties in turn, we state a general theorem which allows the creation of multiple-path embeddings for any graphs having the three properties. We conclude the chapter by illustrating the general conversion strategy on the *k-ary n-cube*.

6.1 Cross-Product Decomposition

Throughout the preceding chapters we have made frequent use of the ability to decompose graphs into the cross-product of smaller graphs. In Chapter 3.3 we noted that hypercubes and meshes can be naturally decomposed into cross-products and that the height-h complete binary tree is a subgraph of the product of two height-$(h/2)$ complete binary trees. We used these facts to aid in the embedding of meshes and trees into hypercubes; each factor of the guest was embedded into a factor of the host.

The role of cross-products is not, however, confined to allowing each factor of a guest to be embedded in a factor of the host. The cross-product composition supplies a structure which easily accommodates multiple paths. The creation of multiple-path embeddings relies on the ability to route paths through the neighboring subcubes in the cross-product. In this section we take a closer look at the advantages that the cross-product structure brings to network routing.

As a counterpoint to the use of cross-products in producing communication efficient embeddings of fixed, structured communication patterns we look at the ability of a network to route messages between arbitrary pairs of processors. In

order to avoid trivial congestion at the sources and destinations we consider only sets of source-destination pairs in which each processor sends and receives at most a number of messages equal to the degree of the network.

Our use of the cross-product structure to aid in the formation of routes derives from the three-phase routing algorithm for cross-product graphs of Annexstein and Baumslag[6]. (They in turn credit Benes' non-blocking routing scheme as inspiration.) Given a method of routing on each factor graph they produce a method of routing on the product graph. We restate their algorithm in our notation and then extend it to make full use of the communication links.

Permutation Routes

Our goal is to send many messages at the same time. In order to fairly test the network we consider two situations: in the first each processor is allowed to send and receive at most one message, in the second each processor can send and receive a number of messages equal to the number of incident communication links. The second situation makes full use of the communication resources since the number of messages sent is equal to the number of communication links.

> A *permutation mapping* on an N-node network is a set of N source-dest. pairs (s_i, d_i) such that each node is a source in one pair and the destination in one pair.

> A *full-width permutation mapping* on a δ–regular network is a permutation mapping with each pair replicated δ times. A set of routes for a full-width permutation supplies δ paths for each pair.

> A *full mapping* on a δ–regular, N-node network is a set of δN pairs such that each node is a source (resp. destination) of δ pairs. Note that full-width permutation mappings are a special case of full mappings.

> A network is said to be *routable* (resp. *full-width routable, fully routable*) with cost c if for any permutation mapping (resp. full-width permutation mapping, full mapping) a packet can simultaneously be sent from every source in the mapping to its destination in c communication steps.

Bipartite Graph Tools

In order to produce efficient routes we will need to be able to decompose the mappings into matchings. The following bipartite graphs along with Lemma 16 aid in the creation of such decompositions into matchings. (Figure 6.1 provides a pictorial example of these bipartite graphs.)

The *source-to-destination bipartite graph (SDBG)* of mapping Π on network G is a bipartite multi-graph (S, D). For each vertex in G there is a corresponding vertex in both S and in D. There is an edge of multiplicity m between $s \in S$ and $d \in D$ iff there are m instances of the pair (s,d) in Π. The SDBG of a full mapping on a δ-regular network is δ-regular.

For a cross-product graph, $G \times H$, and a mapping, Π, the *factor-SDBG with respect to* G is the bipartite multi-graph (S, D) with the following properties. For each vertex of G there is a corresponding vertex in both S and D. An edge of multiplicity m exists between $s \in S$ and $d \in D$ iff Π contains m pairs whose source has coordinate s in G and whose destination has coordinate d in G. The factor-SDBG with respect to H is defined symettrically.

Lemma 16 (Hall's Theorem[21]) *Any δ-regular bipartite graph can be decomposed into δ perfect matchings.*

Cross-Product Routes

We first illustrate the idea of a full routability and the use of SDBGs with the following simple lemma about complete graphs.

Lemma 17 *K_n is fully routable with cost 2.*

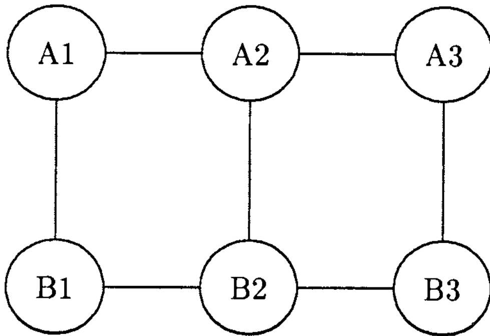
Proof: Create the source to destination bipartite graph of an arbitrary full mapping Π. By Lemma 16 the edges (and thus the pairs) can be decomposed into $n - 1$ perfect matchings. The fact that a pair is a member of a perfect matching means that no other pair in the perfect matching uses the same source or destination.

Π is routed in two phases. In phase one the messages in all pairs contained in the ith matching are routed to processor i. In phase two all messages are routed to their destination.

Since each vertex is the source of exactly one pair from each perfect matching the first phase takes only a single step. Similarly since each vertex is the destination of exactly one pair from each perfect matching the second phase takes a single step. ∎

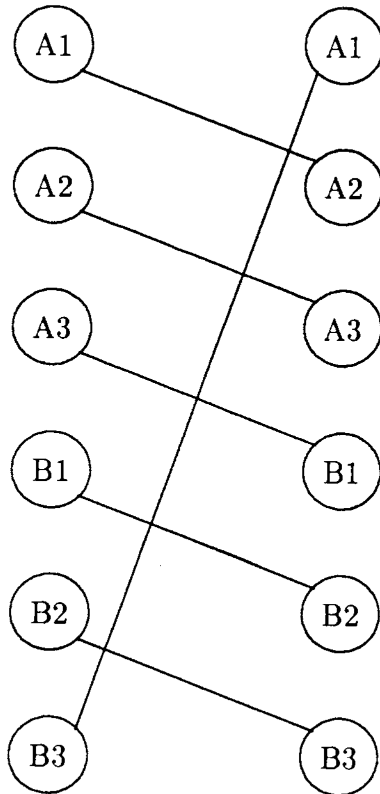
We are now ready to state and prove Annexstein and Baumslag's cross-product routing theorem[6].

Theorem 9 *If G_1 and G_2 are routable with costs $c_1 \leq c_2$ then $G_1 \times G_2$ is routable with cost $2c_1 + c_2$.*

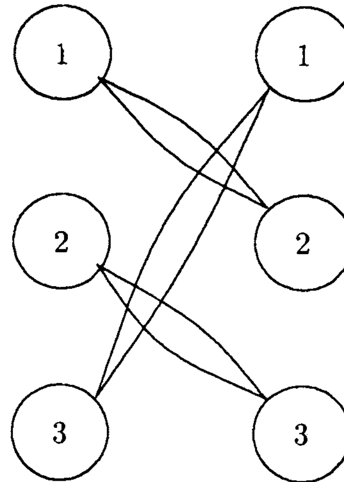


$G = C_2 \times C_3$

$$\begin{aligned} \Pi \;=\; & (A1 \rightarrow A2) \\ & (A2 \rightarrow A3) \\ & (A3 \rightarrow B1) \\ & (B1 \rightarrow B2) \\ & (B2 \rightarrow B3) \\ & (B3 \rightarrow A1) \end{aligned}$$



SDBG of G for Π



Factor-SDBG of G wrt C_3 for Π

Figure 6.1: Source Destination Bipartite Graphs

Proof: If we apply the two phase strategy of first routing along the G_1 components (subgraphs with a fixed address in G_2) and then along the G_2 component (subgraphs with a fixed address in G_1) then we may run into congestion. If all the pairs whose source node have the same address in the G_1 component go to destination nodes which have the same address in the G_2 component then they will all arrive at the same node at the end of the first phase. Thus we add a preliminary phase which spreads the pairs out so that they can be routed along the successive components without congestion in the next two phases.

Given an arbitrary permutation, Π we need to determine how to route this preliminary phase. We wish to route within the G_1 component so that all pairs arriving at a node with the same address in G_1 will have distinct addresses within G_2 for their destinations.

By forming the factor-SDBG of Π with respect to G_2 all pairs within a G_1 component are mapped to a single SDBG node. Consequently, since Π is permutation mapping every vertex of the SDBG will have degree $N = |G_1|$. By Lemma 16 the factor-SDBG can be partitioned into N perfect matchings. The perfect matchings correspond to sets of N source-destination pairs, such that there is exactly one source and one destination from each G_1 component. As we will see this is exactly what we need.

For each pair, $\pi \in \Pi$, let $\rho(\pi)$ be the index of the perfect matching containing the edge corresponding to π in the factor-SDBG. Since both the vertices of G_1 and the perfect matchings have names in the range 0 to $N-1$ a reference to vertex $\rho(\pi)$ of G_1 has a natural interpretation.

Consider the following three phase algorithm for routing Π.

1. Route each pair $\pi = (\langle s_1, s_2\rangle, \langle d_1, d_2\rangle)$ from $\langle s_1, s_2\rangle$ to $\langle \rho(\pi), s_2\rangle$.
2. Next route π from $\langle \rho(\pi), s_2\rangle$ to $\langle \rho(\pi), d_2\rangle$.
3. Lastly route π from $\langle \rho(\pi), d_2\rangle$ to $\langle d_1, d_2\rangle$.

The following properties derive directly from the fact that ρ indexes perfect matchings:

1. Within every G_1 component the mapping induced by step one (each pair, $\pi = (\langle s_1, s_2\rangle, \langle d_1, d_2\rangle)$, maps from $\langle s_1, s_2\rangle$ to $\langle \rho(\pi), s_2\rangle$) is a permutation. Consider any G_1 component, denote by c its address in G_2. All the pairs having source address c in G_2 correspond to edges in the SDBG incident to a single node; they must each be assigned to a different matching and each have a different value of ρ. Thus within each component no two pairs are routed to the same node.

2. Within every G_2 component the mapping induced by step two ($\langle \rho(\pi), s_2\rangle$ to $\langle \rho(\pi), d_2\rangle$) is a permutation. All the pairs which reach a given row

are assigned to the same perfect matching and a perfect matching on the factor-SDBG is the same as a permutation on the factor.

3. Within every column the mapping induced by step three ($\langle \rho(\pi), d_2 \rangle$ to $\langle d_1, d_2 \rangle$) is a permutation. The previous steps are permutations and the entire mapping is a permutation so this step must also be a permutation.

The cost of each phase is now easily bounded. Phases 1 and 3 map permutations on individual copies of G_1 and thus complete with cost c_1. Phase 2 maps permutations on individual copies of G_2 and thus completes with cost c_2. Thus all three phases can complete with cost $2c_1 + c_2$. ∎

Full Cross-Product Routes

In the preceding only one message is sent from each processor and thus at each time step only a fraction (one over the degree of the graph) of the edges are used. Rather than starting with a permutation map for each factor, which makes inefficient use of the edges, we instead assume the ability to fully route the factors. The result is full routability for the product.

Like multiple-path embeddings, full routability allows us to break messages of one-to-one communication into pieces and send them in parallel. In a degree-δ network the full-width version of a permutation allows δ packets to be sent from sources to destinations in parallel. Alternatively each processor can send a separate message to each of δ different destinations as long as no processor is the destination of more than δ messages.

Theorem 10 *If G_1, G_2 of degree δ and size N are fully routable with cost c_1, c_2 then $G_1 \times G_2$ (of degree 2δ) is fully routable with cost* $3\max(c_1, c_2)$.

Proof: The proof is quite similar to that for Theorem 9. Once again three phases will be used, with each phase routing within a single factor. The main difference will be that whereas in the previous theorem a single permutation over the cross-product was broken into three permutations each over a factor it is now necessary to break a full-permutation on the cross-product into full-permutations on the factors.

We start with a full-permutation, Π, on $G_1 \times G_2$. Every vertex is the source and destination of 2δ messages. By forming the SDBG for Π we can partition Π into 2δ perfect matchings, M_i. Call all the messages in the first δ perfect matchings Π_1 and the remaining ones Π_2. Π_1 will be routed on edges derived from G_1 in the first and third phase and on edges derived from G_2 in the second phase. Π_2 will use edges from G_2 in the first and third phase and edges derived from G_1 in the second phase.

We form the factor-SDBGs of each $M_1 \in \Pi_1$ with respect to G_2. Each factor-SDBG is N-regular and can be decomposed into N perfect matchings. As in Theorem 9 each source-destination pair, $\pi \in \Pi_1$ gets a routing class $0 \leq \tau(\pi) < N$.

The routes for each source-destination pair are now almost identical to those in Theorem 9.

1. Route each pair, $\pi = (\langle s_1, s_2 \rangle, \langle d_1, d_2 \rangle)$, to vertex $\langle \tau(\pi), s_2 \rangle$.

2. Continue the path for π by routing from $\langle \tau(\pi), s_2 \rangle$ to $\langle \tau(\pi), d_2 \rangle$.

3. Complete the path for π by routing from $\langle \tau(\pi), d_2 \rangle$ to $\langle d_1, d_2 \rangle$.

We need only to verify that each phase is a full-route on individual factors.

1. Phase one is a full-route on G_1 components. Each vertex is the source of δ pairs since we are routing δ perfect matchings. Consider any vertex (r, c). All the pairs having source column c correspond to edges in the SDBG incident to a single node; only those in matching r of one of the factor-SDBG decompositions of a M_i can have (r, c) as a destination. Since there are exactly δ such matchings, each contributing one pair, the vertex (r, c) (and all other vertices) are the destination of exactly δ pairs in this phase.

2. Phase two is a full-route on G_2 components. Each component of G_2 corresponds to a single node in the factor-SDBGs and thus received δ complete perfect matchings from the previous phase. As in Theorem 9 a perfect matching on the factor-SDBG corresponds precisely to a permutation on G_2.

3. Phase three is a full route on G_1 components. Each vertex is the source of δ routes since it was the destination of δ routes in the previous phase. Each is the destination of δ because Π_1 routes δ pairs to each destination.

The routing of the pairs in Π_2 along edges of G_2 for phases 1 and 3 and edges of G_1 for phase 2 proceeds in the same manner and does not conflict since the edges used for Π_1 are always orthogonal to those used by Π_2. Thus both complements of each phase can be routed with cost $\max\{c_1, c_2\}$ and the entire route has cost at most $3 \max\{c_1, c_2\}$. ∎

Quality of Decomposition

We have now seen two instances, the multiple-path embeddings and the full routes, in which the cross-product structure aids in making use of all the communication links (and thereby, of course, the chip pins). In each of these cases the quality of the result, the width of the multiple-path embedding or the number of messages which can be routed from each node, depends critically on the degrees of the factors.

Higher minimum degree of the factors corresponds to better performance. The best efficiency is achieved if the degrees of the two factors are equal.

6.2 Multiple-copy Decompositions

Besides the decomposition of the hypercube into cross-products we also used its decomposition into edge-disjoint Hamiltonian cycles. In fact, we relaxed the condition that copies be edge-disjoint and generalized the decompositions to multiple-copy embeddings. A cost-c multiple-copy embedding ensures that all copies can send packets to all neighbors in c communication steps. (See Section 5.4.)

As mentioned in the previous chapter the multiple-copy embeddings can be used to amortize the communication costs of several computations which employ the same processors. Ho[37], for example, used edge-disjoint spanning binomial trees to allow multiple nodes to simultaneously broadcast a message to all other nodes. The Hamiltonian decompositions of Alspach, Bermond, and Sotteau[5] can be used to execute multiple pipelined algorithms with parallel communication.

Our use of multiple-copy embeddings in this chapter is less direct. An n-copy embedding of X into G induces n different embeddings of X into G. Call the image of each embedding a *version*. The standard use of a multiple-copy embedding applies all versions to a single copy of G. We will, instead, have many interconnected copies of G and apply a single version to each.

The interconnected copies of G come from a cross-product. The cross-product of G and H induces many copies of G which when treated as supernodes are connected as H. Two copies of G which share a neighbor are called *2-neighbors*. We wish to apply a version of X to each copy of G so that no pair of 2-neighbors are both assigned the same version. In the next section we will discuss how many versions are necessary for this to be possible but for the remainder of this section we assume that we have enough versions.

Once we know that no 2-neighbors are assigned the same version then we can utilize the cross-product structure. The version assigned to each copy of G can *claim* its image edges in all neighboring copies of G. All the claims on image

edges will together form the multiple-copy embedding. When packets are sent over all the length-three paths which start at the tail of a version edge, cross into a neighboring copy of G, traverse the image in G, and return to the head of the version edge then the middle edges of the paths can be simulated in c steps.

The effectiveness of a multiple-copy embedding when combined with a cross-product to create multiple paths is thus directly dependent on its packet-cost.

6.3 Neighbor-Distinguishing Labellings

In order to use multiple-copy decompositions efficiently with cross-products we need to assign different versions to each neighbor of each factor. A *neighbor-distinguishing* labelling does exactly this; a labelling is neighbor distinguishing if no two neighbors of any node are given the same label (alternatively, no 2-neighbors are given the same label). Clearly a neighbor-distinguishing labelling must use at least a number of labels equal to the degree of the graph being labelled.

A coloring of the square of the graph will always be neighbor distinguishing since no two nodes a distance two apart are given the same color. For a degree δ graph $\delta^2 - \delta + 1$ colors suffice. In Chapter 4.2 we gave a neighbor distinguishing labelling for the n-dimension hypercube which uses just n labels when n is a power of two and always uses fewer than $2n$ labels. In Section 6.5 we give a similar labelling of the k-ary n-cube.

The creation of multiple-path embeddings relies on having as many versions in the multiple-copy as there are labels. Clearly fewer labels are better. If there are not enough versions than each can be used several times in return for a multiplicative cost in the congestion. If a k-copy embedding with cost c is used with a labelling which requires $\lambda > k$ labels then a cost of at most $\lceil \lambda/k \rceil \cdot c$ results.

6.4 From Multiple-copies to Multiple-paths

Multiple-copy embeddings and neighbor distinguishing labellings of two graphs can be used to create a multiple-path embedding on their cross-product. As in Theorem 7 of the last chapter a generalized cross-product (the copies of the factors are permuted with respect to each other) of the versions of the multiple-copy embeddings will be induced by the cross-product. By assigning version i to factors with label i it is ensured that multiple paths can be routed for each edge of each embedded version at the required cost.

Theorem 11 *For all graphs G and H; neighbor-distinguishing labellings of G and H; and k-copy embeddings of X and Y into G and H there exists a width*

min $\{\delta(G), \delta(H)\}$ *multiple-path embedding of the generalized cross-product of* X *and* Y *into* $G \times H$ *with cost* $\max\{\delta(X)+\rho(X)\lceil\lambda(Y)/k\rceil, \delta(Y)+\rho(Y)\lceil\lambda(X)/k\rceil\}$. *Where* $\delta()$ *is the degree of a graph,* $\rho()$ *is the cost of the embedding into a graph, and* $\lambda()$ *is the number of labels in a labelling of a graph.*

6.5 k-dimension cubes

We illustrate Theorem 11 by producing multiple-path embeddings of cycles into the k-dimension cube. The k-dimension, length-ℓ cube, $G_{k,\ell}$, is defined as the cross-product of the length-ℓ cycle with itself k times. Clearly $G_{k,\ell}$ has degree $2k$ and ℓ^k nodes.

$G_{2k,\ell}$ decomposes naturally into the cross-product of two copies of $G_{k,\ell}$. Each factor has degree $2k$ and thus multiple-path embeddings based on this decomposition can have width at most $2k$.

As a multiple-copy embedding we will use a Hamiltonian decomposition. The constructions of Alspach, Bermond, and Sotteau[5] allow Hamiltonian decompositions of two graphs to be extended into a Hamiltonian decompositions of their cross-product. Starting with the trivial decomposition of the cycle we can thus produce k edge-disjoint undirected cycles in $G_{k,\ell}$. Each cycle can be directed to yield $2k$ edge-disjoint directed cycles.

The neighbor-distinguishing labelling will be similar to the labelling induced by the moments define in Chapter 4.2. The central idea of the moment labelling was that labels across each dimension differed by a multiple (xor) of a single value. It is not possible for two neighbors in the same dimension to get the same label since there is only one neighbor. When $\ell > 2$ then we need two values to multiply (xor) into the labels for each dimension since nodes have two neighbors in the dimension. Our solution is that along dimension i the label will be alternately multiplied by $2i$ and $2i+1$.

Formally the label of node $a = a_1 a_2 \ldots a_m)$ is

$$L(v) = \bigoplus_{i=1}^{m} a_i \bmod 2 \cdot \mathrm{b}(2i) + \lceil (a_i \bmod 4)/2 \rceil .$$

An argument similar to that in Chapter 4.2 shows that when ℓ is divisible by four no two neighbors of any node are given the same label. There are $2^{\lceil \log 2k \rceil}$ labels used ($2k$ when k is a power of two).

When k is a power of two then there are exactly as many labels as directed, edge-disjoint cycles and a different cycle can be assigned to each label. The construction of the multiple-path generalized cross-product is then the same as in Chapter 5.4. First and last edges of paths will be disjoint because their origins (resp. destinations) are unique. The middle edges will be disjoint because they form Hamiltonian decompositions within each factor cube.

Corollary 4 *For all $k = 2^i$ and $\ell = 4j$ the $2\ell^k$-node cycle can be embedded in $G_{k,\ell}$ with load 2, width k, and k-packet cost 3. For general ℓ and k the cost is 6.*

When ℓ is not divisible by four the strict alternation of multiplication by $2i$ and $2i + 1$ will not be possible. Both neighbors in a dimension of some node will get the same label. When the middle edges of the neighbors are projected into the node's factor the congestion will thus be doubled. An additional doubling of the middle edge congestion arises when k is not a power of two and thus the number of labels is only guaranteed to be greater than half the number of edge-disjoint cycles rather than equal to the number of cycles.

When the cube is constructed from the cross-product of unequal length cycles the transformation is unaffected. Since the degree and number of edge-disjoint cycles remains the same the assignments of labels, special cycles, and length-three paths are all still possible.

Chapter 7

Wire lengths

In the preceding chapters we have assumed that data transmission time across a communication link is independent of the length of the wire. Yet at a clock speed of 40 MHz light can only travel approximately 7.5 meters per cycle. Noakes and Dally[59] estimate that signal paths longer than just six inches require attention to length related effects.

The effect of long wires on communication time depends on how transmission time grows with edge length. Define $f(\ell)$ as the time to transmit a single bit along a wire of length ℓ. The *resistive* model of transmission time argues that the speed of transmission is limited by the speed of light and thus $f(\ell) = \Theta(\ell)$. The *capacitive* model uses "exponential horn" drivers to allow the resistive component to be neglected and thus make $f(\ell) = \Theta(\log \ell)$. In the *constant delay* model it is assumed that other factors in communication time outweigh any length dependent factors and thus $f(\ell) = \Theta(1)$. (For a more detailed discussion see Ranade and Johnsson [69].)

As has been mentioned earlier the one and two dimensional N-node meshes have natural VLSI implementations with unit-length wires and $O(N)$ area. Binary trees admit less perfect implementations. Bhatt and Leiserson [17] showed that VLSI layouts of N-node binary trees with $\Theta(N)$ area and $\Theta(\sqrt{N}/\log N)$ wire length were the best possible. Any layout of an N-node hypercube also requires wires of length $\Omega(\sqrt{N}/\log N)$.

One way of mitigating the effect of long wires is to use them as *transmission lines* rather than as single-cycle connections. Instead of sending a single bit on each clock cycle, bits are placed on the line as a serial signal. When wave spreading is avoided then multiple bits can reside on the wire at the same time. The delay in the wire becomes an additive effect rather than multiplicative; the time to send b bits is $O(\ell + b)$ rather than $O(\ell b)$. The Vulcan group at IBM plans to take this approach[81].

In this chapter we examine the alternative possibility of making the longer communication links have more wires. In essence we will use greater width to

make up for the delay due to longer length. This examination of using increased width to compensate for long wire length necessitates a slight shift in our model. We will still be concerned with the comparative advantages of different networks and with the importance of off-chip pins but the focus shifts to wire length.

7.1 Length-Weighted Design

The major difference from earlier chapters is that in this chapter the number of pins per chip and the number of processors per chip are allowed to grow with the size of the network. The per chip number of both pins and processors in a N-node network is, however, limited to $O(\log N)$. The growth in number of pins and processors allows investigation of the pin resource necessary to compensate for longer wire length.

In previous chapters it was assumed that in a single communication time step one bit could be sent across each pin and transmitted across the attached wire to the neighboring chip. Thus if W pins were assigned to an edge then a length-m message could be transmitted in $\lceil m/W \rceil$ steps. In this chapter the time to send a single bit will be $f(\ell)$ for a wire of length ℓ, where f is the cost function. Thus if an edge of the network is of length ℓ in the layout then a length-m message requires $\lceil m/W \rceil f(\ell)$ time steps.

Having different delays over different wires complicates the maintenance of even a weakly synchronous system such as that described in Chapter 2. Synchrony requires either delaying fast wires to wait for slower ones or changing the routing algorithms to make differential use of the wires depending on their length. Instead we attempt to restore synchrony by using more pins for the longer edges.

If each edge of length ℓ is assigned $f(\ell)$ pins then a length-m message can be sent across any length-ℓ edge in $\lceil m/f(\ell) \rceil f(\ell)$ time steps. If the messages are of length greater than $f(\ell_m)$ then all transmissions takes at most $m + f(\ell) < 2m$ time steps regardless of edge length.

Formally we define:

Definition 2 *A* length-weighted *implementation of an N-node interconnection network is a set of C chips (each containing at most* $\log N$ *nodes) and a layout such that the number of pins assigned to edges of length ℓ equals $f(\ell)$. An implementation is said to be* pin-optimal *if each chip uses at most $O(f(\ell_m))$ pins; where ℓ_m is the minimum over all C-chip layouts of the maximal edge length. The implementation is said to be* chip-optimal *if $C = O(N/\log N)$.*

If $o(f(\ell_m))$ pins are allowed then no length-weighted implementation is possible since there are not enough pins for the longest wire. However, allowing $O(f(\ell_m))$ pins means that any constant-degree graph admits a trivial solution; one chip is used for each node of a standard optimal layout and then $f(\ell)$ pins

are available for *every* edge. Interesting solutions for constant-degree networks thus requiring allowing several nodes to be placed on each chip.

Note that in the logarithmic delay model the wires of trees and hypercubes will require $\Omega(\log N)$ pins. Thus the model change to allow the number of pins to grow with the size of the network is necessary.

7.2 Length-weighted Layouts for Trees

In this section we restrict our attention to logarithmic delay functions. These include the capacitive and constant delay models but not the resistive model.

We first examine arbitrary binary trees as our interconnection network. As noted earlier $\ell_m = \Omega(\sqrt{N}/\log N)$ and pin-optimal, length-weighted implementations which assign one node per chip are trivial. We use the following lemma of Aiello and Leighton[2] to produce implementations which are also chip-optimal.

Lemma 18 (Aiello and Leighton[2]) *For any N-node binary tree T and any positive integer $M < N$, it is possible to partition T into subtrees by removing $M - 1$ edges so that every subtree has at most $\frac{12N}{M} + 1$ nodes and so that each subtree is incident to at most three removed edges.*

By setting $M = N/\log N$, Lemma 18 partitions any tree into $N/\log N$ subtrees, none of which has more than $O(\log N)$ nodes nor more than three connections to other subtrees. This is exactly what we need for a pin-optimal, chip-optimal, length-weighted implementation of binary trees.

Theorem 12 *There exists a pin-optimal, chip-optimal, length-weighted implementation of every binary tree.*

7.3 Length-weighted Layouts for Hypercubes

Length-weighted implementations of hypercubes are more expensive than those for trees. We begin with a lemma which shows that not only are there long edges in any layout of the hypercube but that some node will be incident to many long edges. There are, unfortunately, two distance measures involved: Euclidean distance on the plane of the layout and graph distance within the hypercube.

Lemma 19 *For any $0 < c < 1$ and sufficiently large N every layout of a N-node hypercube has at least one node with $c \log N$ incident edges of Euclidean length at least $N^{(1-c)/5}$.*

Proof: Suppose the lemma were false and that there exists a layout in which every node has fewer than $c \log N$ edges incident to it of Euclidean length greater than $L = N^{(1-c)/5}$. We will show that there exists a node, v, for which this layout must place $N^{(1-c)/2}$ nodes within a Euclidean distance of $N^{(1-c)/5} \log N/2$ from the layout image of v. Since there are only $N^{2(1-c)/5} \log^2 N/4$ such grid points a contradiction is reached.

We start at any node, v, and consider the set S of all nodes which are a graph distance $(1-c)\log N/2$ from v in the hypercube via only edges of Euclidean length less than L in the layout. Each node in S corresponds to a path of graph length $(1-c)\log N/2$ in the hypercube, where each step has a choice of at least $(1-c)\log N$ dimensions to follow. No dimension can be chosen more than once. There are at least $\binom{(1-c)\log N}{(1-c)\log N/2}$ possible paths of this type (the dimensions available may not always be the same so there may in fact be more choices). Using standard approximations for the binomial coefficient we find that there are at least $2^{(1-c)\log N/2} = N^{(1-c)/2}$ nodes in S. But at a Euclidean distance $L(1-c)\log N/2$ in the layout from the starting node there are at most $(L(1-c)\log N/2)^2 = N^{2(1-c)/5} \log^2 N/4$ grid points. For sufficiently large N there are thus more hypercube nodes to be assigned than grid points to receive them and a contradiction is reached. ∎

Theorem 13 *There is no pin-optimal length-weighted implementation of hypercubes.*

Proof:

If each node is assigned to its own chip then Lemma 19 shows that some chip has $\frac{1}{2}\log N$ incident edges each of length at least $N^{1/10}$. In order to be length weighted each will require $f(N^{1/10})$ pins and thus $\Omega(f(N^{1/10})\log N)$ pins are required and the layout is not pin optimal. For polylogarithmic f this number of pins is asymptotically equal to the number of pins used if *every* edge were assigned $f(\ell_m)$ pins. Thus no careful assignment of pins is significantly better than the trivial upper bound.

It might seem possible that by placing many hypercube nodes on each chip that most of the long wires might be placed on chip. Unfortunately, for any set of $s \leq \log N$ of hypercube nodes the number of distinct neighbors is $\Omega(s \log N)$. This is true even if only a constant fraction of the neighbors of each node is considered. Thus for any $b < 1$ then for N sufficiently large $\Omega(s \cdot \log N \cdot f(N^b))$ pins will be required. ∎

Chapter 8

Conclusions

Graph embeddings provide a framework in which to compare the efficacy of communication networks. The necessity of keeping communication local and ensuring that communication links are not overloaded can be formulated as a graph embedding problem. However, the equally important physical costs of network construction are ignored. Hypercubes will always have longer communication links and be able to use fewer pins per link than meshes. If these costs are ignored then biased comparisons result.

We consider two physical costs: pin limitations and wire delays. In order to investigate the limits imposed by the number of pins on a chip we require that the capacity of network edges be inversely proportional to the network degree; ie. edges of the n-dimension hypercube can transmit only $4/n$ times as many bits as mesh edges in a single communication step. In exploring the effects of wire delay we require that each wire be assigned a number of pins proportional to its delay and ask how many pins will be required.

In the pin-limited model we showed that multiple-path and multiple-copy embeddings can allow high-degree networks to overcome the disadvantage of having lower capacity edges. Multiple-path embeddings allow efficient use of the pins by sending messages over many paths rather than using only a single low-capacity path. The multiple-copy embeddings execute the communication of many graphs at once in order to amortize the increased cost due to low-capacity edges. In particular, hypercubes were shown to be able to efficiently simulate meshes, trees, and FFT variants under this model.

In contrast, our investigation of wire delay in Chapter 7 was less favorable for the hypercube. When the number of pins available on a chip is allowed to grow at the same rate as the number of processors on the chip then it is possible to assign more pins to longer wires in a tree layout. For a capacitive/logarithmic wire delay each wire can be assigned a number of pins proportional to its delay; messages over long wires can be divided into many small pieces and arrive at the same time as messages sent in one piece over a single-pin, short wire. In

this logarithmic delay model it is impossible, however, to create pin-optimal, length-weighted implementations of hypercubes; the equalizing of the time to send a message over all links by assigning more pins to longer links will essentially require $O(\log N)$ pins per link in a N-node hypercube.

8.1 Beyond Network Comparisons

The techniques developed to make efficient use of the pins may have independent applications. Once there are multiple paths for each communication link a natural fault tolerance arises. If a redundant data encoding, such as Rabin's IDA[64], is used then messages can be reconstructed even if some of the paths fail.

Multiple paths can also be used to facilitate on-line routing. Aiello, Leighton, Maggs, and Newman[1] have designed dilated butterflies which allow bit-serial connections to be made while guaranteeing that no messages are significantly delayed. An alternative to their embedding of the dilated butterfly into the hypercube is to use the permuted cross-product of cube-connected-cycles from Chapter 5. More recently, Aiello and Leighton[2] discovered an embedding of the cross product of the n-node complete graph and the n-dimension hypercube into the $(n + \log n)$-dimension hypercube which provides a simpler way of achieving bit-serial routes.

8.2 Open Questions

Parameter Tuning

The simplest type of open questions ask whether constants can be improved. Probably the most important such question left by this thesis is to determine whether width-n embeddings of cycles in hypercubes can be created. The embeddings of cycles in this thesis yield only marginal speed-up on current machines. Although the cost is only three, the number of multiple paths on today's largest hypercubes (which have twelve dimensions) is only six. A similar, two-fold speed-up can be achieved by simply sending messages in both directions on the cycle. Larger machines will make the advantage of the multiple paths greater but the missing factor of two could be the margin between being useful in practice and practically useless.

The constants for meshes and trees appear to be too large for practical consideration even with the improvements made by Aiello and Leighton. Perhaps there are alternate techniques which will yield better constants.

Assumptions about Physical Resources

At the end of Chapter 2 we listed some assumptions for this thesis. We decided to concentrate primarily on pin limitations and secondly on wire delays. Even a restriction to these resources left many possible questions. We assumed that pins were dedicated to point to point communication links. The possibility of using shared buses remains to be explored. We concentrated on the delay across wires. An analysis in which the throughput of the wires was the key option would be interesting.

Of course there are many other physical resources to be considered. The complexity of the switches within the network would be an ideal resource to include. The question is how to model it. One possibililty is to generalize the strong switch vs. weak switch of Aiello, Leighton, Maggs, and Newman[1]. They propose that a weak switch can only establish a single input to output path through its node on a single step. Once a path is established data can continue to stream through the path in parallel with other paths. Perhaps the time to establish a path should be made a function of the degree of the network; a larger number of potential input/output pairings would lead to slower path formation.

Another possibility is to limit the number or type of input/output permutations available in each switch. Plaxton[62] attributes the following scheme to Varman and Doshi. The edges at each node are numbered cyclically. A message arriving on one edge can only leave on the next time step only on the next edge. If no new message arrives on the next edge then the original message may be shunted around the cycle and leave on the edge after next on the time step after next. In general, if there are no conflicts with in-coming messages a message arriving on edge i can leave on edge $i + j$ after j time steps. Plaxton examines deterministic sorting on this cyclically restricted hypercube.

Rather than restrict the messages to traversing a cycle of edges a small number of permutations might be allowed. How general a switch is necessary for efficient randomized routing? By using the embedding of the FFT into the hypercube of Chapter 4.3.1 the hypercube can simulate an FFT while requiring only a simple internal route of messages between two incoming wires and two outgoing wires at each node. Any permutation can then be routed using randomized routing on the FFT. However, the hypercube is capable of randomized routing of full-width permutations. How complicated a switch is necessary for this?

Other Assumptions

We made the decision to concentrate on problems which have structured, iterative algorithms. This assumption caused us to think in terms of synchronous time steps. A more asynchronous view, perhaps based on queueing theory might be more appropriate for other problems.

The overall load on the network strongly affects the relative importance of physical parameters. We have assumed a high load, leading to potential congestion and the necessity of maintaining data locality. The concern for locality led us to base our models on graph embeddings. The work-preservation model[47] is an alternative to graph embeddings. A model which allows processes to be relocated depending on network traffic might yield an alternative approach to avoiding congestion. In addition such a model could help in creating strategies for dynamically evolving systems.

Other Networks

In this thesis we have only looked at hypercubes, k-ary n-cubes, and trees as candidates for networks whose performance is affected by physical constraints. Do FFTs, Butterflies, and CCCs admit pin-optimal, chip-optimal, length-weighted implementations? How about Multi-butterflies[7] and Star networks[3]?

Are there multiple-copy embeddings of other constant degree graphs such as shuffle-exchanges or fat-trees into hypercubes or k-ary n-cubes or Star networks? Can the conversion techniques be applied to additional networks? Annexstein and Baumslag[6] point out that although the Star network is not a cross product it has similar properties that allow the creation of permutation routes. Are these properties enough to allow the creation of multiple-path embeddings?

New Networks

An important question is whether the structure we have discovered in existing networks can be used to create new better ones. In order to have more than two paths between two nodes in the hypercube the paths must be at least length three; is there a relative of the hypercube with shorter multiple paths?

The ability to be decomposed into cross products and/or into disjoint Hamiltonian cycles yielded opportunity to make efficient use of edges. There are related products, such as the wreath product. Can they be used in similar ways? Other commonly studied graph properties include path width, tree width, and bisection width; can they be used to aid in network design?

Suppose we know in advance that only certain communication patterns will be used. For example, the computer might be designed for finite difference algorithms which use only five point stencils, nine point stencils, and FFTs. We want to be able to use as many of the communication edges as possible no matter which pattern is being used. Can a network be designed which can use all its edges when any of the patterns is in use?

8.3 Final thought

Ideally there would be a single model which could be used to compare network designs and allow the choice of a best network. The range of algorithmic requirements and physical resources involved, however, makes such a universal model unlikely. Research has instead focused on various pieces of the problem. Most factors are held constant and a one or two parameters varied. In this way a better understanding of the varied parameters can be reached.

In this thesis we chose to look at large scientific computations and see what the effects of pin limitations would be on their communication costs when run on hypercube-based machines. The partition of pins among the logarithmic number of communication links necessitated that even when there was a direct connection between the source and destination of a message that multiple paths be used. We found that, although standard single-path embeddings became congested when multiple paths were used, new multiple-path embeddings could be created which significantly reduced the effects of pin limitations on hypercubes.

Bibliography

[1] W. Aiello, F. Thomson Leighton, B. Maggs, and M. Newman. Fast algorithms for bit–serial routing on a hypercube. In *2nd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 55–64, 1990.

[2] William Aiello and F. Thomson Leighton. Coding theory, hypercube embeddings, and fault tolerance. In *3rd Annual ACM Symp. on Parallel Algorithms and Architectures*, to appear, 1991.

[3] S.B. Akers, D. Harel, and B. Krishnamurthy. The star graph: an attractive alternative to the n-cube. In *Intl. Conf. on Parallel Processing*, pages 393–400, 1987.

[4] Romas Aleliunas and Arnold L. Rosenberg. On embedding rectangular grids in square grids. *IEEE Trans. Comp.*, 31:907–913, 1982.

[5] Brian Alspach, Jean-Claude Bermond, and Dominique Sotteau. Decomposition into cycles I: Hamilton decompositions. Technical Report 87-12, Simon Fraser University, 1987.

[6] Fred Annexstein and Marc Baumslag. A unified approach to off-line permutation routing on parallel networks. In *2nd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 398–406, 1990.

[7] S. Arora, T. Leighton, and B. Maggs. On-line algorithms for path selection in nonblocking networks. In *22nd Annual ACM Symposium on Theory of Computing*, 1990.

[8] Fluid dynamics section. In *Fourth Conf. on Hypercubes, Concurrent Computers, and Applications*, pages 917–981, 1989.

[9] K.E. Batcher. Design of a massively parallel processor. *IEEE Trans. on Comp.*, 29:836–840, 1980.

[10] Paul Bay and Gianfranco Bilardi. Deterministic on-line routing on area-univeral networks. In *31st Annual Symposium on Foundations of Computer Science*, pages 297–306, 1990.

[11] Bob Benner. Personal communication.

[12] S. Bettayeb, Z. Miller, and I.H. Sudborough. Embedding grids into hypercubes. In *3rd Aegean Workshop on Computing*, 1988.

[13] Sandeep N. Bhatt, Fan R.K. Chung, Jia-Wei Hong, F. Thomson Leighton, and Arnold L. Rosenberg. Optimal simulations by butterfly networks. In *20th Annual ACM Symposium on Theory of Computing*, pages 192–204, 1988.

[14] Sandeep N. Bhatt, Fan R.K. Chung, F. Thomson Leighton, and Arnold L. Rosenberg. Optimal simulations of tree machines. In *27th Annual Symposium on Foundations of Computer Science*, pages 274–282, 1986.

[15] Sandeep N. Bhatt, Fan R.K. Chung, F. Thomson Leighton, and Arnold L. Rosenberg. Universal graphs for bounded–degree trees and planar graphs. *SIAM J. on Discrete Math*, 2.2:145–155, 1989.

[16] Sandeep N. Bhatt and Ilse C.F. Ipsen. How to embed trees in hypercubes. Technical Report 443, Yale University, Dec 1985.

[17] Sandeep N. Bhatt and F. Thomson Leighton. A framework for solving VLSI graph layout problems. *JCSS*, 28.2, 1984.

[18] Sandeep N. Bhatt and Charles E. Leiserson. Minimizing the longest edge in a VLSI layout. Technical Report 82-86, MIT VLSI Memo, 1982.

[19] Sandeep N. Bhatt and Charles E. Leiserson. How to assemble tree machines. In F. Preparata, editor, *Advances in Computing Research 2*, pages 274–282. JAI press, 1984.

[20] G. Bilardi, M. Pracchi, and F.P. Preparata. A critique of network speed in VLSI models of computation. *IEEE J. Solid-State Circuits*, 17:696–702, 1982.

[21] Kenneth P. Bogart. *Introductory Combinatorics*. Pitman Publishing Inc., 1983.

[22] S. A. Browning. The tree machine: a highly concurrent computing environment. Technical Report 3760, California Institute of Technology, 1980.

[23] S. A. Browning and C. L. Seitz. Communication in a tree machine. In *2nd Caltech Conf. on VLSI*, pages 509–526, 1981.

[24] Mee Yee Chan. Dilation–2 embeddings of grids into hypercubes. In *Int. Conf. on Parallel Processing*, pages 295–298, 1988.

[25] Mee Yee Chan. Embedding of d–dimensional grids into optimal hypercubes. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 52–57, 1989.

[26] Bernard Chazelle and Louis Monier. A model of computation for VLSI with related complexity results. *JACM*, 32:573–588, 1985.

[27] William J. Dally and Chuck L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comp.*, 36:547–553, 1987.

[28] Alvin M. Despain and David A. Patterson. X-tree: A tree structured multiprocessor computer architecture. In *5th Annual Symp. on Computer Architecture*, pages 144–151, 1978.

[29] John A. Ellis. Embedding rectangular grids into square grids. Typescript 1990.

[30] Charles M. Flaig. VLSI mesh routing system. Technical Report 5241, California Institute of Technology, May 1987.

[31] David S. Greenberg. Minimum expansion embeddings of meshes into hypercubes. Technical Report 535, Yale University, 1987.

[32] David S. Greenberg, Lenwood S. Heath, and Arnold L. Rosenberg. Optimal embeddings of butterfly-like graphs in the hypercube. *Math. Syst. Th.*, 23:61–77, 1990.

[33] Ronald I. Greenberg. Efficient interconnection schemes for VLSI and parallel computation. Technical Report 456, Massachusetts Institute of Technology, August 1989. PhD. Thesis.

[34] William D. Gropp and Edward B. Smith. Computational fluid dynamics on parallel processors. *Computers & Fluids*, 18:289–304, 1990.

[35] Johan Hastad, Frank Tom Leighton, and Mark Newman. Reconfiguring a hypercube in the presence of faults. In *19th Annual ACM Symposium on Theory of Computation*, pages 274–284, 1987.

[36] Danny Hillis. *The Connection Machine*. MIT Press, 1985.

[37] Ching-Tien Ho. *Optimal Communication Primitives and Graph Embeddings on Hypercubes*. PhD thesis, Yale University, 1990.

[38] Ching-Tien Ho and S. Lennart Johnsson. On the embedding of arbitrary meshes in boolean cubes with expansion two dilation two. In *Int. Conf. on Parallel Processing*, pages 188–191, 1987.

[39] Ching-Tien Ho and S. Lennart Johnsson. Embedding hyper-pyramids into hypercubes. Technical Report 667, Yale University, Dec 1988.

[40] Ching-Tien Ho and S. Lennart Johnsson. Spanning balanced trees in boolean cubes. *SIAM J. Sci. Statist. Comput.*, to appear, 1990. also as Yale University Technical Report 508, 1987.

[41] Carl D. Howe. An overview of the butterfly GP1000; a large-scale parallel UNIX computer. In *Third International Conference on Supercomputing, vol 2*, pages 134–141, 1988.

[42] Intel Scientific Computers, 15201 NW Greenbrier Parkway, Beaverton, OR 97006. *iPSC/2: Publication number 280110-001*, 1987.

[43] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J.Par. and Dist. Comp.*, 4:133–172, 1987.

[44] S. Lennart Johnsson. Supercomputers: Past and future. Technical Report 778, Yale University, March 1990.

[45] S. Lennart Johnsson and Ching-Tien Ho. Multiplication of arbitrarily shaped matrices on boolean cubes using the full comunications bandwidth. Technical Report 721, Yale University, July 1989.

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

[47] Richard R. Koch, F. Thomson Leighton, Bruce Maggs, Satish Rao, and Arnold L. Rosenberg. Work-preserving emulations of fixed-connection networks. In *21st Annual ACM Symposium on Theory of Computing*, 1989.

[48] Smaragda Konstantinidou. Adaptive, minimal routing in hypercubes. In *Sixth MIT Conf. on Advanced Research in VLSI*, pages 139–153, 1990.

[49] S. Rao Kosaraju and Mikhail J. Atallah. Optimal simulations between mesh–connected arrays of processors. Technical Report 561, Purdue University, September 1986.

[50] P. Lancaster and M. Tismenetsky. *The Theory of Matrixes.* Academic Press, 1985.

[51] F. Thomson Leighton. Layouts for the shuffle-exchange graph and lower bound techniques for VLSI. Technical Report 274, Massachusetts Institute of Technology, June 1982. PhD. Thesis.

[52] F. Thomson Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[53] Charles E. Leiserson. Area-efficient graph layouts (for VLSI). In *21st Annual Symposium on Foundations of Computer Science*, pages 270–281, 1980.

[54] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computing*, c-34(10):892–901, Oct 1985.

[55] Carver Mead and M. Rem. Cost and performance of VLSI computing structures. *IEEE J Solid State Circuits*, sc-14, 1979.

[56] N. Nayar, G. M. Prabhu, and Charles T. Wright. Implementing graph algorithms on multiprocessors. In *Fourth Conf. on Hypercubes, Concurrent Computers, and Applications*, pages 425–428, 1989.

[57] John Y. Ngai and Charles L. Seitz. A framework for adaptive routing. In *ACM Symp. on Parallel Algorithms and Architectures*, pages 2–10, 1989. Also CalTech Report 5246.

[58] Michael Noakes and William J. Dally. System design of the J-machine. In *Sixth MIT Conf. on Advanced Research in VLSI*, pages 179–194, 1990.

[59] Torstein Opsahl, Ruben R. Steck, and Christopher L. Kuszmaut. Advanced image processing applications on the connection machine. In *Third International Conference on Supercomputing, vol. 1*, pages 453–462, 1988.

[60] M.S. Paterson, W.L. Ruzzo, and L. Snyder. Bounds on minimax edge length for complete binary trees. In *13th Annual ACM Symposium on Theory of Computing*, pages 293–299, 1981.

[61] David Peleg and Eli Upfal. The token distribution problem. In *27th Annual Symposium on Foundations of Computer Science*, pages 418–427, 1986.

[62] C. Gregory Plaxton. Efficient computation on sparse interconnection networks. Technical Report 89-1283, Stanford University, Sept 1989. PhD Thesis.

[63] Frank P. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *CACM*, 24.5:300–309, 1981.

[64] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. Technical Report 02-87, Harvard University, 1987.

[65] M.M. Rai. Unsteady three-dimensional Navier-Stokes simulations of turbine rotorstator interaction. In *23rd Joint AIAA/ASME Propulsion Conf.*, 1987.

[66] Abhiram G. Ranade. Equivalence of message scheduling algorithms for parallel communication. Technical Report 512, Yale University, January 1987.

[67] Abhiram G. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[68] Abhiram G. Ranade and S. Lennart Johnsson. The communication efficiency of meshes, boolean cubes and cube connected cycles for wafer scale integrations. Technical Report 579, Yale University, Nov 1987.

[69] John H. Reif and James A. Storer. Real-time dynamic compression of video on a grid-connected parallel computer. In *Third International Conference on Supercomputing, vol. 1*, pages 453–462, 1988.

[70] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[71] Arnold L. Rosenberg. Issues in the study of graph embeddings. In *Graph-theoretic Concepts in Computer Science, Lecture Notes in Computer Science 100*, pages 150–176. Springer-Verlag, 1981.

[72] C.P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh connected computers. In *18th Annual ACM Symposium on Theory of Computing*, pages 255–261, 1986.

[73] Charles L. Seitz. Multicomputers. Typescript 1991.

[74] Charles L. Seitz. The cosmic cube. *CACM*, 28.1:22–33, 1985.

[75] Charles L. Seitz. Submicron systems architecture project. Technical Report 89-04, California Institute of Technology, March 1989.

[76] Charles L. Seitz. Submicron systems architecture project. Technical Report 90-05, California Institute of Technology, March 1990.

[77] Charles L. Seitz. Submicron systems architecture project. Technical Report 90-14, California Institute of Technology, October 1990.

[78] E. Senata. *Non-negative Matrices and Markov Chains.* Springer, 1980.

[79] Diglio A. Simoni, Barbara A. Zimmerman, Jean E. Patterson, Chialin Wu, and John C. Peterson. Synthetic aperture radar processing using the hypercube concurrent architecture. In *Fourth Conf. on Hypercubes, Concurrent Computers, and Applications*, pages 1023–1030, 1989.

[80] Marc Snir. Personal communication.

[81] Quentin F. Stout and B. Wagar. Intensive hypercube communication, I: Prearranged communication in link-bound machines. Technical Report 9-87, University of Michigan, 1987.

[82] C.D. Thompson. Area-time complexity for VLSI. In *11th Annual ACM Symposium on Theory of Computing*, pages 81–88, 1979.

[83] C.D. Thompson. *A Complexity Theory for VLSI.* PhD thesis, Carnegie-Mellon University, 1980.

[84] James J. Tuccillo and Francis J. Balint. Future needs for supercomputers in weather forecasting. In *Third International Conference on Supercomputing, vol. 1*, pages 453–462, 1988.

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

[86] V.E. Vickers and J. Silverman. A technique for generating specialized gray codes. *IEEE Trans of Comp*, c-29.4:329–331, 1980.

[87] Jean Vuillemin. A combinatorial limit to the computing power of VLSI circuits. *IEEE Trans of Comp*, c-32:294–300, 1983.

[88] A.C. Yao. Some complexity questions related to distributive computing. In *11th Annual ACM Symposium on Theory of Computing*, pages 209–213, 1979.

[89] A.C. Yao. The entropic limitations on VLSI computations. In *13th Annual ACM Symposium on Theory of Computing*, pages 308–311, 1981.