Randomized Routing in Transputer Hypercubes using Virtual Channels

Sramana Mitra

Submitted to the Department of Computer Science
of Smith College in partial fulfillment of the
requirements for the degree of
Bachelor of Arts with Honors

Dominique F. Thiébaut, Faculty Adviser

April 15, 1993

ABSTRACT

A randomized routing algorithm for message passing on hypercube networks is presented, along with its application to transputer networks. The performance of the algorithm is compared to other previously published models, including the shortest path algorithm, Valiant's two-phase randomized algorithm and Greenberg and Hajek's deflection algorithm. The results of a comparative study concentrating on *mean packet delay, queue length, node population, throughput,* and the *distribution of deflection of a packet from its shortest path* are given. The new algorithm is shown to perform better than its counterparts when packet destinations exhibit hot spots.

CONTENTS

LIST OF FIGURES **ix**

ACKNOWLEDGEMENTS **x**

1 INTRODUCTION **1**

1.1 Overview of the Transputer 1

1.2 Multitasking: An Example 5

1.3 Performance Evaluation 7

1.4 Thesis Outline 8

2 RANDOMIZED ROUTING FOR MESSAGE PASSING USING VIRTUAL CHANNELS: THE INEBRIATED ROUTER ALGORITHM **10**

2.1 Background 12

2.2 Virtual Channels and The Experiments 13

2.2.1 Logical to Physical Mapping 16

2.2.2 Synchronization: Phase 1 of The Experiment 16

2.2.3 Phase 2: Routing 18

2.3 Results 20

2.3.1 Effects of Folding High-Dimension Hypercubes Onto Lower-Dimension Ones on the Mean Packet Delays 20
2.3.2 Effect of Folding on Queue Size 22
2.3.3 Effect of Folding on Node Population 26
2.3.4 Tuning The Performance of the Inebriated Router Algorithm 27
2.3.5 Deflection of Packets from the Shortest Path 32
2.4 Conclusions 34

3 ACCURACY OF THE SYNCHRONIZATION METHOD 37

3.1 Introduction 37
3.2 The Experiment 38
3.3 Results 39
3.4 Conclusion 42

4 PERFORMANCE MEASURE OF THE "INEBRIATED ROUTER" IN PRESENCE OF BACKGROUND TRAFFIC 43

4.1 Introduction 43
4.2 The Experiment 44
4.3 Performance Evaluation of the Inebriated Router 46
4.3.1 Deflection Factor 46
4.3.2 Mean Delay per Hop 51
4.3.3 Effectiveness of the Inebriated Router Algorithm 53
4.3.4 Queue Size and Node Population 59
4.4 Comparison of Valiant, Greenberg, Shortest Path and Inebriated Router 62
4.4.1 Mean Delay 63

4.4.2 Throughput 64
4.4.3 Node Population and Queue Size 65
4.5 Conclusions 65

5 CONCLUSIONS **69**

BIBLIOGRAPHY **72**

APPENDIX A. PROGRAM LISTINGS **74**

LIST OF FIGURES

1.1 Block diagram of the transputer 2

1.2 Two different configurations of transputers: (a) Linear chain and (b) Mesh. 4

1.3 Block diagram of the different tasks that control *multitasking* and *multiprogramming* in the prime finding algorithm. (Task 1 controls the relay and Task 2 computes the primes.) 6

2.1 A 3-dimensional hypercube mapped to a 2-dimensional hypercube. Each physical transputer supports the code for two logical nodes. Each hardware link supports two virtual channels. 14

2.2 Software mapping. Main procedure runs logical_main() procedures in parallel, one for each logical node mapped to the transputer. 15

2.3 Block diagram of software controlling a logical node. 17

2.4 Average packet delay incurred by Shortest Path algorithm controlling the routing of messages flooding a 4-dimensional hypercube mapped onto a 4-D, 3-D and 2-D physical hypercubes. 21

2.5 Histograms of queue size seen by incoming packets for 4-D hypercube on 16 physical nodes. 23

2.6 Histograms of queue size seen by incoming packets for 4-D hypercube on 8 physical nodes. 24
2.7 Histograms of queue size seen by incoming packets for 4-D hypercube on 4 physical nodes. 25
2.8 Histogram of node population for the four algorithms under 16 on 16 folding. 26
2.9 Histogram of node population for the four algorithms under 16 on 8 folding. 27
2.10 Histograms of node population for the four algorithms under 16 on 4 folding. 28
2.11 Mean packet delay as a function of number of hot spots and probability p of misdirection. 29
2.12 Mean Delays for the three randomized algorithms in the presence of 1, 2,4 and 8 hot spots in a 4-dimensional hypercube. 31
2.13 Deflection of packets from their shortest path 33

3.1 Accuracy of the computed mean delay in the synchronization of transputers. (Delay from node 0 to node 2) 38
3.2 Accuracy of the computed mean delay in the synchronization of transputers. (Delay from node 0 to node 5) 39
3.3 Accuracy of the computed mean delay in the synchronization of transputers. (Delay from node 0 to node 7) 40
3.4 Accuracy of the computed mean delay in the synchronization of transputers. (Delay from node 0 to node 15) 41

4.1 Modified block diagram of software controlling a logical node. (Highlighted section shows new addition.) 45

4.2 Inebriated Router: Variation of deflection factor with probability and effects of physical to logical mapping ($LMAX = 32$, $PMAX = 4, 8, 16$; Probability of a packet going to a hot spot = 30%; Number of Hot Spots = 1). 47

4.3 Inebriated Router: Variation of deflection factor with probability and effects of physical to logical mapping ($LMAX = 32$, $PMAX = 4, 8, 16$; Probability of a packet going to a hot spot = 100%; Number of Hot Spots = 1). 48

4.4 Inebriated Router: Variation of deflection factor with probability and effects of network size. 50

4.5 Inebriated Router: Variation of Mean Delay per hop with probability and effects of hot probability. 52

4.6 Inebriated Router: Variation of Mean Delay per hop with probability and effects of scaling. 54

4.7 Inebriated Router: Deflection Factor as a function of probability, demonstrating that an optimal probability exists. 55

4.8 Inebriated Router: Optimal Probability based on Effective Delay. . . 56

4.9 Inebriated Router: Optimal Probability based on Throughput. . . . 58

4.10 Inebriated Router: Optimal Probability based on Mean Delay. . . . 60

4.11 Inebriated Router: Variation of Node Population. 61

4.12 Inebriated Router: Variation of Queue Size. 62

4.13 Comparison of the four algorithms in terms of Mean Delay: (Hot Probability = 30%, Background Traffic = 70%; p = 96% (for Inebriated Router), 1 Hot Spot). 63

4.14 Comparison of the four algorithms in terms of Throughput: (Hot Probability = 30%, Background Traffic = 70%; p = 96% (for Inebriated Router), 1 Hot Spot). 64

4.15 Comparison of the four algorithms in terms of Node Population: (Hot Probability = 30%, Background Traffic = 70%; p = 96% (for Inebriated Router), 1 Hot Spot). 66

4.16 Comparison of the four algorithms in terms of Queue Size: (Hot Probability = 30%, Background Traffic = 70%; p = 96% (for Inebriated Router), 1 Hot Spot). 67

ACKNOWLEDGMENTS

"Here, where the familiar sunset
from my window,
and the little white houses
speak to me
of random moments passed,
and of pure dawns forgotten -
yet remembered so well!
Here, where the silent, postered walls
whisper about those nights
I've spent with them -
Alone."

As the time to begin a new chapter approaches, I remember those who helped me shape these happy and successful years:

My father, who has always taught me to aim for the highest and the impossible.

My mother, with whom I have always been in conflict, and yet, who has been my source of silent strength.

And **Dominique**, with whom I have shared every moment of frustration, agony, joy, and glory during this work, and who has brought Computer Science alive for me.

They each made a difference.

CHAPTER 1

INTRODUCTION

In this thesis we present an algorithm for routing messages in a multiprocessor system and evaluate the performance of the algorithm in a network of *transputers*. Inter-processor communication in a multiprocessor system often gives rise to "hot spots" which slow down the transmission of messages in the rest of the system. Randomized routing has been proposed as a solution to this problem. We propose a new randomized routing algorithm and compare its performance to other previously published algorithms. In our experiments, we use Computer System Architect's (CSA) Logical System C as the programming language. In this chapter we present an overview of the basic architecture of a transputer and some relevant features about multitasking and multiprogramming in transputers.

1.1 Overview of the Transputer

The name *Transputer* is derived from two English words that are commonly used in computer jargon. The first one, **Trans**fer, is applied most often in describing communication of information. The other word, not surprisingly, is com**puter**, which is essentially what a transputer is. Very simply described, a transputer (manufactured

by Inmos, Ltd.) is a computer in its own right, having a processor, some memory and up to four I/O ports[1]. Figure 1.1 shows a block diagram of the transputer.

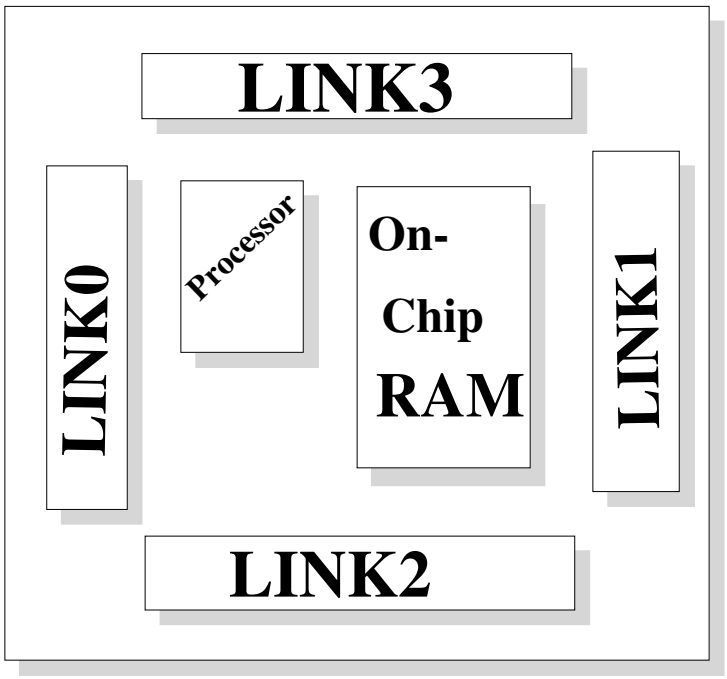

Figure 1.1: Block diagram of the transputer

The IMS T805 has 4 Kbytes of fast internal static memory for achieving higher execution speed. The transputer can also access 4 Gbytes of external memory. The

[1]The transputer comes in several versions. The T800 series that we have been using for our work has four I/O ports although the T400 has only two.

internal and external memory are part of the same 32-bit linear address space. Each transputer in a system uses its own local memory.

Communication between two transputers is achieved by connecting a link interface (I/O port) of one transputer to a link interface on another transputer by two uni-directional signal lines along which data is transmitted serially. The transmission of data over the serial link is *synchronized* and *unbuffered*. By wiring the I/O ports differently, network configurations like *linear chains, meshes,* etc., can be implemented. Figure 1.2 shows a linear chain of 3 transputers and a mesh of 9 transputers. Moreover, the added facility of *virtual channels* (discussed in Chapter 2) provides a wide scope for implementing more complicated networks of transputers.

Communication also needs to take place between processes during *multitasking* and is achieved by means of soft channels. The scheduling of processes in a multitasking environment requires timers to keep track of the quantum of time for which a particular task may hold the processor. The transputers have two 32-bit timers, corresponding to two priorities. The *High Priority Timer* ticks every microsecond, while the *Low Priority Timer* ticks every 64 microseconds. The timers are directly related to the priority levels of the tasks. A *High Priority Task*, once it gets hold of the processor, runs till one of the following happens: *i)* the task finishes; *ii)* the task blocks on I/O; or *iii)* the task is asked to wait until a certain time (a value of the High Priority Timer). A *Low Priority Task*, on the other hand, deschedules itself as soon as its quantum runs out. We use the *High Priority Timer* to keep track of the delay in passing messages from one node to another in our experiments with interprocessor communication because of its higher accuracy. Chapter 2 includes an

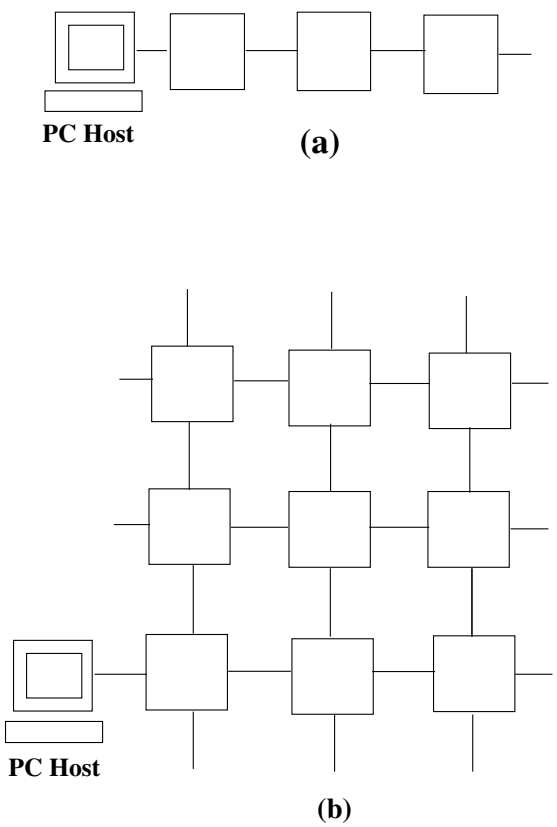

Figure 1.2: Two different configurations of transputers: (a) Linear chain and (b) Mesh.

extensive discussion on this feature.

1.2 Multitasking: An Example

The following example gives an idea of how multitasking and multiprogramming are implemented in software using the features of the transputer and CSA's Logical System C.

Suppose that we want to compute the number of primes in the range 1 to 100. We may use both multiprogramming and multitasking to do this with a high processing speed. We may use each transputer in a 2-node network to do half the work. For instance, transputer 1 computes primes in the range 1 to 50, while transputer 2 computes those in the range 51 to 100. The tasks that need to be performed by each node include *i)* computing the primes and *ii)* transmitting the results to the **host**[2]. We may start two parallel tasks to do these two jobs.

Clearly, it is possible for a single task to do both *(i)* and *(ii)*. In that case, the computation of primes would have to stop intermittently even when there is no relaying to be done. This is where the performance gain through multitasking comes in. With two parallel tasks, the relaying task can be descheduled whenever there are no primes to transmit, and computation of primes may go on uninterrupted. The same code runs on each node and transfer of information takes place via hard channels (signal lines connecting the I/O ports). Figure 1.3 gives a block diagram of the different tasks and processes for this example.

[2]The transputer node that is connected to the PC Host is called root.

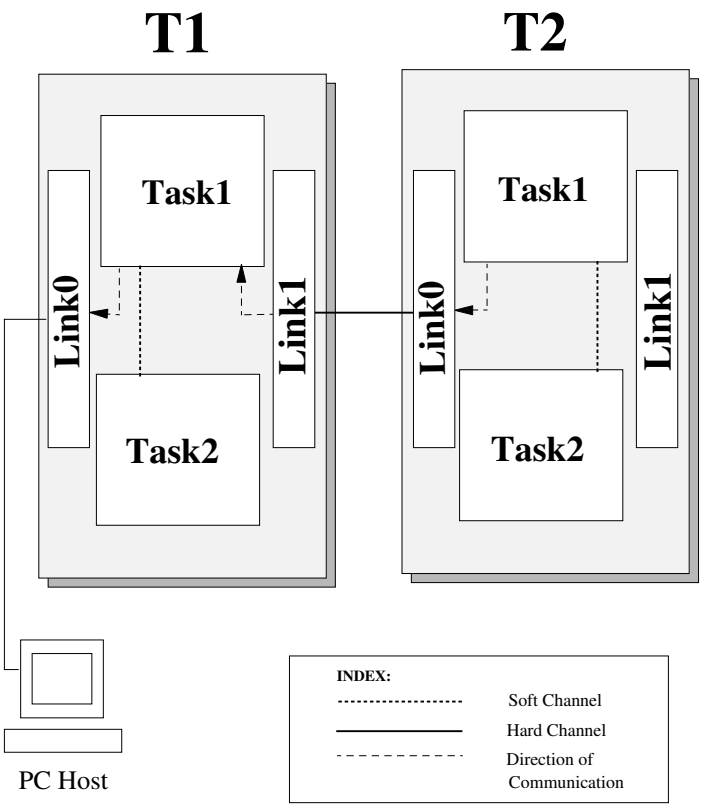

Figure 1.3: Block diagram of the different tasks that control *multitasking* and *multiprogramming* in the prime finding algorithm. (Task 1 controls the relay and Task 2 computes the primes.)

1.3 Performance Evaluation

In our experiments, we look at several different measures of performance: *mean packet delay, queue length, node population,* and the *distribution of deflection of a packet from its shortest path.* We also look at *throughput* and *effective delay.*

Average Latency [Dal90], otherwise referred to as *Mean Packet Delay* is defined as:

$$\text{Average Latency} = T_{net} + T_{node} \tag{1.1}$$

where

$$T_{net} = \text{Network latency}$$

and

$$T_{node} = \text{Latency due to the processing node}$$

The time needed for the actual transmission of the message would thus be included in T_{net} and the time needed for buffering, routing and selecting the channel in which to send out a message would be a part of T_{node}.

Dally also provides a model for measuring T_{net}:

$$T_{net} = T_c(D + L/W) \tag{1.2}$$

where T_c is channel cycle time, D is mean distance between 2 nodes, L is average message length and W is channel width. These models are proposed in the context of *wormhole routing*[3] [Dal90], although they are equally applicable to *store-and-*

[3]Wormhole routing operates by advancing the head of a packet directly from incoming to outgoing channels. Only a few flits are buffered at each node.

forward routing[4] [Tan81]. In our experiments, since we are actually measuring the delays and not simulating, it is not necessary to split up the average latency. Hence, we simply measure *average latency* or *mean delay* by:

$$\text{Mean Delay} = \sum_{i=1}^{N} Delay_i/N \tag{1.3}$$

where N is the number of packets in the network. *Throughput* is defined as the number of packets transmitted per unit time (1 second, in our case). We introduce a new measure called *Effective Delay* and defined as:

$$\text{Effective Delay} = \text{Mean Delay per Hop} * \text{Deflection Factor} \tag{1.4}$$

where

$$\text{Mean Delay per Hop} = \sum_{i=1}^{N} Delay_i / \sum_{i=i}^{N} Hops_i \tag{1.5}$$

and

$$\text{Deflection Factor} = \sum_{i=1}^{N} Hops_i / \sum_{i=1}^{N} Shortest\ Path_i \tag{1.6}$$

When a packet travels from *node i* to *node j* and on the way to its final destination gets relayed by k intermediate nodes, it is said to have $k + 1$ *hops*.

1.4 Thesis Outline

We use the above-mentioned concepts in our experiments in Chapter 2. We implement hypercubes of dimensions up to four (16 nodes) by using *virtual channels*.

[4] In Store-and-forward routing the whole packet is buffered at a node and then transmitted to the next node.

Virtual channels use the same physical hardware links to communicate between processors, but can be defined as separate logical entities. Their actual implementation remains transparent to the user. Virtual channels may connect nodes that are physically remote and not neighbors. The software layer makes sure the message going from one of the nodes to the other uses the shortest route. We investigate the effects of mapping logical hypercubes of larger dimensions to physical hypercubes of smaller dimensions by using virtual channels. Our results indicate a linear relationship in the mean packet delay as a function of scaling. The queue size and node population statistics, however, show little variation due to the scaling.

In a complicated network of processors, interprocessor communication may entail larger delays due to "traffic-jams" at certain routes. This is due to the presence of some nodes that receive more packets than others and which are thus referred to as *hot*. In Chapter 2, we explore the effects of introducing randomness as a solution to this problem. We propose a new randomized algorithm for message passing and compare its measured performance statistics with other previously published algorithms.

In Chapter 3 we take a closer look at the accuracy of the method of synchronization described in Chapter 2. In Chapter 4 we analyze the performance of the newly proposed algorithm in greater detail, and present further results of mapping cubes of higher dimensions to those of smaller dimensions. We also introduce background traffic (not directed towards a hot spot) in Chapter 4 and analyze its effects.

CHAPTER 2

RANDOMIZED ROUTING FOR MESSAGE PASSING USING VIRTUAL CHANNELS: THE INEBRIATED ROUTER ALGORITHM[1]

[1]

In this section, we propose a new randomized routing algorithm for hypercube networks and analyze its performance on a transputer hypercube. Earlier published randomized routing algorithms for hypercubes outperform direct routing ones by adopting a two-phase approach. First, packets are sent to randomly selected intermediary destinations. Then the packets are sent from these scattered nodes to their original destination. The randomness introduced in the first phase reduces hot-spots and the resulting contention at relay nodes, yielding improved performance measures, such as mean packet delay or mean queue length. Our algorithm differs from these approaches in that the two phases are merged and the randomness is distributed over time.

When a packet is generated by a source and sent towards its destination, it accomplishes its travel by hopping from relay node to relay node, until it reaches

[1]This chapter has been accepted for presentation at the North American Transputer User Group (NATUG) meeting, May 1993, Vancouver, British Columbia.

the destination node. Our algorithm divides the transmission links leaving each relay node into two groups. The first group, *Group I*, contains the links that offer the *shortest path* to the destination of the packet and are often referred to as *preferred links*. The second group, *Group II*, contains the links that do not. The relay nodes introduce randomness in the routing in two ways.

- Each relay node conducts a Bernoulli trial to select one of the two groups. With a (high) probability p used homogeneously throughout the network, the relay node picks Group I. With a much lower probability (1-p), Group II is chosen.

- Once a group is selected, one of its links is selected at random and the packet is sent on that link. Thus, two levels of randomness control the non deterministic routing of the packets throughout the network.

As the title indicates, this method of routing is similar to having an *inebriated router* which, when faced with several possible choices for routing a message, would always pick one at random and every so often, would send a packet on the wrong path, sometimes even back to the node from which the packet just came. As a result, routed packets travel through two different kinds of random paths. The first kind can be illustrated by setting p to 1. In this case two packets issued from the same source and traveling toward the same destination may follow different shortest paths. The length of the path is identical for both packets, but the nodes visited on the way may be different. The reason is that although Group I is chosen with probability 1, the link selected to carry the packet is chosen at random if more than one such link is available. On the other hand, when p is lower than 1, it is possible

for some relay nodes on the path to route the packet in the wrong direction by choosing a link from Group II. This time two packets with identical sources and destinations may follow paths of different lengths.

In this chapter we explore the influence of the mapping of high dimensional hypercubes onto smaller ones on three performance measures (mean packet delay, queue size and node population) and we show that when the nodes are flooding the network with packets having a few allowed destinations, our algorithm produces mean packet delays that are shorter than those of the other randomized routing algorithms.

2.1 Background

The two-phase random algorithm was first introduced by Valiant [Val82] and applied to various other networks. Mitra and Cieslak [MC87], along with Leighton *et al.* [LMR88] represent such efforts. Greenberg and Hajek [GH92] use an approach similar to ours in a different context. In their *deflection routing* algorithm, relay nodes with output buffers that are non-empty deflect packets by sending them on a randomly selected "empty" link. Their randomness is controlled by the status of the buffers at the relay nodes, while ours is not. To our knowledge, this paper is the first to provide measured performance for the deflection algorithm. Compared to two-phase algorithms, the main advantage of such a method is that it does not require a lengthy first phase of packet scattering, which, although it can be made to add only a constant number of parallel steps, introduces significant overhead. The disadvantage is that reordering delays will be introduced at the destinations where packets may arrive out of sequence, because consecutive packets follow different

paths. However, such delays (very likely significant) can easily be accounted for by using estimates such as the ones presented by Baccelli [BGP84].

Peleg and Upfal have studied *oblivious routing* algorithms [PU90] in which the route of one packet is not influenced by the source or destination of any other packet in the system. Our algorithm differs from theirs in the way we choose the next node to receive a packet. Consequently, the path is independent of the other packets, even though it is influenced by the destination of the packet itself. Aiello [ALMN91] describes algorithms for bit-serial routing on hypercubes, but none of the algorithms is oblivious. Aiello's approach incorporates some degree of randomization in that each packet is first routed to a random intermediate node and then routed to its true destination, using Valiant's original paradigm discussed earlier.

2.2 Virtual Channels and The Experiments

To measure the performance of our algorithm, we conducted experiments on a 32-transputer Superset system manufactured by CSA (Computer Systems Architects) and Version 91.1 of Logical System's C compiler is used to code the different test programs. The recent release of Logical Systems' C compiler supports *virtual channels*, making it appealing to implement hypercubes of degrees larger than 4 (the current limit with 4-channel transputers) and preparing the ground for the T9000, which will support virtual channels in VLSI [MT90]. Hypercubes of degrees larger than 4 can be mapped to 4- or lower dimension hypercubes by collapsing the cubes formed in higher dimensions onto single transputers and by using virtual channels to connect the logical nodes mapped to different transputers.

Figure 2.1 shows a 3-D hypercube mapped onto a 2-D hypercube. The connection

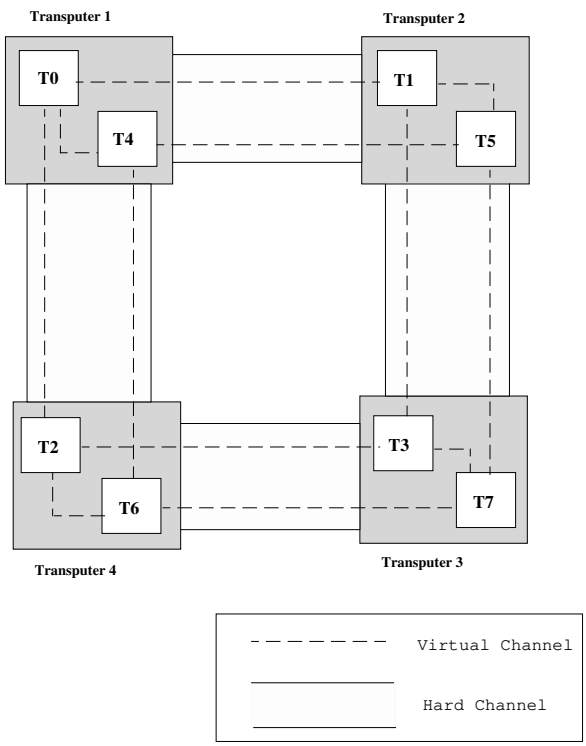

Figure 2.1: A 3-dimensional hypercube mapped to a 2-dimensional hypercube. Each physical transputer supports the code for two logical nodes. Each hardware link supports two virtual channels.

between node T0 and T1, for example, uses a virtual channel sharing the same hardware link as the virtual channel connecting nodes T4 and T5. The connection between T0 and T4, on the other hand is via a virtual channel that is memory mapped, since both nodes are implemented on the same physical transputer. This feature makes it ideal to test the behavior of routing algorithms such as the ones we are considering in hypercubes of varying size.

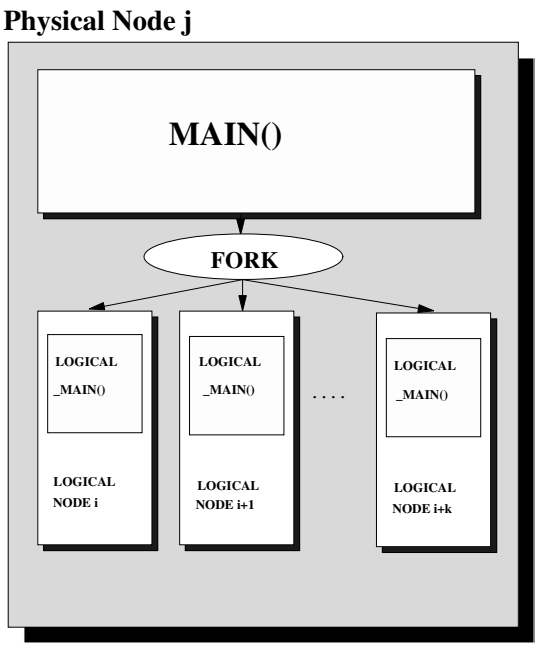

Figure 2.2: Software mapping. Main procedure runs logical_main() procedures in parallel, one for each logical node mapped to the transputer.

2.2.1 Logical to Physical Mapping

In this section we briefly describe our software implementation. In our description of the implementation of the different parallel tasks, we shall use the C terminology that the first procedure that starts the program is called *main()*. Following this convention, figure 2.2 shows how the main program of a particular transputer calls in parallel the *logical_main()* programs representing the logical nodes that are mapped to that transputer. If there are $PMAX$ physical transputers and $LMAX$ logical nodes mapped to them, then each physical node runs $LMAX/PMAX$ logical_main task(s) in parallel. Each logical_main, in turn, starts up a *router* in charge of routing packets on the network and several *input* and *output managers* to control the reception and transmission of packets on the virtual channels.

These three tasks are run at high priority. The logical_main procedure also starts *Task1* and *Task2* (as low priority tasks), which control the generation of new packets (Task1), the reception of messages and the recording of the statistics (Task2). Figure 2.3 shows a block diagram of these different tasks.

2.2.2 Synchronization: Phase 1 of The Experiment

Running the experiments requires 3 phases: synchronization, routing and measurement.

In the first phase, all the transputers must be synchronized so that accurate recording of packet delays is possible. This step is important since the measuring of the delay experienced by a packet going from A to B requires that both A and B tag the packet with a time stamp. Hence, both A and B should maintain the same time.

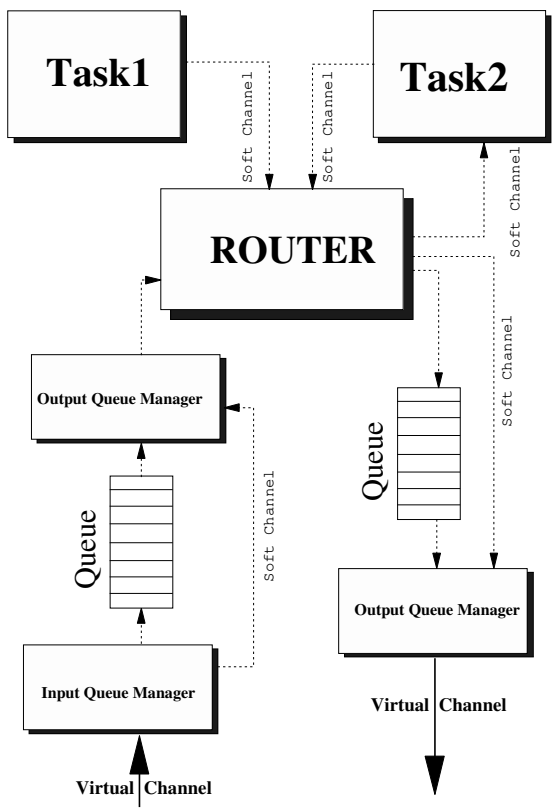

Figure 2.3: Block diagram of software controlling a logical node.

The synchronization is initiated by the Logical Node 0, which first sets its own high priority clock to T_0 and then sends a series of 20 *ping-pong* packets to each node in the network. When a node receives such a packet, it automatically returns it to its sender. This way, Logical Node 0 can measure the average round trip delay and compute an accurate estimate of the time it takes a packet to reach a node. After the 20th ping-pong packet is sent and received, a *Set-Time* packet is then sent, containing the time at which the packet left Logical Node 0, plus the estimated travel time to the destination. Upon receiving this packet, the destination sets its own (high priority) timer to the time value included in the packet just received.

2.2.3 Phase 2: Routing

Once the synchronization is over, the experiment moves on to the *routing* phase, where several packets are sent to random destinations, flooding the network. In each node Task1 sends a predetermined number of packets to random destinations, one after the other. These packets are tagged as *Info* packets and are used to measure packet delays, the sizes of queues entered along the path and the populations of nodes visited.

Although the destinations are chosen at random, the user can control the degree of *hotness* of the network nodes. We define the hotness as the number of nodes that are allowed to receive packets. A hotness of 1 indicates that only one node in the network will receive packets. This node is referred to as a *hot node*. A hotness of *LMAX*, on the other hand, indicates that each node in the network is a potential target. The need for sophisticated routing algorithms stems from the creation of hot spots in networks resulting from the data exchange required by the running

application and it is necessary to create artificial hot spots to rate the effectiveness of the routing mechanism. A hotness of 1 yields the worst case conditions for a routing algorithm, while a hotness of $LMAX$ (uniform distribution) does not present the routing algorithm with any challenge.

In our implementation, the router task communicates with input and output managers via soft channels and is in charge of implementing the details of four different routing algorithms, which we will refer to as the Inebriated Router, Valiant, Greenberg and Shortest Path algorithms.

All input and output virtual channels are buffered and are serviced by the input and output managers, all high priority tasks. The managers communicate with the router via soft channels.

Measurement Conventions

All the data required for our analysis are gathered from the information stored in the *Info* packets. The *Info* packet has fields for recording *source, destination, time sent, time received, number of hops, queue sizes* and *node populations* at the different nodes visited while traveling from source to destination. We refer to these intermediary nodes as *hops.*

In our implementation of the four algorithms we have adopted the following conventions: queue sizes are recorded as soon as a packet enters a queue and node populations as soon as a packet enters a node. The number of hops maintained in each packet is incremented by the router of each node, including the destination node. The *Deflection* of a packet is defined as the difference between the actual number of hops taken by a packet and the number of hops corresponding to the

shortest path.

During all experiments, the packets are sent one after the other, as fast as the network can absorb them. We refer to this method as *flood filling*, or *flooding* the network.

2.3 Results

2.3.1 Effects of Folding High-Dimension Hypercubes Onto Lower-Dimension Ones on the Mean Packet Delays

When several logical nodes are mapped on to one physical transputer, measured statistics may or may not represent the true statistics that would be measured in the unfolded network, and several different mechanisms alter these measured statistics. It is thus interesting to see how the statistics change when high dimension networks are folded onto smaller ones and to see if prediction of the true performance measures is still possible.

The first element that influences the measures is that each transputer is running the code for several logical nodes in parallel. So the computation time available for each logical node decreases with the number of logical nodes per physical transputer.

On the other hand, we can expect communication between logical nodes mapped to the same transputer to be faster, since in this case the channels are memory mapped and the transfer of messages becomes a memory copy operation which is faster than message passing over a physical link.

Figure 2.4 shows the variation of the average packet delays for the Shortest Path algorithm as a function of the number of hops visited by the packets and as a function of the logical/physical node mapping: 16 logical nodes mapped onto a

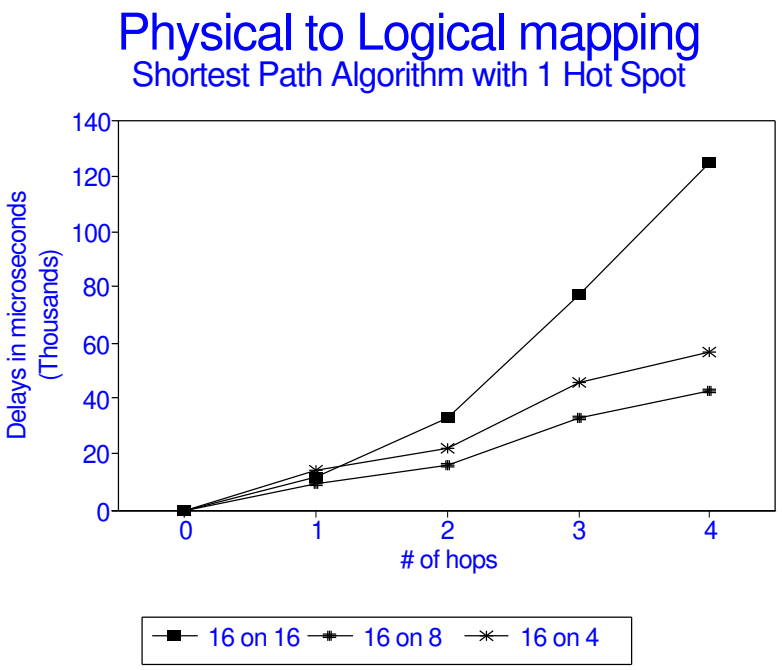

Figure 2.4: Average packet delay incurred by Shortest Path algorithm controlling the routing of messages flooding a 4-dimensional hypercube mapped onto a 4-D, 3-D and 2-D physical hypercubes.

16-node hypercube, onto an 8-node hypercube and onto a 4-node hypercube. This figure shows the 1 hot spot case, where all packets are sent to one destination only. The Shortest Path algorithm is the only one shown here, as the other algorithms exhibit similar behavior. We observe that the delays for the 16-on-16 mapping are the largest and that those for the 16 on 8 case are the smallest, with the 16 on 4 case in-between. We see here the varying effects of the mapping as indicated earlier. The 16 on 8 case presents shorter paths for the packets, while not stressing the transputers much. For the 16 on 4 case, although most paths require on-chip travel, the transputers must run the code for 4 logical nodes in parallel, slowing down the processing of each logical node and yielding longer delays than for the 16 on 8 case. We also observed (data not shown here for the sake of brevity) that when the packets are distributed uniformly (hotness[2] equal to $LMAX$) the delay curves for the same network mapped onto different smaller physical nets are parallel to each other. Clearly this area is very rich in research potential and requires a more thorough investigation.

2.3.2 Effect of Folding on Queue Size

Figure 2.5, figure 2.6 and figure 2.7 show the histograms of the variation of the queue size seen by packets as they enter nodes. The first observation is that the queue size seen by incoming packets is relatively independent of the mapping of the logical nodes to the physical ones. This is an interesting property that we observed throughout our experiments.

Of the four algorithms, the Shortest Path is the one which creates larger queues

[2]Degree of hotness is equal to the number of hotspots in the network.

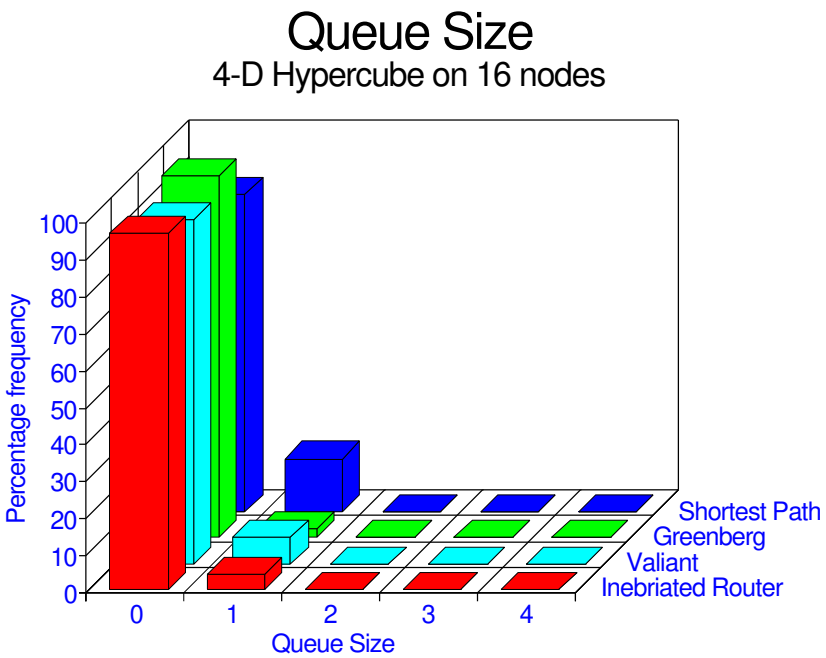

Figure 2.5: Histograms of queue size seen by incoming packets for 4-D hypercube on 16 physical nodes.

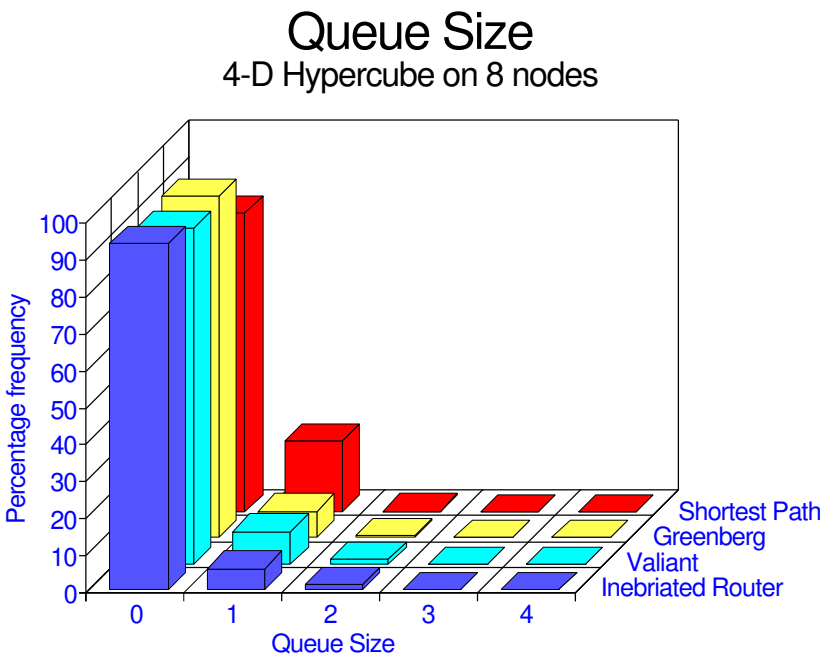

Figure 2.6: Histograms of queue size seen by incoming packets for 4-D hypercube on 8 physical nodes.

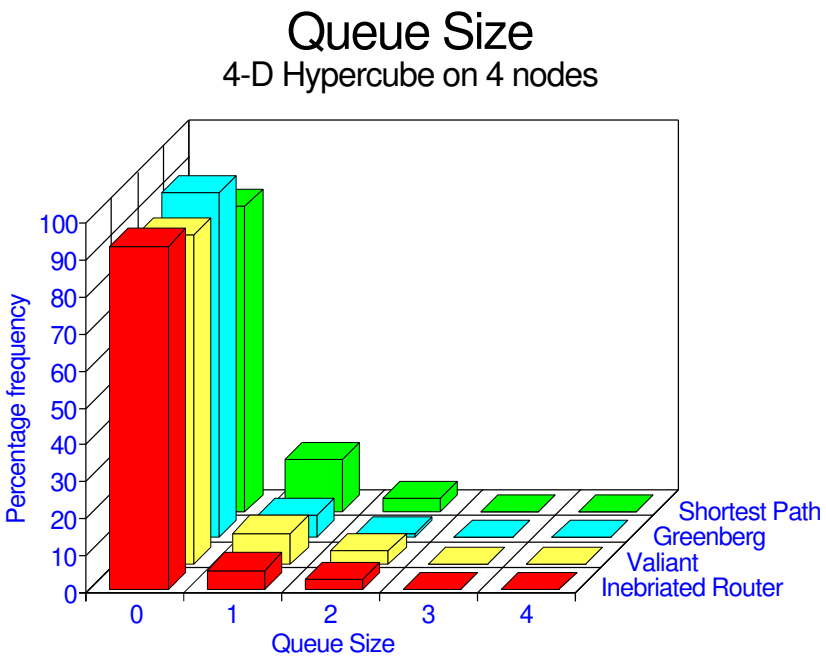

Figure 2.7: Histograms of queue size seen by incoming packets for 4-D hypercube on 4 physical nodes.

since it makes no efforts to avoid possible hot spots. We see that the other three algorithms behave similarly regarding queue size, with a nonsurprising edge for Greenberg and Hajek's algorithm which is designed to avoid non empty queues.

2.3.3 Effect of Folding on Node Population

The node population represents the total number of messages present in a node when an incoming packet arrives at that node, including itself. Figure 2.8, figure 2.9 and figure 2.10, shows the histograms for the node population for the four algorithms, under the same three mappings used in the previous section.

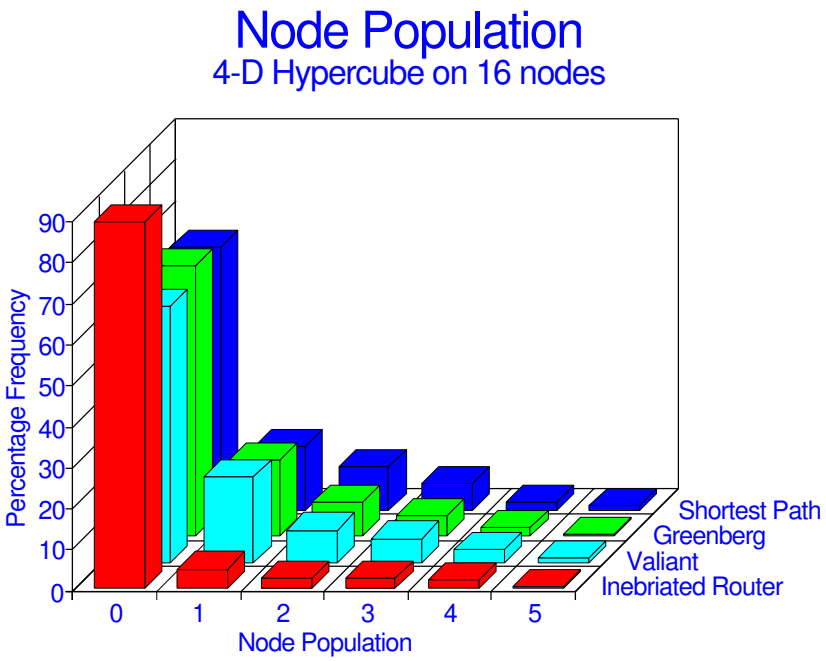

Figure 2.8: Histogram of node population for the four algorithms under 16 on 16 folding.

Here again we observe that the histograms show almost consistently the same

trends when algorithms are compared to each other and across populations. The exception lies in the scale of the population encountered, with maximums of 9, 6 and 5, for the 16-on-4, 16-on-8 and 16-on-16 mappings, respectively.

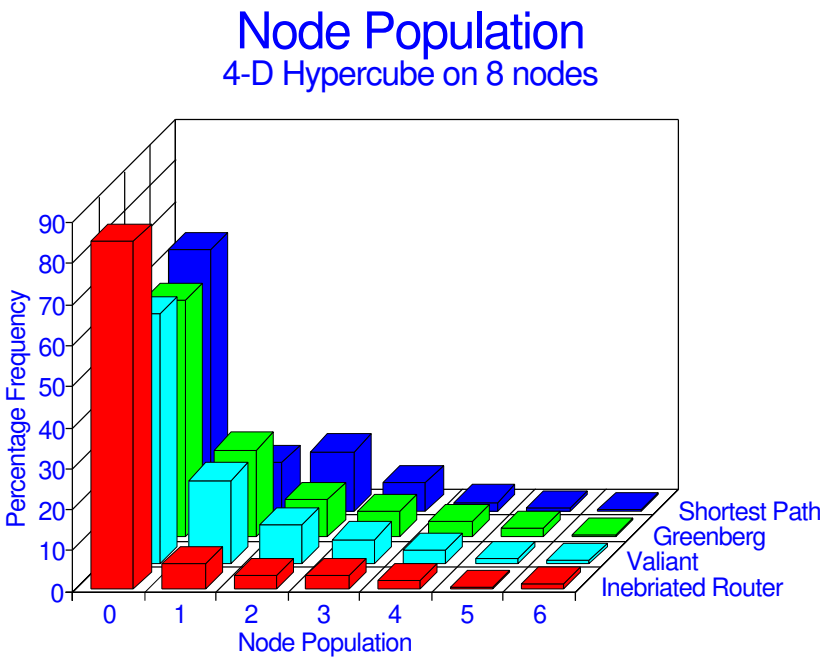

Figure 2.9: Histogram of node population for the four algorithms under 16 on 8 folding.

In terms of absolute performance, we observe that the Inebriated Router algorithm performs best for total node storage, while Valiant and Greenberg maintain the highest node population over the three mappings.

2.3.4 Tuning The Performance of the Inebriated Router Algorithm

The performance of the Inebriated Router algorithm depends on the probability 1-p used to misdirect packets. When p is 1, the algorithm degrades simply to the

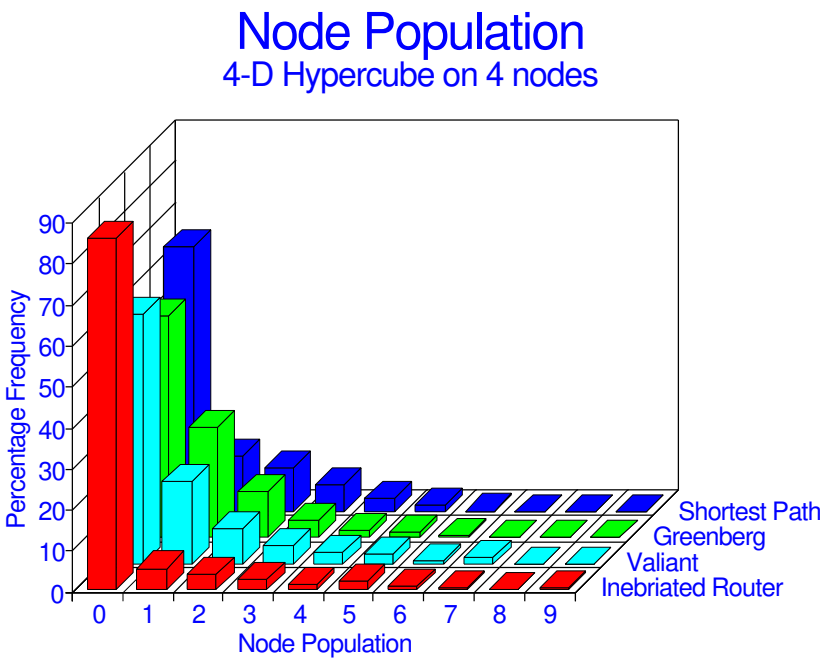

Figure 2.10: Histograms of node population for the four algorithms under 16 on 4 folding.

Shortest Path algorithm. As p decreases however, more packets are misdirected and in cases when the network contains hot nodes, performance can be gained.

In the remainder of this section we will concentrate on the mean packet delay as the target measure of performance.

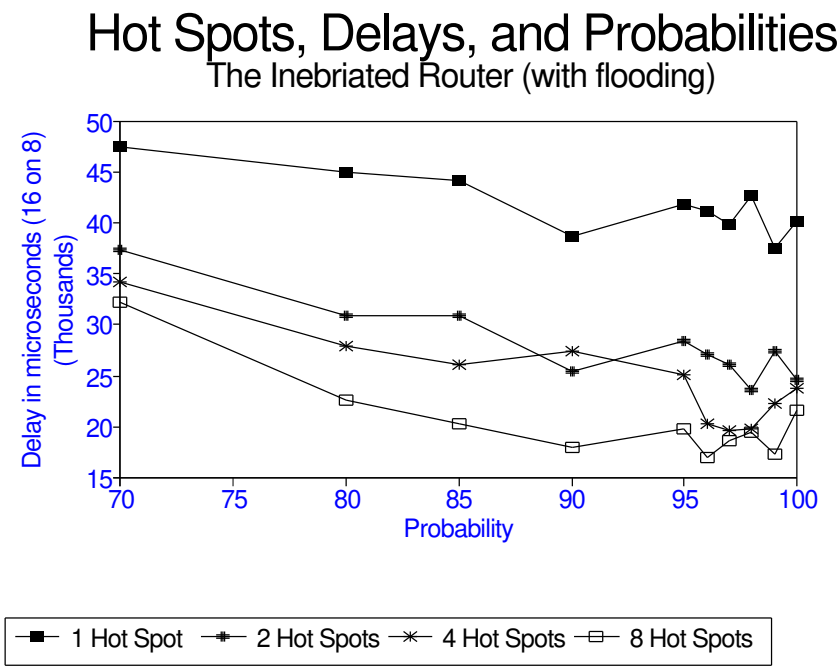

Figure 2.11: Mean packet delay as a function of number of hot spots and probability p of misdirection.

To select the best value of p for a particular case, we measured the mean packet delay, independent of the number of hops, for several probabilities p, and for several possible numbers of hot spots. The resulting graph is shown in figure 2.11. The jagged shape of the curves may be due to the fact that the mean delay is computed as the average of all the delays, independent of the number of hops. We will present a more refined analysis taking into account the hops in Chapter 4.

It is interesting to note that, as we would expect, all four curves seem to exhibit a minimum. When p is 1, the path selected is always the shortest one and the delays measured are shortest-path delays. When p decreases, however, messages start to suffer misdirection, and the farther they go, the more likely they will experience misdirection. As a result, the paths start to lengthen, at random and the congestion at nodes close to the hot spots is reduced. The mean delays decrease. But, as p becomes even smaller, the packets get misdirected too often, to the point that some may travel a long time before they finally "hit" their destination. The delays start increasing.

Therefore, we can expect that for each hot-spot configuration there exists a best probability that yields the lowest delays and figure 2.11 is an indication that such a probability exists.

To test the performance of the inebriated router algorithm against the other two randomized routing algorithms of Valiant and Greenberg, we selected the best probability for each hot spot configuration and ran our experiments, measuring the mean packet delays for all three algorithms.

Figure 2.12 shows the mean packet delays for the three randomized algorithms when the network is flooded with packets going to 1, 2, 4 or 8 hot spots in a 4-dimensional hypercube.

Two interesting observations can be made. The first one is that both the Inebriated Router and Greenberg perform better than Valiant for 2 or more hot spots. When the network sends messages to 1 hot-spot, the Greenberg algorithm performs worst. The possible explanation is that the queues leading to the hot spot start becoming non empty early on, which forces Greenberg and Hajek's routers to create

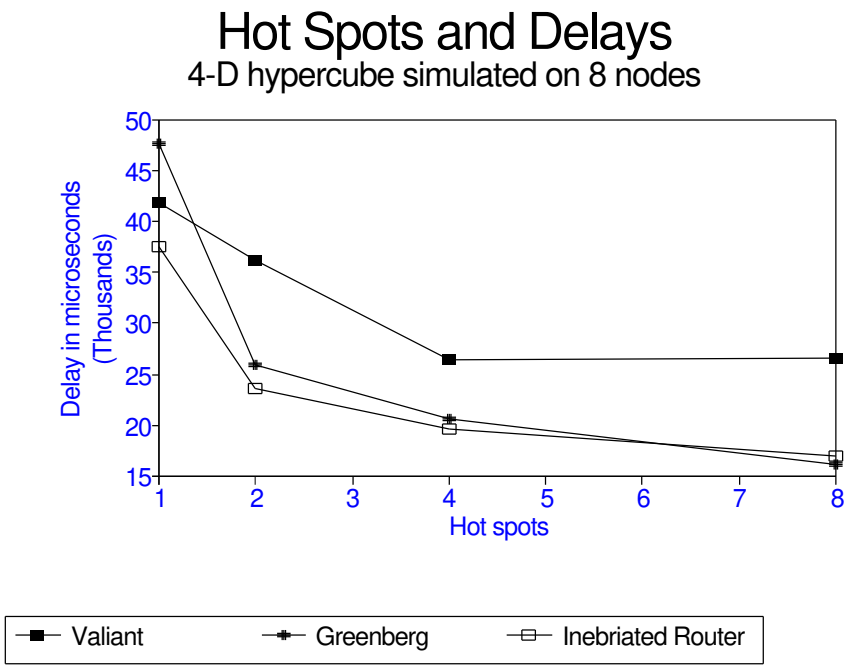

Figure 2.12: Mean Delays for the three randomized algorithms in the presence of 1, 2,4 and 8 hot spots in a 4-dimensional hypercube.

an artificial flow of packets in the colder part of the network[3]. This results in packets exhibiting longer average delays.

The second observation is that the Inebriated Router algorithm performs better than Greenberg for 4 and fewer hot spots and slightly worse for 8 hot spots. This is good news for the proposed algorithm, but the reader should remember that for each network hotness, we tuned our algorithm by selecting the best probability for our algorithm, while Greenberg dynamically adjusts its routing to avoid contention. Nevertheless, the figure shows that the mean packet delay can be reduced with our scheme.

If the network has packets traveling only towards the hot spot, then the Inebriated Router algorithm does not perform as well as the Shortest Path algorithm. All the packets *have* to reach the same destination node and the random scattering does not help. The time spent in waiting in the queues is compulsory and the random deflection does not improve on Shortest Path routing.

2.3.5 Deflection of Packets from the Shortest Path

The last measure of interest is the deflection of packets from their shortest path. Since the randomized routing schemes introduce randomness in the routing, some packets may find themselves circulating not only far from their ideal route, but, as we saw in the previous section in the case of the Greenberg algorithm, may be kept

[3]Greenberg's original work considers only queues of size 1 and the router selects a longer path for a packet when the queues leading to the shortest paths are full. In our case, since we have queues with capacities larger than 1, we adopted non empty queues as the threshold for the router. It is possible that a different threshold may improve the observed performance of Greenberg and Hajek's algorithm in this case.

circulating in the colder part of the network until the congestion is removed around the hot nodes.

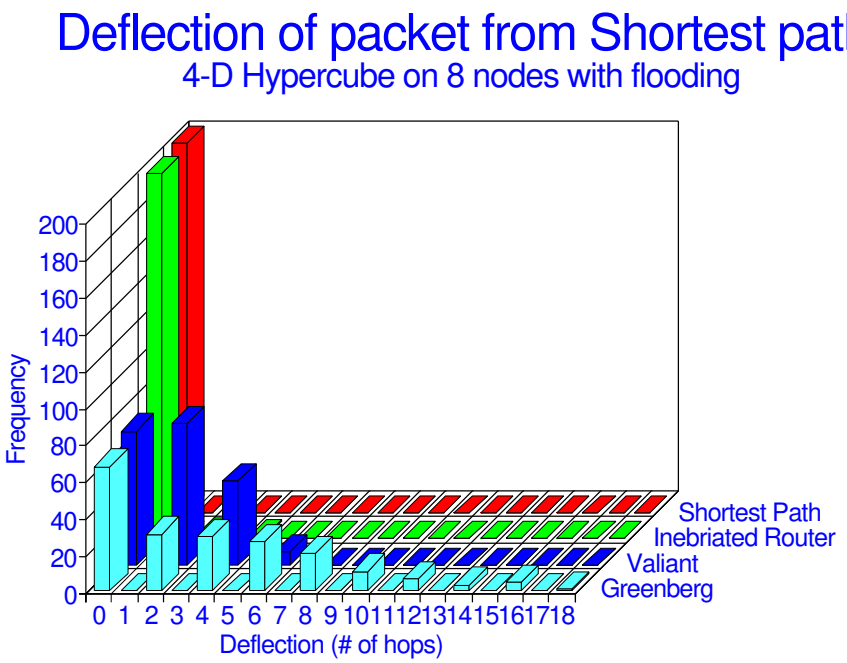

Figure 2.13: Deflection of packets from their shortest path

We compute the deflection as the difference between the actual number of hops taken by a packet and the number of hops on the shortest path. Figure 2.13 shows the histogram of the distribution of the amount of deflection incurred by packets for the four algorithms. The network is a 4-dimensional hypercube mapped onto a 3-D hypercube. By definition, the Shortest path algorithm exhibits no deflection, which translates in the histogram in a 100% bar for 0 hops. For the network considered and the number of hot spots (only 1), the Inebriated router exhibits almost no deflection from the shortest path. This is a particular case due to the high probability p that

had been chosen for this network and degree of hotness.

The interesting data in this histogram, however, appear in the deflection for the Greenberg and Valiant algorithms. We see here the justification of our previous remark on Greenberg and Hajek's algorithm in networks with a small number of hot spots. A significant number of packets are deflected far from their path. In a 4-dimensional network, the shortest path may contain 1, 2, 3 or 4 hops. The data shown in figure 2.13 shows that some packets may go through 18 additional hops on their journey to their destination under Greenberg and Hajek's algorithm. Not unexpectedly, Valiant also exhibits significant deflection due to the two-phase approach, but this deflection is limited, since a packet always uses the shortest path in the first (randomized) and second phase. Note that in both algorithms the deflection is always an even number, since each "wrong" turn taken by a packet on its way to its destination must be negated by another "right" turn to bring it back to a shortest path, due to the network topology.

2.4 Conclusions

We have presented a new randomized routing algorithm for message passing networks and we have measured the performance of the algorithm on a transputer network. The algorithm was compared to two previously published schemes by Greenberg and Hajek and by Valiant. The implementation of the experiments along with the data recorded suggest several interesting points.

- When the physical hypercube hosts a larger logical hypercube, the performance statistics vary differently. The mean packet delay seems to vary linearly, with a slope depending on two factors: the degree of multitasking required of

each physical node (transputer), and the reduced communication time when on-chip communication occurs. The mean queue size and node population, on the other hand, showed similar histograms when algorithms are compared.

- The Inebriated Router algorithm outperforms the other randomized routing algorithms when the network exhibits a few very hot nodes. In this case, the Inebriated Router introduces enough randomness in selecting the path that the traffic of packets around the hot-spots is reduced, resulting in the shortest mean packet delays. It seems that the same conditions create a worst case routing for the Greenberg algorithm which keeps packets circulating away from the hot spots, hence resulting in additional computation time.

Several areas call for improvement and future work. One of them is a better selection of the probability p. The current implementation requires the user to measure the best probability for a particular network and a specified number of hot spots. This method is currently rather crude and we are in the process of refining it. One of the attractive aspects of the Greenberg algorithm is it dynamically adapts to the flow of packets. Such a feature would be useful for the Inebriated Router algorithm.

Maybe the most likely area that we shall concentrate on is the extension of our study to larger hypercubes, of degree five and higher and on a more thorough analysis of the influence of the folding on the different performance measures. Although this extension is theoretically a simple change of a few constants in our program (e.g., $LMAX$), we have found that the size of the code and data structures stored at each node quickly exhausts the one megabyte of memory currently associated with each

transputer. Our next step will thus be to trim the code and fine tune the allocation of memory to processes to allow the cohabitation of several logical nodes on each transputer.

CHAPTER 3

ACCURACY OF THE SYNCHRONIZATION METHOD

In this chapter we investigate the accuracy of the proposed solution to the synchronization problem described in Chapter 2.

3.1 Introduction

In Chapter 2 we described that the different physical nodes in a hypercube of transputers do not use the same clock. Hence, an initial synchronization phase is necessary to ensure that during the actual communication phase, the clocks are all maintaining the same (high priority) time. As a solution to this problem, we proposed exchanging a *ping-pong* message between each node and the root and recording the round-trip delay. We repeat this step N times and compute the average delay. The root then sends a *Set_Time* message to the node it just communicated with, in which it specifies the time as its current time, plus its estimated delay for that node. The receiving node then sets its (high priority) timer to this time.

It is important to know the accuracy of this method because all the experiments that we execute to measure the performances of the different algorithms are depen-

dent on the transputers being synchronized. If not, the *delays* that we measure have no significance.

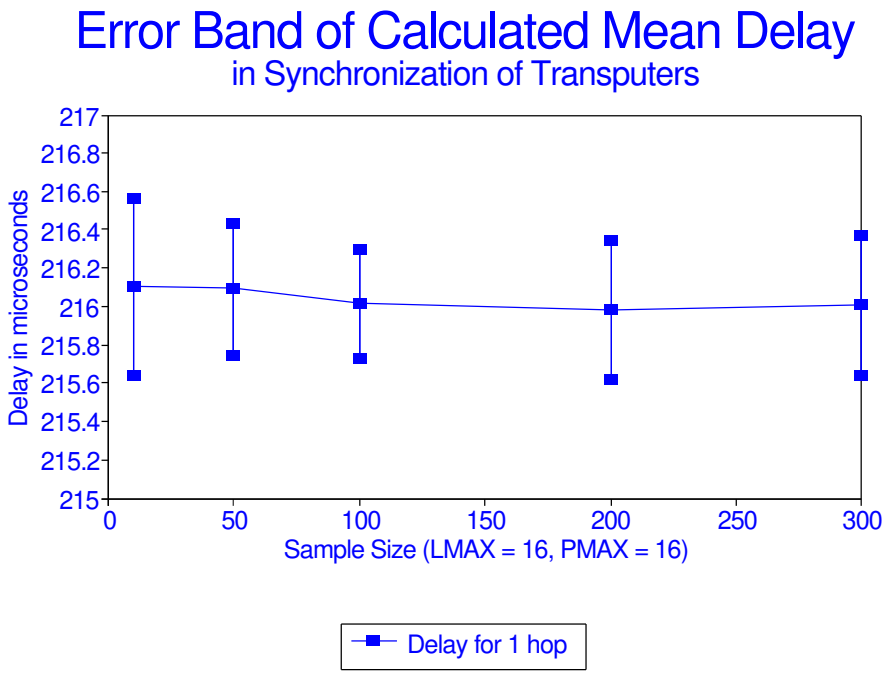

Figure 3.1: Accuracy of the computed mean delay in the synchronization of transputers. (Delay from node 0 to node 2)

3.2 The Experiment

In order to analyze the accuracy of the synchronization method, we have made the root exchange N *ping-pong* packets with each node, N varying from 10 to 300. In figures 3.1, 3.2, 3.3 and 3.4, N is shown as the *Sample Size* and is plotted along the *x-axis*. The *mean delay* for the packets traveling between the root and *node* i is plotted on the *y-axis*. The standard deviation of the delays for each node is also computed. The *mean delay + standard deviation* and *mean delay - standard*

deviation for each sample size are plotted on the graphs and show the error band of the measurement.

3.3 Results

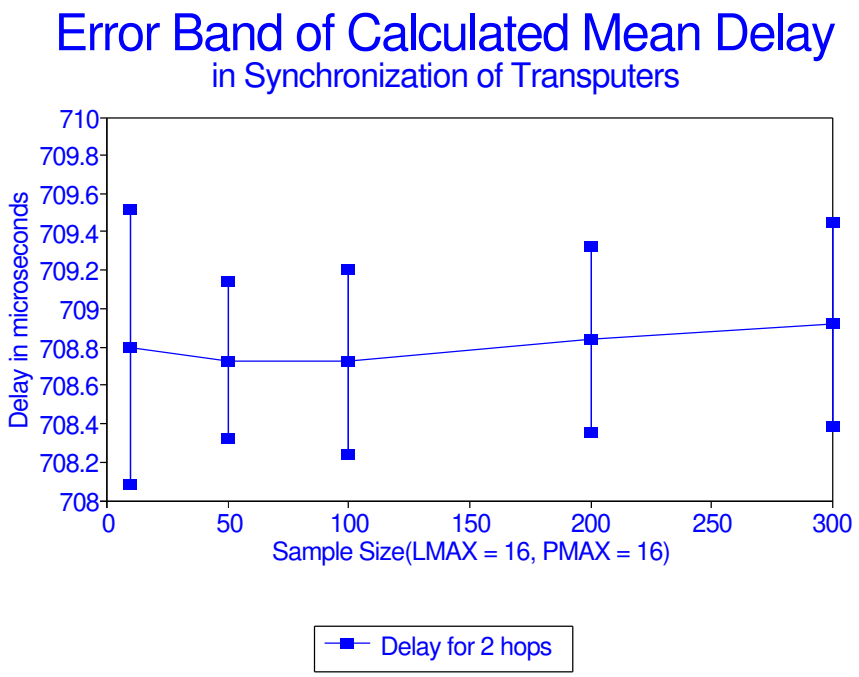

Figure 3.2: Accuracy of the computed mean delay in the synchronization of transputers. (Delay from node 0 to node 5)

The graphs show that this method always computes delays within a 1 microsecond error range. Figures 3.1, 3.2, 3.3 and 3.4 show the error bands of calculated mean delays. As is evident from the graphs, the error bands do not narrow with increasing sample size. Thes sample size of 20 packets used in our experiment yield an accuracy that is acceptable, as it is at most 0.06% worse than if we had chosen a sample size of 50 or 100 packets, which appear to yield the smallest deviation.

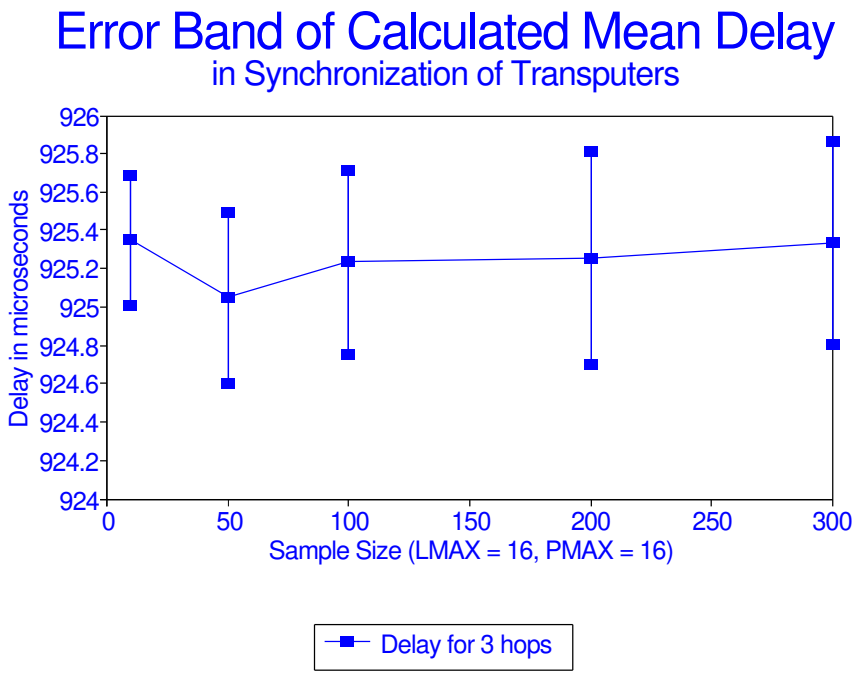

Figure 3.3: Accuracy of the computed mean delay in the synchronization of transputers. (Delay from node 0 to node 7)

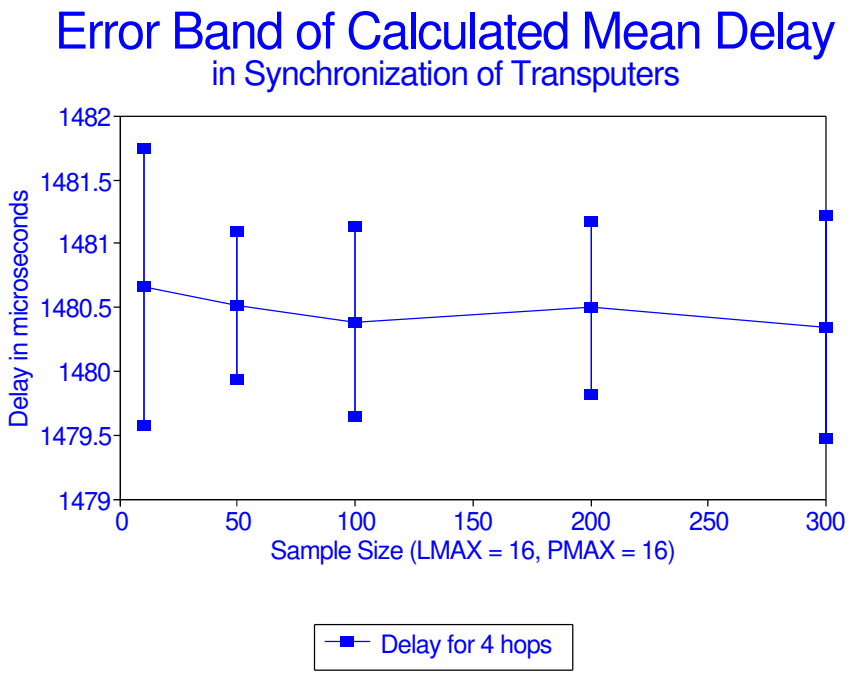

Figure 3.4: Accuracy of the computed mean delay in the synchronization of transputers. (Delay from node 0 to node 15)

3.4 Conclusion

The proposed solution to the problem of synchronizing the clocks on several transputers to the same time ensures an error range that is less than 1 microsecond which is the resolution of our clock.

CHAPTER 4

PERFORMANCE MEASURE OF THE "INEBRIATED ROUTER" IN PRESENCE OF BACKGROUND TRAFFIC

4.1 Introduction

In this chapter, we refine our analysis of the sources of performance gain in the Inebriated Router algorithm. We show that the Inebriated Router algorithm performs well in terms of *mean delay per packet* and *throughput* under conditions where there is substantial background traffic, as well as a *hot spot*. Background traffic is defined as traffic that is not directed towards a "hot" node.

We first describe the few minor changes that we have made in the conditions of the experiment. Then we show that the variation of the deflection factor with probability[1] is independent of $PMAX$ (the number of physical transputers in the network) but is dependent on $LMAX$ (the number of logical transputers in the network). We also present other new results of mapping larger hypercubes of dimensions 5 and 6 to those of lower dimensions. In particular, we show the variation

[1] Henceforth, whenever we refer to *probability (p)*, we imply the probability of the router choosing a *Group I* link (the group of links that offer shortest paths from source to destination).

of mean queue size, mean node population, deflection factor and mean delay per hop under such mappings.

Finally, we compare the four algorithms, namely, Shortest Path, Inebriated Router, Greenberg and Valiant and show that the Inebriated Router performs better than the other algorithms under some conditions, in particular when levels of background traffic are high and when the network exhibits one hot spot. In Chapter 2, we have shown that the Inebriated Router is competitive with Valiant and Greenberg in presence of hot spots, but not with Shortest path. In this chapter, we show that in presence of background traffic, the Inebriated Router does perform better than Shortest Path.

4.2 The Experiment

For the experiments described in this chapter, we have used a modified version of the code used in the experiments of Chapter 2. Figure 4.1 shows a block diagram of the new version of the software. In order to speed up the routing, we have introduced a queue between *Task2* and the *router*. As a result, the *flooding* of the network is more realistic. We have also removed the soft channels between the enqueuing tasks and the *output queue managers*, allowing the queues to be filled up more than before. This removes a serialization mechanism. In the earlier version, whenever a packet would be enqueued, the output manager would be awakened by the enqueuing routine and the enqueuing and dequeuing would operate in a synchronized pattern. The removal of the soft channels decouples this procedure and increases the parallelism.

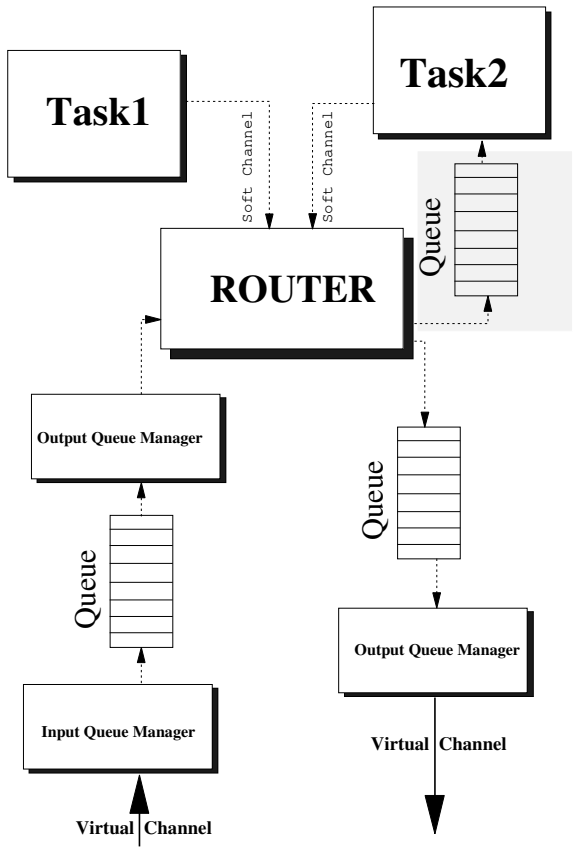

Figure 4.1: Modified block diagram of software controlling a logical node. (Highlighted section shows new addition.)

4.3 Performance Evaluation of the Inebriated Router

In this section we analyze the different performance measures mentioned before with respect to probability of the router choosing *Group I*. We first show the trends of the *deflection factor* and the *mean delay per hop* with probability p[2] and then combine the two statistics to show the source of the performance gain of the Inebriated Router algorithm.

4.3.1 Deflection Factor

Figure 4.2 and figure 4.3 show that the deflection factor has a linear trend that decreases with probability. This is logical because the higher the probability of a packet being sent out on a preferred link, the lower the probability of that packet being deflected from its shortest path. With $p = 100\%$, the packets never get deflected from their shortest path, and the deflection factor equals 1. This case is equivalent to applying the Shortest Path algorithm.

It is necessary to mention that we have studied the behavior of the Inebriated Router algorithm only in the range 80% to 100%, because our measurements show that this was a range in which "interesting" things were happening. Thus, the linear trend that we observe has been verified only in the 80% to 100% range and we expect the curve to take a nonlinear form at lower probability. For $p = 0\%$, the packets will never reach their destination and the deflection factor should go to infinity, giving the curve a parabolic shape.

Figure 4.2 and figure 4.3 also show that the deflection factor is independent of

[2]This is the probability of a preferred link (*Group I* link) being selected, as opposed to *Hot Probability*, which is the probability of a packet's destination becoming a *hot* node.

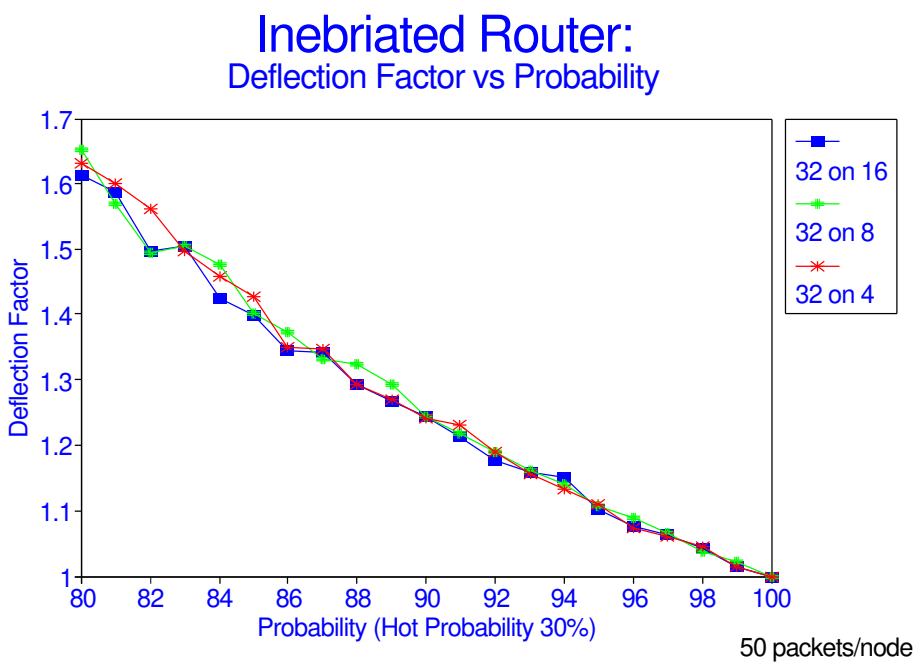

Figure 4.2: Inebriated Router: Variation of deflection factor with probability and effects of physical to logical mapping ($LMAX = 32$, $PMAX = 4, 8, 16$; Probability of a packet going to a hot spot = 30%; Number of Hot Spots = 1).

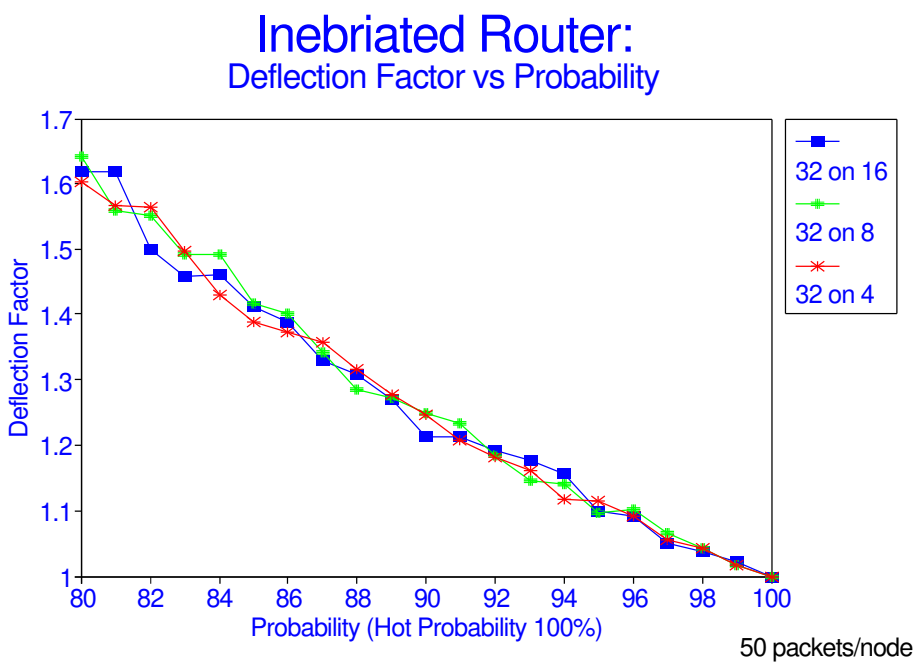

Figure 4.3: Inebriated Router: Variation of deflection factor with probability and effects of physical to logical mapping ($LMAX = 32$, $PMAX = 4, 8, 16$; Probability of a packet going to a hot spot = 100%; Number of Hot Spots = 1).

$PMAX$ (the number of physical nodes), but depends on $LMAX$ (the number of logical nodes). This is evident from the fact that the curves for 32 on 16, 32 on 8 and 32 on 4 almost overlap. The graphs show the deflection factor versus probability curves for $LMAX = 32$ and $PMAX = 4, 8$ and 16. For the case for which we have presented data ($LMAX = 32$), the following equation is obtained through linear regression:

$$\text{deflection factor} = 4 - 0.03 * p \tag{4.1}$$

For each 1% of increased probability of deflection, the factor increases by 0.03. However, as explained before, this equation holds only in the $p = 80\%$ to 100% range.

Effects of mapping

Figure 4.4 shows that the slopes of the deflection factor curves become steeper with increasing network size, although they all converge at $p = 100\%$, where the deflection factor equals 1. The steeper slope of the curves can be explained by the fact that larger networks offer a larger number of nodes to which a packet may travel. In other words, there is a larger number of alternative paths from source to destination. Consequently, the deflection factor increases at a more rapid rate as the probability of deflection increases. For instance, in figure 4.4, the slope of the curve for the 4 on 4 configuration is flatter than the 8 on 8 and the 16 on 16 configurations. In the 4 on 4 case, a packet can have a maximum deflection factor of 2, while in the 16 on 16 case, a packet can be deflected much more (maximum deflection factor = 4).

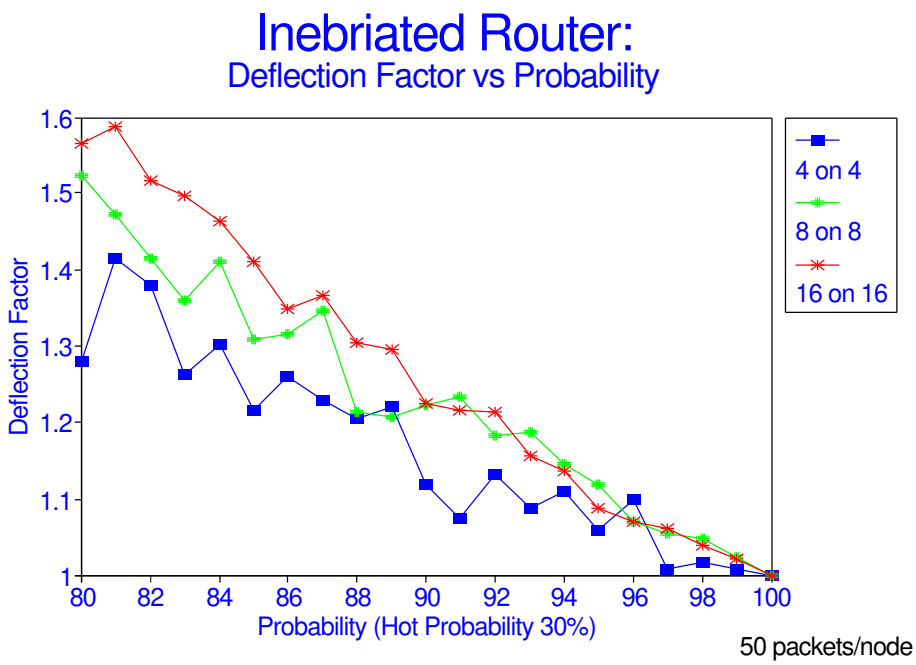

Figure 4.4: Inebriated Router: Variation of deflection factor with probability and effects of network size.

Finally, the Inebriated Router does not perform well if we evaluate the algorithm only in terms of deflection factor. However, as we will show in the following sections, its performance in terms of *mean delay per hop* sets off this comparative disadvantage.

4.3.2 Mean Delay per Hop

Thus far, we have looked only at the probability of a packet being sent to a Group I link. In this section, we consider *hot probability* (the probability of a packet being sent to a hot spot). As hot probability increases, the time a packet spends waiting in queues increases proportionally. This trend can be seen in figure 4.5. The *mean delay per hop* (defined by Equation 1.5) is about 100 milliseconds when the hot probability is 30%. In other words, when 30% of the traffic is going to a hot spot, the mean delay per hop is in the 100 milliseconds range. It goes up to over 500 milliseconds for 100% hot probability (when all the packets are going to the hot node).

From figure 4.5 we also see that the mean delay per hop versus probability curves get steeper with increasing hot probability. This can also be explained by the longer queueing delays faced by packets when there is less background traffic (traffic not directed towards a hot node), and more traffic directed towards the hot spot. When background traffic is high, (i.e. hot probability is low), the packets exhibit much more random movement and the extra reduction in delay per hop due to further increase in randomness (with lower p) is not present. As we have shown in the previous section, deflection factor increases with increasing probability of packets being misdirected. This also implies that the number of hops that the packet travels

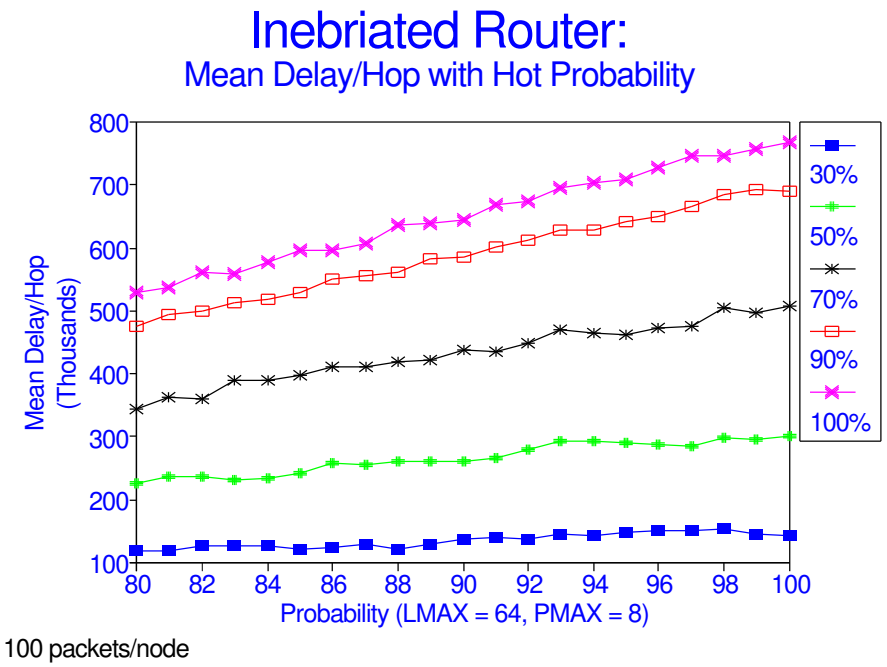

Figure 4.5: Inebriated Router: Variation of Mean Delay per hop with probability and effects of hot probability.

increases. As shown in equation 1.5, mean delay per hop, contains *total number of hops* in the denominator. Consequently, as the denominator increases, mean delay per hop decreases. **Mean delay per hop is thus a source of performance gain for the Inebriated Router algorithm.**

Effects of mapping

In figure 4.6 we show the effect of physical to logical scaling on mean delay per hop. Not surprisingly, mean delay per hop increases with increasing $LMAX$ (the number of logical nodes in the network). That is, with more logical nodes mapped on to a constant number of physical nodes, the mean delay per hop increases. When several logical nodes are mapped to a single transputer, all the logical nodes share one processor. Hence, each logical node has to wait longer than it would have had to wait if fewer logical nodes were mapped to a single physical node. This results in longer delays during the routing computation.

The larger per packet delays due to the larger network sizes are balanced by the larger number of hops that each packet has to go through. However, the processor sharing time is an overhead. That is why in figure 4.6 the mean delay per hop versus probability curves increase proportionally with $LMAX$.

4.3.3 Effectiveness of the Inebriated Router Algorithm

To measure the effectiveness of the Inebriated Router algorithm, we combine the effects of the mean delay per hop and the deflection factor to compute the effective delay. It is the product of mean delay per hop and deflection factor and represents how long a packet travels, on the average, to go from one node to its neighbor.

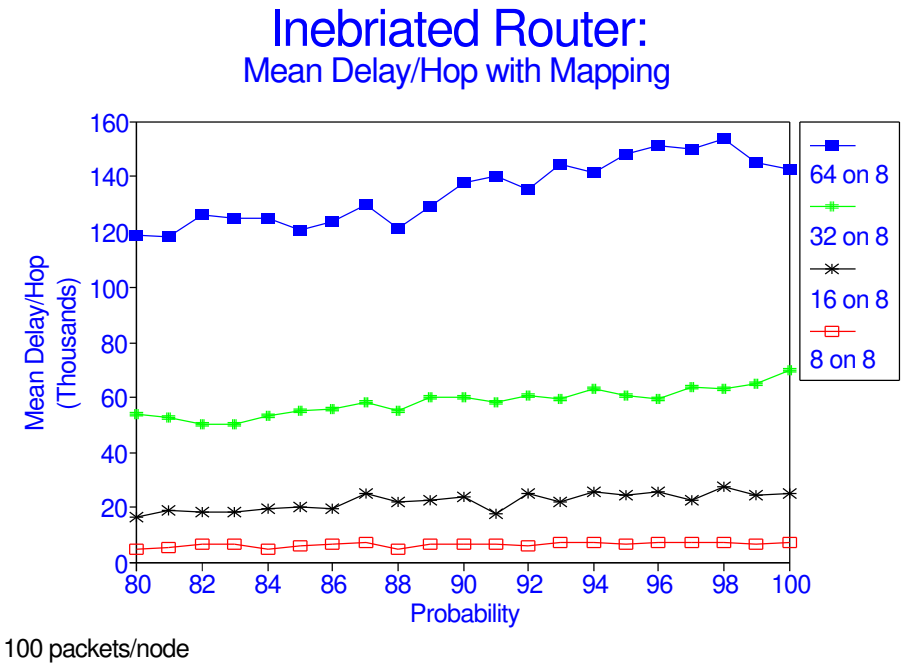

Figure 4.6: Inebriated Router: Variation of Mean Delay per hop with probability and effects of scaling.

In figure 4.7 we show all three curves together. As the graph demonstrates, the effective delay hits a minimum at a probability lower than 100%. This implies that the Inebriated Router algorithm, with a probability lower than 100%, performs better than Shortest Path.

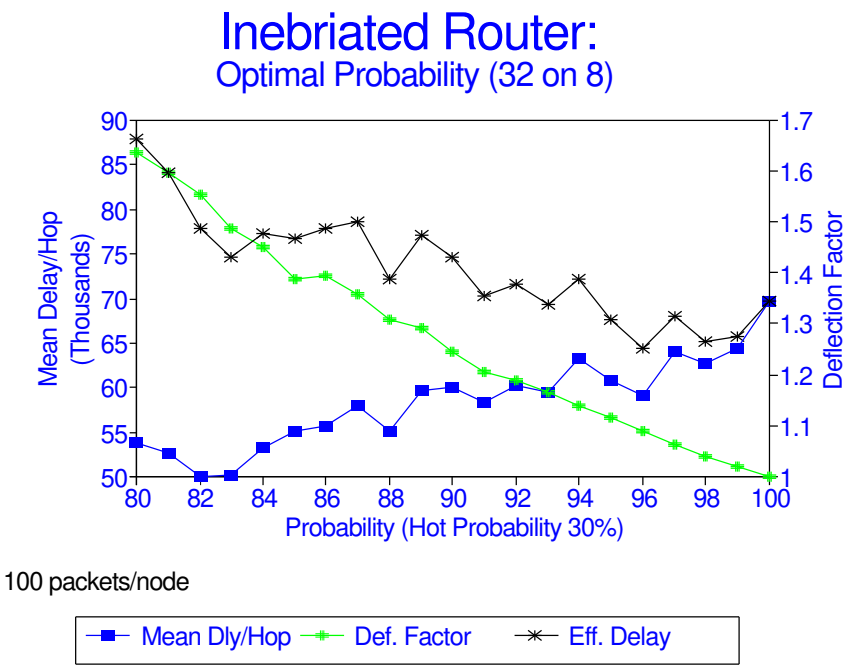

Figure 4.7: Inebriated Router: Deflection Factor as a function of probability, demonstrating that an optimal probability exists.

In figure 4.8 we show the effective delay curve separately, along with the regression curve through the observed data points. The "bow" that is shown by the product curve in figure 4.7 is confirmed by the fifth order regression curve in figure 4.8 and we notice a minimum at $p = 96\%$.

The data for figures 4.7 and 4.8 are from experiments performed with a configuration of $LMAX = 32$ and $PMAX = 8$. The hot probability in each case is 30%

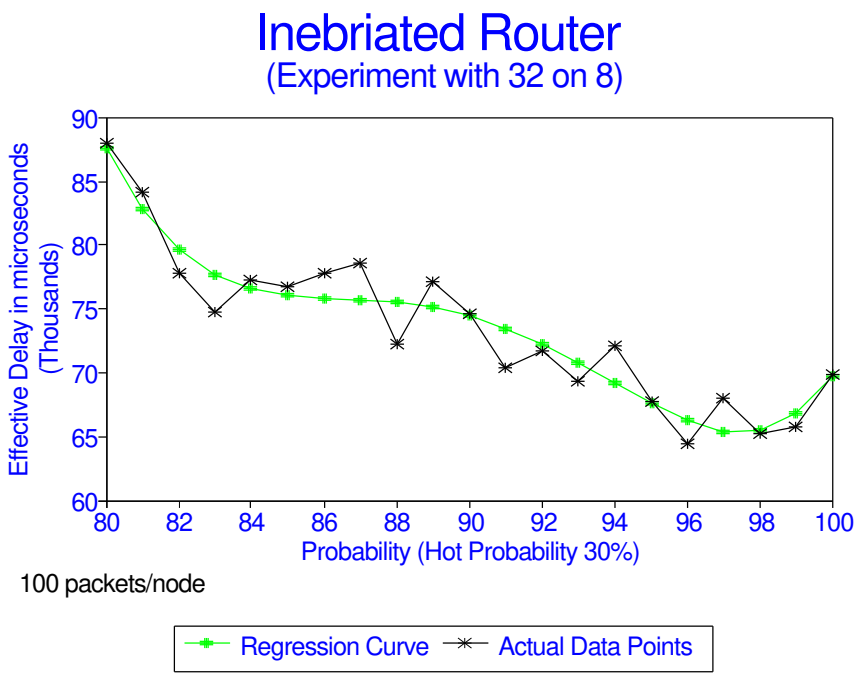

Figure 4.8: Inebriated Router: Optimal Probability based on Effective Delay.

and, as a result, the network experiences substantial background traffic. We have observed that Inebriated Router performs better (shows the presence of a minimum) with high background traffic. If the network has packets traveling only towards the hot spot (hot probability equals 100%), then the Inebriated Router algorithm does not perform as well as the Shortest Path algorithm. All the packets *have* to reach the same destination node and the random scattering does not help. The time spent in waiting in the queues is compulsory and the random deflection does not improve on Shortest Path routing.

However, if there is one (or more) hot spot(s), along with substantial background traffic, then the packets that are directed towards the hot spot(s) have little to gain from being deflected randomly, but the background traffic benefits from the advantage of being able to take an alternate route to its destination. This way, the "cool spot" traffic often takes a longer route (i.e. more hops), but this loss is offset by the gain from not having to wait in large queues.

Figure 4.9 shows the variation of *throughput* with probability for the same configuration and parameters ($LMAX = 32$, $PMAX = 8$, Hot probability = 30%). The gain from minimizing the effective delay is reflected in the measure of the *throughput*. For $p = 100\%$, which is equivalent to the Shortest Path routing, the throughput is lower than for $p = 95\%$, 96%, 97%, 98% and 99%, with a maximum relative improvement of 2.5%.

The mean delay curve is shown in figure 4.10 and has a similar shape (including the minimum) as the effective delay curve (figure 4.8). We have used effective delay as our main measurement statistic because it is independent of the number of hops that a packet has to travel. For instance, say the network traffic consists of 5 packets.

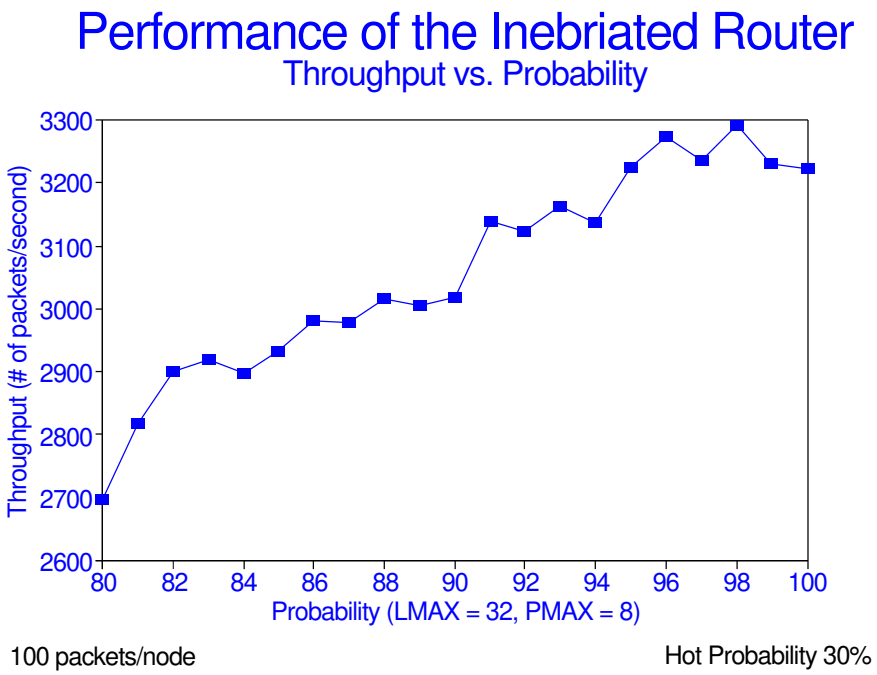

Figure 4.9: Inebriated Router: Optimal Probability based on Throughput.

Each packet is directed to a destination 3 hops away. The mean delay would then be

$$3 * delay/hop$$

If, however, 2 of the 5 packets are directed to a destination only 1 hop away, then the mean delay would be

$$(3 * 3 * delay/hop + 2 * 1 * delay/hop)/5$$

$$= (11 * delay/hop)/5$$

$$= 2.2 * delay/hop$$

This discrepancy comes from the fact that the destinations of the packets are randomly generated and therefore, vary from one experiment to another.

Effective delay gets rid of this discrepancy by individually computing the mean deflection factor and the mean delay per hop and then multiplying the two statistics. The deflection factor is independent of the destination of the packets and so is the delay per hop.

Effective Delay is defined as the the product of mean delay per hop and deflection factor.

4.3.4 Queue Size and Node Population

The Inebriated Router algorithm shows a positive linear trend in both *node population* and *queue size* when plotted against probability. The positive slope is due to the higher scattering of packets when probability of taking a preferred link is low. Figures 4.11 and 4.12 show this trend in a configuration of $LMAX = 64$ and $PMAX = 8$.

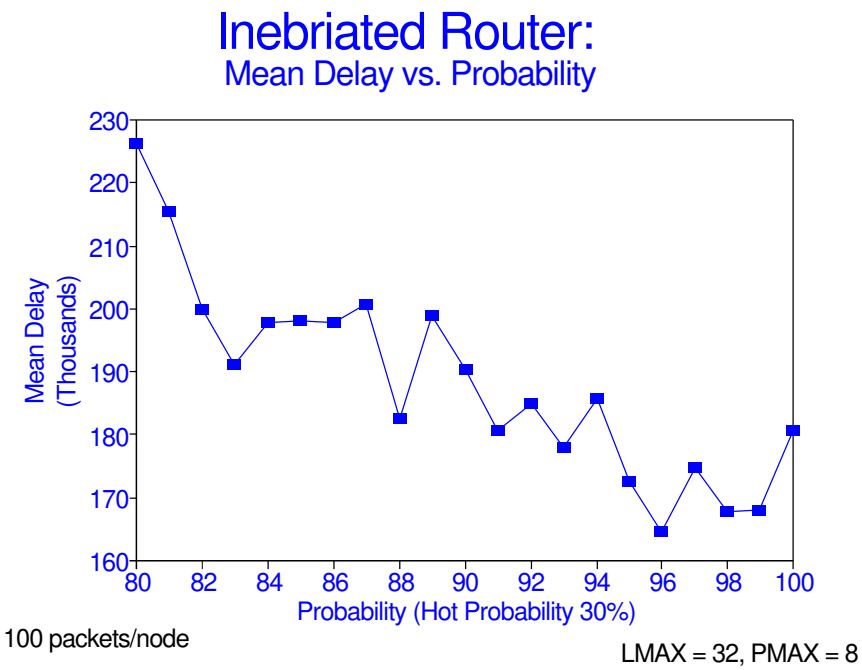

Figure 4.10: Inebriated Router: Optimal Probability based on Mean Delay.

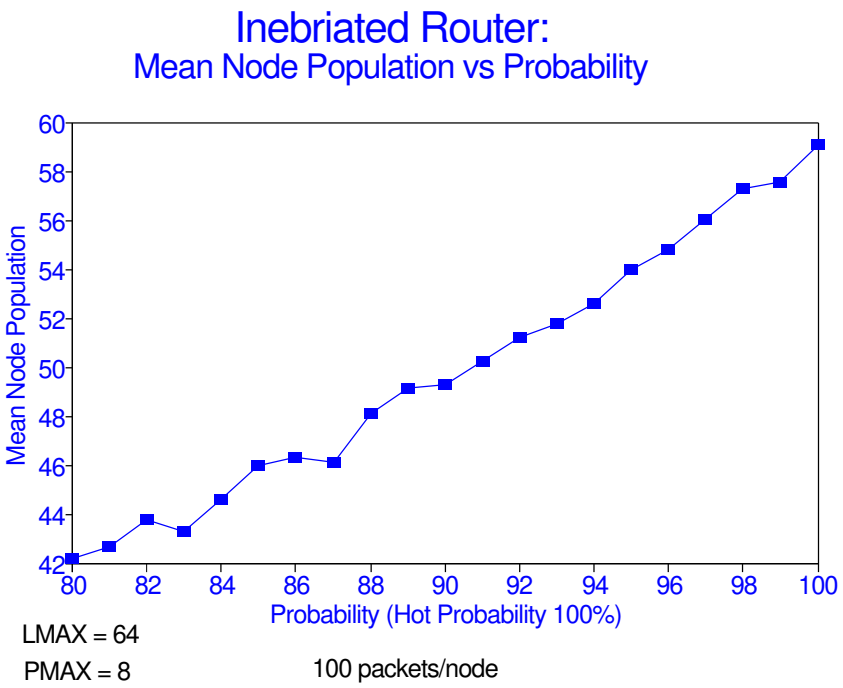

Figure 4.11: Inebriated Router: Variation of Node Population.

As the probability of packets suffering deflection in the "wrong" direction increases, the congestion *at* and *near* the "hot" nodes decreases. Thus, both node population and queue sizes decrease as probability of taking a preferred route decreases.

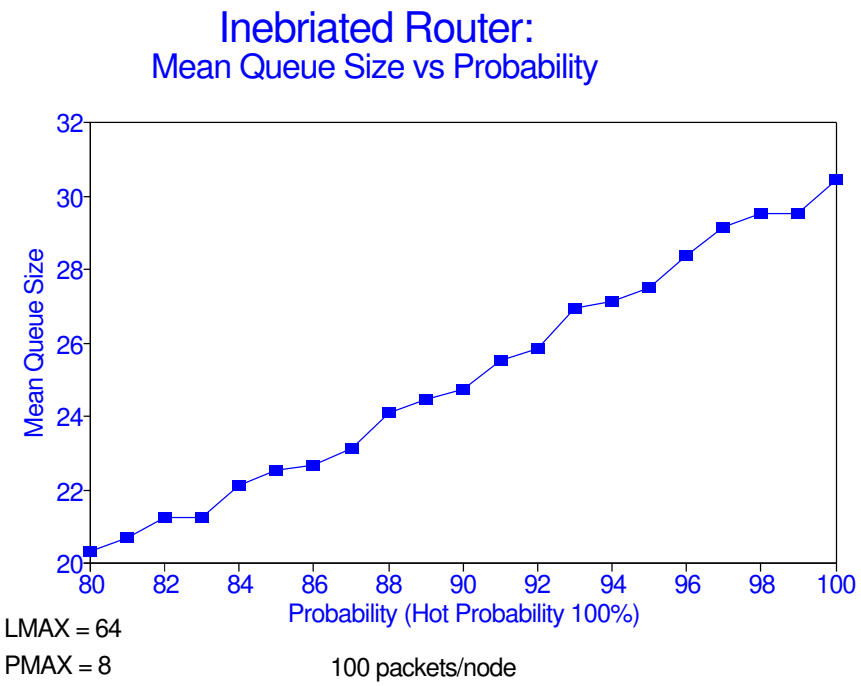

Figure 4.12: Inebriated Router: Variation of Queue Size.

4.4 Comparison of Valiant, Greenberg, Shortest Path and Inebriated Router

In Chapter 2, figure 2.12 shows that the Inebriated Router performs better than Valiant and Greenberg in terms of *mean delay* when the number of hot spots is lower. In that chapter, we did not consider any background traffic and hence could not show

that the Inebriated Router can also perform better than the Shortest Path algorithm. In this section, we will show that the Inebriated Router algorithm performs better than the other three algorithms when background traffic is introduced.

4.4.1 Mean Delay

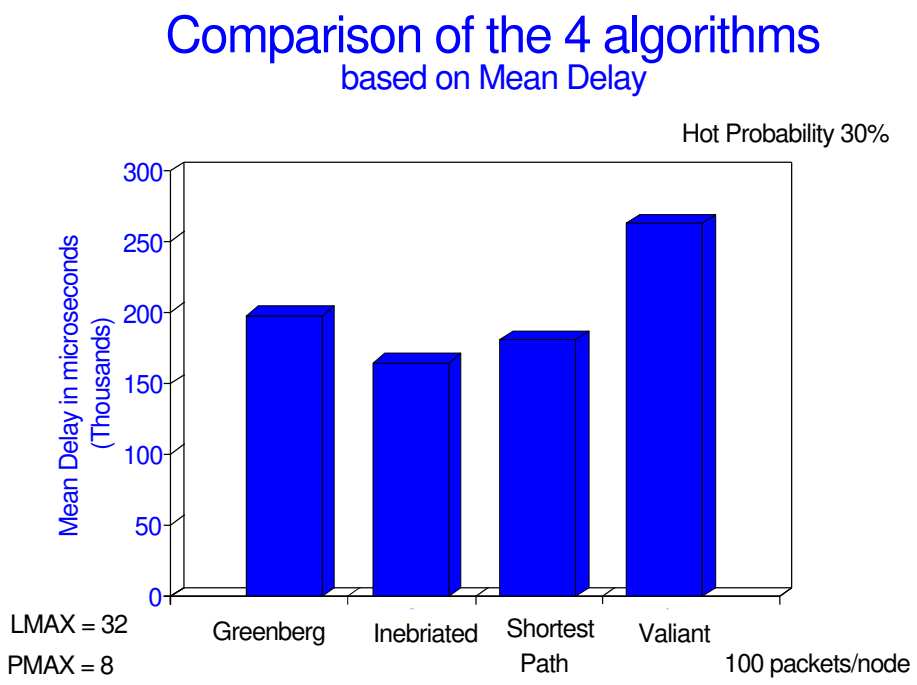

Figure 4.13: Comparison of the four algorithms in terms of Mean Delay: (Hot Probability = 30%, Background Traffic = 70%; p = 96% (for Inebriated Router), 1 Hot Spot).

Figure 4.13 reinforces what we have shown in figure 2.12. It also shows that the Inebriated Router algorithm with p = 96% has a lower mean delay than that of the Shortest Path algorithm. We have explained the reason behind the performance gain of the Inebriated Router algorithm when the network has spot background traffic

in Section 4.3.3. Valiant (as explained in Chapter 2) suffers from the expensive initial scattering phase, while Greenberg suffers because of its very low queue size threshold (1 in the experiments).

4.4.2 Throughput

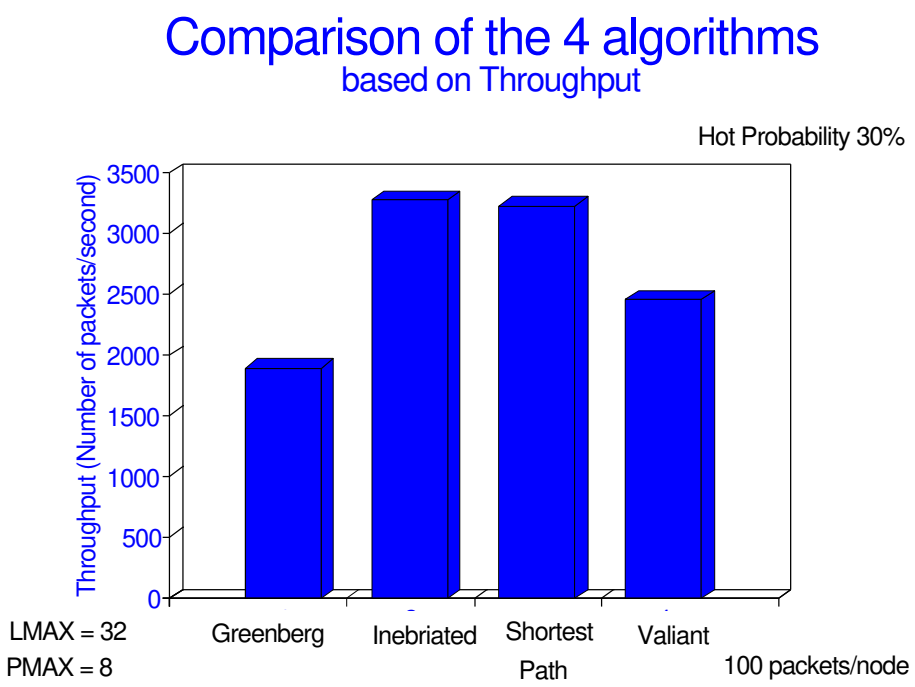

Figure 4.14: Comparison of the four algorithms in terms of Throughput: (Hot Probability = 30%, Background Traffic = 70%; p = 96% (for Inebriated Router), 1 Hot Spot).

In terms of *throughput* (figure 4.14), the Inebriated Router outperforms the other three algorithms. However, Valiant performs better than Greenberg in terms of throughput (worse in terms of mean delay) because of the very high deflection factor from which Greenberg suffers. As is clear from figures 4.13 and 4.14, neither

Valiant nor Greenberg does particularly well in terms of throughput or mean delay, in the presence of background traffic.

In this section, we have shown that the Inebriated Router algorithm performs better than the other three algorithms in terms of *mean delay, effective delay,* and *throughput*, in presence of background traffic. We have also explained the sources of performance gain in each of these cases. In the next section we will compare the Inebriated Router algorithm with the other three algorithms in terms of *node population* and *queue size*.

4.4.3 Node Population and Queue Size

Figures 4.15 and 4.16 show that Greenberg and Valiant outperform Inebriated Router and Shortest Path in terms of Queue Size and Node Population. This is an important conclusion and has implications for machines where memory is an issue. As we stated in the conclusion of Chapter 2, memory is a limitation in case of transputers. Therefore, the ability of mapping very high dimensional cubes on to smaller physical configurations is directly influenced by the ability of restricting queue sizes. The trade-off is that both Valiant and Greenberg have higher latency and lower throughput than Inebriated Router and Shortest Path.

4.5 Conclusions

In this chapter we have analyzed the performance of the Inebriated Router algorithm in greater detail. We have introduced background traffic into the experiment and analyzed the effects of mixing background traffic and hot spots. The following are some of the results that we have found:

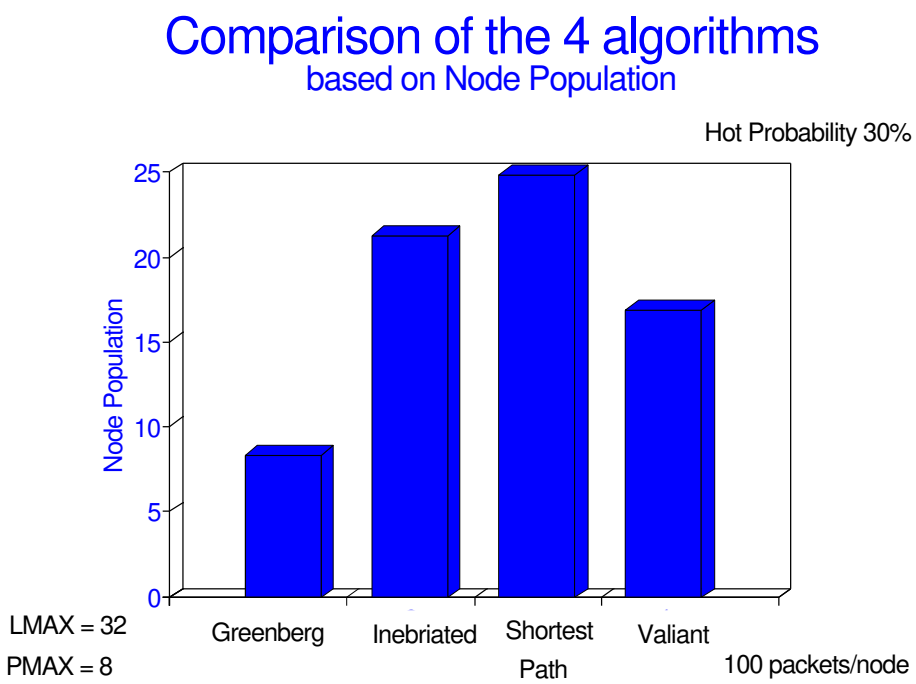

Figure 4.15: Comparison of the four algorithms in terms of Node Population: (Hot Probability = 30%, Background Traffic = 70%; $p = 96\%$ (for Inebriated Router), 1 Hot Spot).

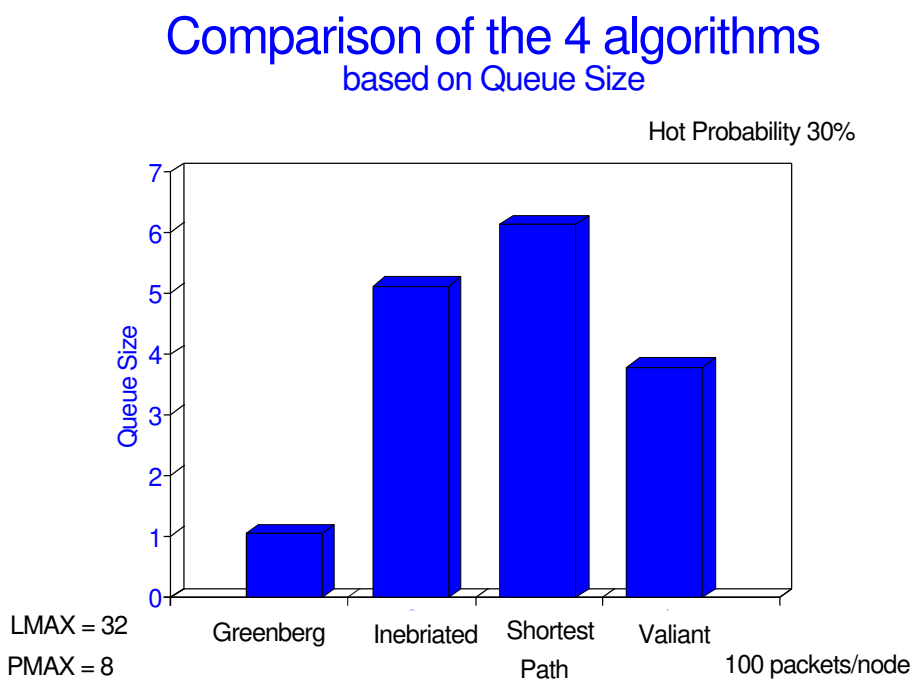

Figure 4.16: Comparison of the four algorithms in terms of Queue Size: (Hot Probability = 30%, Background Traffic = 70%; $p = 96\%$ (for Inebriated Router), 1 Hot Spot).

- The Inebriated Router algorithm benefits from high probabilities of deflection by reducing average delay per hop.
- The gain from reduced mean delay per hop is counter-balanced by the high deflection factor, enough to prevent the Inebriated Router algorithm from outperforming Shortest Path routing.
- However, when background traffic is introduced, the Inebriated Router algorithm does show better performance than Shortest Path.
- Even with background traffic, the Inebriated Router algorithm does not perform very well in terms of queue sizes and node populations.

The above results lead us to conclude that the Inebriated Router algorithm has the potential to perform better than Shortest Path routing, as well as the other algorithms—namely, Valiant and Greenberg, in situations where the transfer of information creates some nodes that are hotter than the others. However, this performance gain comes at the cost of high queue sizes and node populations, for which the other algorithms (particularly Greenberg) offer better performances.

CHAPTER 5

CONCLUSIONS

In this thesis, we studied inter-processor communication in a message passing environment. We proposed a new randomized routing algorithm called the "Inebriated Router" and studied its performance in a network of transputer hypercubes. We compared the performance of the Inebriated Router in terms of *mean delay, mean queue size, mean node population, throughput* and *deflection from the shortest path* with the previously published algorithms of Valiant and Greenberg, and with the traditional Shortest Path algorithm.

We also studied the effects of mapping hypercubes of larger dimensions to those of smaller dimensions on the statistics mentioned above.

Our experiments show the following main results:

- When larger hypercubes are mapped on to smaller physical hypercubes, the performance statistics vary differently. The mean packet delay shows a linear trend, with a slope depending on the degree of multitasking required of each physical node and the reduced communication time when on-chip communication takes place. The deflection factor increases with increasing $LMAX$ (the number of logical nodes), because of the presence of larger number of alternative routes. Interestingly, the queue size and node population, however, are

not affected by logical to physical mapping. These results imply that studies of high dimensional hypercubes can be done on smaller networks and the relevant statistics for the high dimensional configurations can be inferred from the observed statistics.

- The Inebriated Router algorithm outperforms Valiant, Greenberg and Shortest Path in terms of mean packet delay, throughput and effective delay in cases where the network exhibit a mixture of background and hot spot traffic. It outperforms Greenberg and Valiant even with no background traffic and the traffic is completely hot spot bound. However, in such situations, randomization does not provide any advantage and the Shortest Path algorithm provides optimum performance.

- The Inebriated Router exhibits a minimum in cases with high background traffic when *effective delay* and *mean delay* are plotted against probability. The optimal probability is observed to be in the high 90% range and produces better performance than the other algorithms. However, the optimal probability is not expected to be the same for all configurations, traffic conditions and number of hot spots. It is likely to vary with $LMAX$, $PMAX$, Hot Probability, and number of hot spots.

In the course of our research we have come across several areas that call for further study.

- The current method of selecting the optimal probability for the Inebriated Router is crude and time consuming and requires the user to measure the statistics for all the different configurations, hot probabilities, and numbers of

hot spots. A more efficient method would require finding a way to automate this process. The ideal solution would be to have a mathematical model to predict the optimal probability based on all these variables.

- Greenberg's algorithm is adaptive in the sense that it adjusts to the condition of the traffic in the network. The Inebriated Router algorithm, we expect, could improve its performance if it could modify the probability p according to the network traffic.

- The trends observed in several statistics, for example, the deflection factor, indicate possibilities of adapting to predictive mathematical models. We show an equation for a particular configuration in Equation 4.1. However, the equation applies only to a specific case and it is possible to predict general mathematical models.

BIBLIOGRAPHY

[ALMN91] W. A. Aiello, F. T. Leighton, B. M. Maggs, and M. Newman. Fast algorithms for bit-serial routing on a hypercube. *Mathematical Systems Theory*, 24(4):253–271, 1991.

[BGP84] F. Baccelli, E. Gelenbe, and B. Plateau. An end-to-end approach to the resequencing problem. *J. ACM*, 31(3):474–485, July 1984.

[Dal90] W. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[GH92] A. Greenberg and B. Hajek. Deflection routing in hypercube networks. *IEEE Trans. Comm.*, 40(6):1070–1081, June 1992.

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

[MC87] D. Mitra and R. A. Cieslak. Randomized parallel communications on an extension of the omega-network. *J. ACM*, 34(4):802–824, October 1987.

[MT90] D. May and P. Thompson. Transputers and routers: Components for concurrent machines. *Proceedings of the 3rd Transputer/Occam International Conference, Tokyo IOS Press*, May 1990.

[PU90] D. Peleg and E. Upfal. A time-randomness trade-off for oblivious routing. *Siam J. Computing.*, 19(2):256–266, April 1990.

[Tan81] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

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

APPENDIX A

PROGRAM LISTINGS

The following is the complete C (LSC) code for the software used in the experiments. The code for Inebriated Router is listed completely and the code for the other three algorithms is almost identical. For Valiant and Greenberg, we listed only the functions and modules that are different. A sample *network information file* and a sample *make* file are also provided.

```
/*****************************************************************
  inebr.c                                                  1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

 This program implements a logical Hypercube of transputers using
 virtual channels for inter-node communication. Each logical node
 generates a random destination. A packet containing the source,
 destination, time received and time sent is sent out by each node
 in a direction determined by XORing the logical node number with
 the destination node number.

 The Inebriated Router Algorithm:

 The algorithm used by the router is: All links coming out of a
 node are divided into 2 groups. Group I contains the links which
 offer a shortest path to the destination of the packet. Group II
 contains links that do not. The relay node introduces randomness
 in the routing in two ways. Each relay node conducts a Bernoulli
 trial to select one of two groups. With a probability p used
 homogeneously throughout the network, the node picks Group I.
 With a much lower probability (1-p), Group II is chosen. Once a
 group is selected, one of its links is chosen at random, and the
 packet is sent out on that link.

 This program implements a logical Hypercube of transputers using
 virtual channels for inter-node communication. Each logical node
 generates a random destination. A packet containing the source,
 destination, time received and time sent is  and sent out by each
 node in a direction determined by XORing the logical node number
 with the destination node number.
*****************************************************************/

#include "packet.h"
#include "bufqueue.h"
#include "globals.h"
#include "math.h"
#include "sync.h"
#include "print.h"
#include "params.h"
```

```
/*********************************************************************/
/* Definitions */
/*********************************************************************/
#define HYPER1_C
#define ONE           1
#define TWO           2

#define SIZEINT       sizeof(int)

#define WS_SIZE1      1024
#define WS_SIZE2      2048
#define WS_SIZE3      3072
#define WS_SIZE4      4096

#define TIMEOUT       2000000/64/* for shutdown */
#define BARRIER_4     6000000   /* so that each processor starts
                                   sending packets at the same
                                   instant in time */
/*********************************************************************/
/* Globals */
/*********************************************************************/
int  NO_RANDOM_PACKETS = 1;
int  PACKET_PRINT_MODE = 0;
char filename[200];
int  PROBABILITY = 100;
int  Hotdegree   = LMAX;
int  dest[LMAX];
int  HEAP_SHOW  = 0;
int  HEAP_SIZE  = 5000;
int  HotProba   = 50;
/*********************************************************************/
/* Prototypes */
/*********************************************************************/
int  GenerateRandomDest(int source);
int  PickRandomDim(int lnn, PACKET_TYPE packet);
int  FlipCoin();
void Task1(Process *p, int lnn, Channel *T1Rsoft);
void Task2(Process *p, int lnn, Channel *T2Rsoft,
           Channel *R2Tsoft, FILE *fp);
void LogicalMain(Process *P, int lnn, FILE *fp);
void Route(Process *P, int lnn, QUEUE_TYPE *Buffer[],
           Channel *T1Rsoft, Channel *T2Rsoft,
           Channel *R2Tsoft, Channel *SoftcArray[]);
```

```
void AbortIf(int c, char *s);
void DisplayHeapSize(void);
void ShowHeapSize(FILE *fp,char *s, int node);
int  TimeHigh();
PACKET_PTR StorePacket(PACKET_TYPE packet, PACKET_PTR Packet_Lst);
void PrintParameters(FILE *fp);
void GetDefaults(void);
void GetParameters(int argc, char *argv[]);
void DisplaySyntax();
void UnStuck(Process *P, FILE *fp);
int  Max(int old, int new);
int  Min(int old, int new);
int  ShortestPath(int source, int dest);
void SwitchSS16();

/*********************************************************************/
main(int argc, char *argv[])
{
     Process *LogicalProcess[LMAX/PMAX +1];
     int i,j;
     FILE *fp = NULL;

     if (argc<=1)
     {
          if (_node_number == 1)
                DisplaySyntax();
          exit(0);
     }
     GetDefaults();
     GetParameters(argc, argv);

     if (( fp = fopen(filename, "at") ) == NULL)
          printf("Couldn't open %s for writing\n", filename);

     DisplayHeapSize();
     if (_node_number == 1)
     {
        PrintParameters(fp);
        SwitchSS16();
     }

     /*--- create each logical node ---*/
     for (i = 0; i < LMAX/PMAX; i++)
```

```
     {
          LogicalProcess[i] = ProcAlloc(LogicalMain, WS_SIZE2, 2,
                                _logical_node_number(i), fp);
          AbortIf(LogicalProcess[i] == NULL, "Error in allocating
                                              memory\n");
     }
     LogicalProcess[LMAX/PMAX] = NULL;

     /*--- spawn them ---*/
     ProcParList(LogicalProcess);

     /*--- reclaim memory ---*/
     for (i = 0; i < LMAX/PMAX; i++)
         ProcFree(LogicalProcess[i]);

     fclose(fp);

     exit(0);
}
/*****************************************************************/
/* The DIM+1 st manager/queue is between Route and Task2         */
/*****************************************************************/
void LogicalMain(Process *P, int lnn, FILE *fp)
{
     Process    *Task1P, *Task2P, *RouteP;
     Process    *ParallelINOutputTasks[DIM],
                *ParallelOUTOutputTasks[DIM+1];
     Process    *ParallelInputTasks[DIM];
     QUEUE_TYPE *INBuffer[DIM], *OUTBuffer[DIM+1];
     Process    *UnStuckP;

     int i, j;
     Channel *SoftcArray[DIM+3];
     Channel *T2Rsoft, *T1Rsoft, *R2Tsoft;

     /* Synchronize the HP timer which will keep the time */
     Sync(lnn);

     /*--- now all nodes run an UnStuck routine to make program
           break in case of failure ---*/
     UnStuckP = ProcAlloc(UnStuck,WS_SIZE1,1,fp);
     if (UnStuckP==NULL)
         exit(3);
```

```
ProcRun(UnStuckP);

for (i=0; i < DIM; i++)
{
     SoftcArray[i] = ChanAlloc();
     AbortIf( (SoftcArray[i] == NULL),
                "Out of soft dyn memory\n");
}
T1Rsoft           = ChanAlloc();
T2Rsoft           = ChanAlloc();
R2Tsoft           = ChanAlloc();
SoftcArray[DIM]   = T1Rsoft;
SoftcArray[DIM+1] = T2Rsoft;


AbortIf( ( (SoftcArray[DIM] == NULL)
           || (SoftcArray[DIM+1] == NULL)
           || (R2Tsoft == NULL) ), "Out of TR dyn memory\n");
SoftcArray[DIM+2] = NULL;

/*--- Create buffers with associated links, and start the
      high-priority managers ---*/
for (i=0; i < DIM; i++)
{
     j = VChanNumber(lnn,i);
     OUTBuffer[i]     = InitQueue(MAX_BUFFER_SIZE,j, VChan(j));
     INBuffer[i]      = InitQueue(MAX_BUFFER_SIZE,j,
     SoftcArray[i]);
     ParallelOUTOutputTasks[i]= ProcAlloc(QueueOutManager,
     WS_SIZE1, 1, OUTBuffer[i]);
     ParallelINOutputTasks[i]= ProcAlloc(QueueOutManager,
     WS_SIZE1, 1, INBuffer[i]);
     ParallelInputTasks[i]= ProcAlloc(QueueInManager,
     WS_SIZE1, 3, VChan(j), INBuffer[i], INBuffer);
     AbortIf(( (ParallelINOutputTasks[i] == NULL) ||
               (ParallelOUTOutputTasks[i] == NULL) ||
               (ParallelInputTasks[i] == NULL) ),
                "Out of manager dyn memory\n");
     ProcRunHigh(ParallelInputTasks[i]);
     ProcRunHigh(ParallelOUTOutputTasks[i]);
     ProcRunHigh(ParallelINOutputTasks[i]);
}
/* initialize buffer between Route and Task2  */
```

```
    OUTBuffer[DIM] = InitQueue(MAX_BUFFER_SIZE, 0, R2Tsoft);
    ParallelOUTOutputTasks[DIM]= ProcAlloc(QueueOutManager,
    WS_SIZE1, 1, OUTBuffer[DIM]);
    ProcRunHigh(ParallelOUTOutputTasks[DIM]);

    /*--- initialize the two main tasks ---*/
    Task1P  = ProcAlloc(Task1, WS_SIZE1, 2, lnn, T1Rsoft);
    Task2P  = ProcAlloc(Task2, WS_SIZE4, 4, lnn, T2Rsoft,
              R2Tsoft, fp);
    RouteP  = ProcAlloc(Route, WS_SIZE1, 6, lnn, OUTBuffer,
              T1Rsoft, T2Rsoft, R2Tsoft/*Not used*/, SoftcArray);

    AbortIf(( (Task1P == NULL) || (Task2P == NULL) ||
              (RouteP == NULL) ),
              "Out of TTR dyn memory\n");

    ProcRunHigh(RouteP);
    ProcPar(Task1P,Task2P,0);

    ProcFree(Task1P);
    ProcFree(Task2P);
}
/*************************************************************/
/* UNSTUCK                                                   */
/* if the program gets stuck this routine will automatic-    */
/* ally exit after 2 minutes                                 */
/*************************************************************/
void UnStuck(Process *P, FILE *fp)
{
        ProcToHigh();
        ProcAfter(Time()+120000000); /* wait 2 minutes */
        fclose(fp);
        exit(3);
}
/*************************************************************/
/* TASK1:  Sends a packet to a random destination            */
/*************************************************************/
void Task1(Process *p, int lnn, Channel *T1Rsoft)
{
   PACKET_TYPE  packet;
   int          i, j;
   char         temp;
```

```
   /*--- send a packet(s) to a random destination ---*/
   packet.source   = lnn;
   packet.type     = INFO; /*--- mark packet as info packet ---*/
   packet.NoHops   = -1;
   for (j = 0; j < DIMSTAT; j++)
   {
        packet.QueueSize[j]      = 0;
        packet.NodePopulation[j] = 0;
        packet.NodeID[j]         = 0;
   }
   ProcToHigh();
   ProcAfter(BARRIER_4);
   ProcToLow();
   for (i=0; i<NO_RANDOM_PACKETS; i++)
   {
          packet.dest     = GenerateRandomDest(packet.source);

          if (FlipCoin() )
               ProcReschedule();

          packet.TimeSent = TimeHigh();
          ChanOut(T1Rsoft, &packet, PACKET_SIZE);
   }

   /*--- Now tell the root that we are done ---*/
   if (lnn != 0)
   {
          packet.dest       = 0;
          packet.type       = NODE_DONE;
          packet.TimeSent   = TimeHigh();
          ChanOut(T1Rsoft, &packet, PACKET_SIZE);
   }
}

/************************************************************* */
/* TASK2:                                                     */
/*   Acts as a relay for messages.  Messages intended for other */
/*   nodes are routed on the appropriate link.                */
/*   Three kinds of packets are dealt with:                   */
/*   INFO packets:  They are the packets sent to random dest-  */
/*                  inations.  If one is received and addressed */
/*                  to the node, this one prints a message indi-*/
/*                  cating that the packet was received with  */
/*                  a given delay.                            */
```

```
/*   NODE_DONE packets: These packets are sent by the nodes when*/
/*                  they are done with their computing job (for */
/*                  now that is sending 1 random packet.)       */
/*   ROOT_DONE packets: They are sent by the Logical Root when  */
/*                  it has received LMAX packets of type        */
/*                  NODE_DONE.  Upon receiving such a packet, a */
/*                  node will prepare to shut down.  It will    */
/*                  wait awhile before doing so, so as to ensure*/
/*                  that it can still relay ROOT_DONE packets   */
/*                  sent to nodes farther away.                 */
/* algorithm:                                                   */
/*     1) synchronize (done outside this code)                  */
/*     2) send data packets and measure statistics              */
/*     3) when a node is done, it sends NODE_DONE packet to root*/
/*     4) when root has gotten LMAX-1 NODE_DONE packets, it     */
/*        will                                                  */
/*           a) print its own statistics, and                   */
/*           b) send PRINT_STATS packet to Node 1.              */
/*     5) Node j will get PRINT_STATS packet, will print its own*/
/*        statistics, and relay the PRINT_STATS to Node j+1.    */
/*        At the same time, it will start its shut down mode.   */
/*     6) But if j == LMAX-1, then Node j send LAST_NODE_PRINTED*/
/*        to root, which will display "end of computation" msg  */
/************************************************************* */
void Task2(Process *p, int lnn, Channel *T2Rsoft, Channel *R2Tsoft,
FILE *fp)
{
   float  thrput = 0;
   PACKET_TYPE packet;
   int i, index, j;
   int Done      = 0;
   int NodesDone = 1; /* 1 because the root will be done first! */
   PACKET_PTR Packet_Lst = NULL;
   int max = 0;
   int min = 9000000;
   int FLAG = 0;
   float Delay = 0.0;
   int DelayCounter = 0;
   int Hop = 0;
   int Shortest = 0;
   int QS = 0;
   int Pop = 0;
   int QCnt = 0;
```

```
int PCnt = 0;

while (1)
{
    /**--If all nodes are done with printing, then start
         shutdown, else keep checking R2Tsoft--**/
    if (Done)
         index = ProcTimerAlt(Time()+TIMEOUT, R2Tsoft, 0);
    else
         index = ProcAlt(R2Tsoft, 0);

    if (index == -1)
         break;

    /**--Receive packet, put Time Stamp--**/
    ChanIn(R2Tsoft, &packet, PACKET_SIZE);

    if ((packet.type == INFO) || (packet.type == NODE_DONE))
         packet.TimeReceived = TimeHigh();
    if (packet.type != NODE_DONE)
    {
         min = Min(min, packet.TimeSent);
         max = Max(max, packet.TimeReceived);
    }
    switch (packet.type)
    {
      case INFO: /**-INFO packet meant for this node,
                       record statistics-**/
          if (PACKET_PRINT_MODE)
             Packet_Lst = StorePacket(packet, Packet_Lst);
          Delay+= 1.0*(packet.TimeReceived-packet.TimeSent);
          DelayCounter++;
          Hop+= packet.NoHops;
          Shortest+= ShortestPath(packet.source, packet.dest);
          for (i = 0; i < packet.NoHops; i++)
          {
                     if (packet.NoHops >= DIMSTAT)
                         break;
                     QS+= packet.QueueSize[i];
                     QCnt++;
                     Pop+= packet.NodePopulation[i];
                     PCnt++;
          }
```

```
        break;

    case NODE_DONE: /**-Root receiving acknowledgement
                        that the sender node is done
                        with sending packets-**/
        /* we are the logical root: Increment count */
        NodesDone++;

        /* if all the other nodes have responded, not
           including the root, tell them to start printing */
        if (NodesDone == LMAX)
        {
           /* root prints its own stats */
           SwitchSS16();
           PrintStart(lnn,Packet_Lst, fp,PACKET_PRINT_MODE);
           if (DelayCounter)
           {
               fprintf(fp,"\"Node\" %d %3f %d %d %d %d %d %d %d\n",
               lnn, Delay, DelayCounter, Hop, Shortest,
               QS, QCnt, Pop, PCnt);
           }
           /*--- then it passes the token around ---*/
           packet.type      = PRINT_STATS;
           packet.source    = lnn;
           packet.dest      = 1;
           packet.NoHops    = -1;
           packet.TimeSent  = min;
           packet.TimeReceived = max;
           ChanOut(T2Rsoft, &packet, PACKET_SIZE);
        }
        break;

    case PRINT_STATS: /**-Print the statistics -**/
        PrintStart(lnn,Packet_Lst, fp,PACKET_PRINT_MODE);
        if (DelayCounter)
        {
           fprintf(fp,"\"Node\" %d %3f %d %d %d %d %d %d %d\n",
           lnn, Delay, DelayCounter, Hop, Shortest,
           QS, QCnt, Pop, PCnt);
        }
        packet.TimeSent  = min;
        packet.TimeReceived = max;
```

```
            if (lnn < (LMAX-1) )
            {
               packet.type      = PRINT_STATS;
               packet.dest      = lnn+1;
            }
            else if (lnn==LMAX-1)
            {
               packet.type      = LAST_NODE_PRINTED;
               packet.dest      = 0;
            }
            else
            {
                printf("\n*** ERROR! *** lnn = %d\n\n",lnn);
                ProcWait(1000000);
            }
            packet.source    = lnn;
            packet.NoHops    = -1;

            ChanOut(T2Rsoft, &packet, PACKET_SIZE);
            Done = 1;
            break;

        case LAST_NODE_PRINTED: /**-Last node has finished printing
            its statistics => start shutdown -**/
            thrput=((1.0*NO_RANDOM_PACKETS*LMAX)/(max-min))*1000000;
            fprintf(fp,"-------------------------------------------
            -----\n\n");
            fprintf(fp,"  THROUGHPUT          = %10.3f\n", thrput );
            fprintf(fp,"-------------------------------------------
            -----\n\n");
            printf("END OF COMPUTATION (%d)\n",lnn);
            fprintf(fp,"END OF COMPUTATION (%d)\n",lnn);
            Done = 1;
            break;

        default:
            printf("\n\n*** ERROR *** Packet received of type %d\n\n",
                     packet.type);
            ProcWait(1000000);
            break;
       } /* end switch */
   } /* end while */
}
```

```
/*************************************************************/
/* ROUTE: The Router is implemented in this function. A simple  */
/*         XORing method is followed to determined which link a */
/*         packet should be sent out on.                        */
/*************************************************************/
void Route(Process *P, int lnn, QUEUE_TYPE *Buffer[],
           Channel *T1Rsoft, Channel *T2Rsoft, Channel *R2Tsoft,
           Channel *SoftcArray[])
{
     int i;
     PACKET_TYPE packet;
     int index = 0;
     int node = 0;

     while(1)
     {

         index = ProcAltList(SoftcArray);
         ChanIn(SoftcArray[index], &packet, PACKET_SIZE);

         if (packet.type == INFO)
         {
              node = ++packet.NoHops;
              if (node < DIMSTAT)
                   packet.NodeID[node]  = lnn;
         }
         if (packet.dest == lnn)
              EnqueuePacket(Buffer[DIM], &packet);
         else
         {
              i = PickRandomDim(lnn, packet);
              EnqueuePacket(Buffer[i], &packet);
         }
     }
}
/*************************************************************/
/* GENERATE RANDOM DEST : generates a random destination     */
/*************************************************************/
int GenerateRandomDest(int source)
{
    int i,j,p;

    p = rand() % 100;
```

```
    if (p > HotProba)
    {
        i = rand() % LMAX;
        while (i == source)
          i = rand() % LMAX;
        return i;
    }
    else
    {
        if ( (Hotdegree == 1) && (dest[0] == source) )
            return ( (source+1) % LMAX);
        i = rand() % Hotdegree;
        while ( dest[i] == source)
            i = rand() % Hotdegree;
        return dest[i];
    }
}
/*************************************************************/
/* FLIP COIN: Head or tail?                                  */
/*************************************************************/
int FlipCoin()
{
    return (rand() % 2);
}
/*************************************************************/
/*PICK RANDOM DIM: With probability p, pick a preferred link. */
/*                 With probability (1-p), misdirect packet.  */
/*************************************************************/
int PickRandomDim(int lnn, PACKET_TYPE packet )
{
    int array[DIM];
    int i, base, counter = 0;

    base = 1;
    for (i = 0; i < DIM; i++, base*=2)
    {
        if ((lnn^packet.dest) & base)
        {
            counter++;
            array[i]        = GROUP1;
        }
        else
            array[i]        = GROUP2;
```

```
    }
    if (packet.type == INFO)
    {
        i = rand() % DIM;
        if ( ((rand() % 100) < PROBABILITY) || (counter == DIM) )
        {
          while (array[i] != GROUP1)
             i = rand() % DIM;
        }
        else
        {
          while (array[i] != GROUP2)
              i = rand() % DIM;
        }
    }
    else
    {
       i = 0;
       while ( (array[i] != GROUP1) && (i < DIM) )
            i++;
       if (i > DIM)
          printf ("***ERROR***\n");
    }
    return i;
}
/*************************************************************/
/*ABORTIF: Abort if memory allocation problem occurs, with   */
/*         an error message.                                 */
/*************************************************************/
void AbortIf(int c, char *s)
{
     if (!c)
        return;
     printf("%s",s);
     exit(0);
}
/*************************************************************/
/* TIME HIGH : Record high priority time                     */
/*************************************************************/
int  TimeHigh()
{
     int temp;
     int Current_Priority;
```

```
    Current_Priority = ProcToHigh();
    temp = Time();
    if (Current_Priority == 1)
        ProcToLow();
    return temp;
}
/*************************************************************/
/*  DISPLAY HEAP SIZE: Displays heap size                    */
/*************************************************************/
void DisplayHeapSize(void)
{
    int  pt1, pt2, base = 1, Offset=0;
    char *ptr;
    if (_node_number==1) printf("heapend = %0X",_heapend);
    _heapend = (void *)0x800FFFFC;
    if (_node_number==1) printf(", reset to 0x800FFFFC\n");

    while (1)
    {
        base *= 2;
        ptr = (char *) malloc(Offset + base * sizeof(char));
        if (ptr==NULL)
        {
                base /= 2;
                if (base <= 1) break;
                Offset += base;
                base = 1;
        }
        free(ptr);
    }
    if (_node_number==1) printf("%d bytes can be
                    allocated\n",Offset+base);
}
/*************************************************************/
/*  SHOW HEAP SIZE (without resetting the heap pointer)      */
/*************************************************************/
void ShowHeapSize(FILE *fp, char *s, int node)
{
    int  pt1, pt2, base = HEAP_SIZE;
    char *ptr;

    while (1)
```

```
    {
        ptr = (char *) malloc(base * sizeof(char));
        if (ptr==NULL)
               break;
        base = base + HEAP_SIZE;
        free(ptr);
    }
    fprintf(fp,"[%d] %s %d bytes left in heap\n",
    node,s,base-HEAP_SIZE);
}
/************************************************************/
/*PRINT PARAMETERS: Prints parameters                       */
/************************************************************/
void PrintParameters(FILE *fp)
{
  fprintf(fp,"-------------------------------------------------\n");
  fprintf(fp,"  NO_RANDOM_PACKETS  = %10d\n", NO_RANDOM_PACKETS );
  fprintf(fp,"  PACKET_SIZE        = %10d\n", PACKET_SIZE       );
  fprintf(fp,"  LMAX               = %10d\n", LMAX              );
  fprintf(fp,"  PMAX               = %10d\n", PMAX              );
  fprintf(fp,"  DIM                = %10d\n", DIM               );
  fprintf(fp,"  MAX_BUFFER_SIZE    = %10d\n", MAX_BUFFER_SIZE   );
  fprintf(fp,"  TIMEOUT            = %10d\n", TIMEOUT           );
  fprintf(fp,"  BARRIER_4          = %10d\n", BARRIER_4         );
  fprintf(fp,"  PROBABILITY        = %10d\n", PROBABILITY       );
  fprintf(fp,"  HOT PROBABILITY    = %10d\n", HotProba          );
  fprintf(fp,"  HOT SPOTS          = %10d\n", Hotdegree         );
  fprintf(fp,"-------------------------------------------------\n\n");
}
/*****************************************************/
/* STORE PACKET : Stores packets in a linked list     */
/*****************************************************/
PACKET_PTR StorePacket(PACKET_TYPE packet, PACKET_PTR
                       Packet_Lst)
{
     PACKET_PTR new;

     new  = (PACKET_PTR) malloc(PACKET_NODE_SIZE);

     AbortIf(new == NULL, "Out of packet dynamic memory\n");
     memcpy((void *)(&(new->StoredPacket)), (void *)(&packet),
             PACKET_SIZE);
```

```
     new->next = Packet_Lst;
     return new;
}
/*************************************************************/
/* GET DEFAULTS:                                             */
/*************************************************************/
void GetDefaults(void)
{
     int i;

     NO_RANDOM_PACKETS = 1;
     PACKET_PRINT_MODE = 0;
     HEAP_SIZE         = 2000;
     HEAP_SHOW         = 0;
     HotProba          = 50;
     strcpy(filename, "hyper1.out");
     PROBABILITY = 98;
     Hotdegree   = LMAX;
     for ( i = 0; i < LMAX; i++)
          dest[i] = i;
}
/*************************************************************/
/* GET PARAMETERS:                                           */
/*************************************************************/
void GetParameters(int argc, char *argv[])
{
     char *p, deststring[3];
     int i;

     GetParams(argc, argv, "-#",   INTEGER, &NO_RANDOM_PACKETS);
     GetParams(argc, argv, "-p",   INTEGER, &PACKET_PRINT_MODE);
     GetParams(argc, argv, "-P",   INTEGER, &PROBABILITY);
     GetParams(argc, argv, "-B",   INTEGER, &HotProba);
     GetParams(argc, argv, "-H",   INTEGER,  &Hotdegree);
     GetParams(argc, argv, "-s",   INTEGER,  &HEAP_SIZE);
     GetParams(argc, argv, "-h",   INTEGER,  &HEAP_SHOW);
     if (Hotdegree != LMAX)
     {
          strcpy(deststring, "-1");
          for (i = 0; i < Hotdegree; i++)
          {
              deststring[1] = (char)('1'+i);
              dest[i] = -1;
```

```
                GetParams(argc, argv, deststring, INTEGER, &dest[i]);
                if ( (dest[i] == -1) && (_node_number == 1) )
                {
                   printf("Missing Hot destination\n");
                   DisplaySyntax();
                   exit(0);
                }
           }
     }
     GetParams(argc, argv, "-f",   STRING, filename);
     for (p=filename;*p !='\0';p++)
          *p = toupper(*p);
}
/*************************************************************/
/*   DISPLAY SYNTAX:                                         */
/*************************************************************/
void DisplaySyntax()
{
   printf("syntax: ld-net inebr [options]\n");
   printf("       -# N          : Number of packets per node\n");
   printf("       -f file_name  : Name of output file\n");
   printf("       -p [0/1]      : Print Packet data\n");
   printf("       -h [0/1]      : Print Heap data\n");
   printf("       -s [0/1]      : Print Heap Size\n");
   printf("       -P [<= 100]   : Probability\n");
   printf("       -H [1..8]     : Number of Hot Spots\n");
   printf("       -B [< = 100]  : Probability of going to Hot Spot\n");
   printf("       -1 -2 .. -8 n : specify hot destinations\n");
}
/*************************************************************/
/*   MAX: maximum                                            */
/*************************************************************/
int  Max(int old, int new)
{
     if (old > new)
        return old;
     else
        return new;

}
/*************************************************************/
/*   MIN: minimum                                            */
/*************************************************************/
```

```
int  Min(int old, int new)
{
     if (old < new)
        return old;
     else
        return new;
}
/*************************************************************/
/*   SHORTEST PATH: from source to destination               */
/*************************************************************/
int  ShortestPath(int source, int dest)
{
     int base = 1;
     int path = 0;
     int i;

     for (i = 0; i < DIM; i++, base*=2)
     {
         if ( (source^dest) & base)
             path++;
      }
      return path;
}
/*********************************************************************/
/*SWITCHSS16:closes the hypercube.                                   */
/*           We haven't really used it because the Virtual Channels */
/*           do the routing independent of hard links anyway.       */
/*********************************************************************/
#define assert      1
#define de_assert   0
#define req_address ((int *) 4)
#define ack_address ((int *) 0)
void SwitchSS16()
{  int *req_line = req_address ;
   int *ack_line = ack_address ;

   return;
   *req_line = assert ;
         while (!(*ack_line & 1) == de_assert) ;
         *req_line = de_assert ;
         while (!(*ack_line & 1) == assert) ;
}
```

```
#****************************************************************
# inebr.mak                                                1/28/93
#
# S. Mitra and D. Thiebaut                Dept. Computer Science
# smitra@sophia.smith.edu                         Smith College
# thiebaut@sophia.smith.edu                Northampton, MA 01063
#
# Make file for inebr program.
#****************************************************************
# ------------------------------------------------
# Implicit rules
# ------------------------------------------------
.c.pp:
      pp $< -dT800 -v

.pp.tal:
      tcx $< -p8 -v

.tal.trl:
      ttasm $< -t -v

.trl.tld:
      tlnk inebr.lnk

#     ld-net inebr

# ------------------------------------------------
# Explicit rules
# ------------------------------------------------
inebr.tld:   inebr.trl bufqueue.trl sync.trl print.trl
                                        params.trl inebr.lnk

inebr.trl:   inebr.c packet.h bufqueue.h sync.h print.h
                                        globals.h params.h

bufqueue.trl: bufqueue.c bufqueue.h  packet.h globals.h

sync.trl:     sync.c sync.h globals.h

print.trl:    print.c print.h packet.h globals.h

params.trl:   params.c params.h
```

```
# ------------------------------------------------
# Constructions:  [make -finebr lnkfile] will
#                 make the inebr.lnk file.
# ------------------------------------------------
lnkfile:
      C:\COMMAND /C ECHO FLAG                    > inebr.lnk
      C:\COMMAND /C ECHO LIST  inebr.map         >> inebr.lnk
      C:\COMMAND /C ECHO INPUT inebr bufqueue
                               sync print params >> inebr.lnk
      C:\COMMAND /C ECHO ENTRY _vcmain           >> inebr.lnk
      C:\COMMAND /C ECHO LIB   t8lib             >> inebr.lnk
      C:\COMMAND /C ECHO LOAD  0x80002000        >> inebr.lnk
      C:\COMMAND /C ECHO STACK 0x80001F00        >> inebr.lnk
# ------------------------------------------------
# Constructions: [make -finebr 8nodes] will
#                make the inebr.nif file for 8 nodes
# ------------------------------------------------
8nodes:
      MAKEHYPE -No 1 -1 inebr -nif inebr -L 8  -V 1

16nodes:
      MAKEHYPE -No 1 -1 inebr -nif inebr -L 16 -V 1

32nodes:
      MAKEHYPE -No 1 -1 inebr -nif inebr -L 32 -V 1

64nodes:
      MAKEHYPE -No 1 -1 inebr -nif inebr -L 64 -V 1
```

```
;/*****************************************************************
; inebr.nif                                                  1/28/93
;
; S. Mitra and D. Thiebaut                Dept. Computer Science
; smitra@sophia.smith.edu                          Smith College
; thiebaut@sophia.smith.edu                Northampton, MA 01063
;
; Network Information File (nif file) for inebr.c.
;*****************************************************************/
buffer_size     200;
host_server     CIO.EXE;
level_timeout   1000;
decode_timeout  20000;

vchan 50,50,ld_log,150,150;

; 50: load for virtual channel used on the node for host output
; 50: load for virtual channel used on the node for host intput
; ld_log: name of output file in which log is kept
; 150: maximum adjacent packet length (in bytes)
; 150: maximum remote packet length (in bytes)

 1,inebr, R0,     0,  5[2],  3[1],  9[0];
 2,inebr, R1, 10[3],  4[2],  6[1],      ;
 3,inebr, R2, 11[3],  1[2],  7[1],  4[0];
 4,inebr, R3,  3[3],  8[2],  2[1], 12[0];
 5,inebr, R4, 15[3],  7[2],  1[1],  6[0];
 6,inebr, R5,  5[3],  2[2],  8[1], 14[0];
 7,inebr, R6,  8[3],  3[2],  5[1], 13[0];
 8,inebr, R7, 16[3],  6[2],  4[1],  7[0];
 9,inebr, R8,  1[3], 11[2], 15[1], 10[0];
10,inebr, R9,  9[3], 14[2], 12[1],  2[0];
11,inebr, R10,12[3], 13[2],  9[1],  3[0];
12,inebr, R11, 4[3], 10[2], 16[1], 11[0];
13,inebr, R12, 7[3], 15[2], 11[1], 16[0];
14,inebr, R13, 6[3], 16[2], 10[1], 15[0];
15,inebr, R14,14[3],  9[2], 13[1],  5[0];
16,inebr, R15,13[3], 12[2], 14[1],  8[0];
1[6],1[11];
1[7],2[17];
1[8],3[28];
1[9],5[49];
1[10],9[90];
```

```
1[12],2[22];
1[13],3[33];
1[14],5[54];
1[15],9[95];
2[16],2[21];
2[18],4[38];
2[19],6[59];
2[20],10[100];
2[23],4[43];
2[24],6[64];
2[25],10[105];
3[26],3[31];
3[27],4[37];
3[29],7[69];
3[30],11[110];
3[32],4[42];
3[34],7[74];
3[35],11[115];
4[36],4[41];
4[39],8[79];
4[40],12[120];
4[44],8[84];
4[45],12[125];
5[46],5[51];
5[47],6[57];
5[48],7[68];
5[50],13[130];
5[52],6[62];
5[53],7[73];
5[55],13[135];
6[56],6[61];
6[58],8[78];
6[60],14[140];
6[63],8[83];
6[65],14[145];
7[66],7[71];
7[67],8[77];
7[70],15[150];
7[72],8[82];
7[75],15[155];
8[76],8[81];
8[80],16[160];
8[85],16[165];
```

```
9[86],9[91];
9[87],10[97];
9[88],11[108];
9[89],13[129];
9[92],10[102];
9[93],11[113];
9[94],13[134];
10[96],10[101];
10[98],12[118];
10[99],14[139];
10[103],12[123];
10[104],14[144];
11[106],11[111];
11[107],12[117];
11[109],15[149];
11[112],12[122];
11[114],15[154];
12[116],12[121];
12[119],16[159];
12[124],16[164];
13[126],13[131];
13[127],14[137];
13[128],15[148];
13[132],14[142];
13[133],15[153];
14[136],14[141];
14[138],16[158];
14[143],16[163];
15[146],15[151];
15[147],16[157];
15[152],16[162];
16[156],16[161];
```

```
/*************************************************************
  globals.h                                              1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

  Global header file that needs to be modified
  every time a new  configuration is to be run.
*************************************************************/
#ifndef globals_h
#define globals_h

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdarg.h>
#include <ctype.h>
#include "conc.h"

#define TRUE          !FALSE
#define FALSE         0

#define LMAX   32
#define PMAX   16
#define DIM    5            /* log of LMAX */
#define MAX_BUFFER_SIZE  (LMAX+2)
#define DIMSTAT    20

/*     Macros     */
#define logical_node_number(i) (((_node_number-1)
                                  *(LMAX/PMAX))+i)
#define VChanNumber(lnn, dim)  (DIM*lnn+6+dim)

#endif
```

```
/*************************************************************
  Inebr:packet.h                                        1/28/93

  S. Mitra and D. Thiebaut               Dept. Computer Science
  smitra@sophia.smith.edu                         Smith College
  thiebaut@sophia.smith.edu               Northampton, MA 01063

  Packet.h contains the definition of the packets that are
  passed around.

*************************************************************/
#ifndef PACKET_H
#define PACKET_H
#include "globals.h"

#define GROUP1        1
#define GROUP2        0

#define INFO        0 /* "computation" packet                */
#define NODE_DONE   1 /* packet sent when node is done       */
#define ROOT_DONE   2 /* packet sent by root when
                         everybody's done                    */

#define PRINT_STATS 3       /* command for printing          */
#define LAST_NODE_PRINTED 4 /* when last node has printed the*/
                            /* statistics, it will send this */
                            /* packet to Root, which will    */
                            /* close down the network        */

typedef struct packet PACKET_TYPE;
struct packet
{
     char      source;
     char      dest;
     char      type;
     int       TimeSent;
     int       TimeReceived;
     int       NoHops;
     char      QueueSize[DIMSTAT];
     char      NodePopulation[DIMSTAT];
     char      NodeID[DIMSTAT];
};
#define PACKET_SIZE  (sizeof(struct packet))
```

```
typedef struct packet_node PACKET_NODE;
typedef PACKET_NODE *PACKET_PTR;
struct packet_node
{
     PACKET_TYPE StoredPacket;
     PACKET_PTR  next;
};
#define PACKET_NODE_SIZE  (sizeof(struct packet_node))

#endif
```

```
/*****************************************************************
  bufqueue.h                                                1/28/93

  S. Mitra and D. Thiebaut               Dept. Computer Science
  smitra@sophia.smith.edu                        Smith College
  thiebaut@sophia.smith.edu               Northampton, MA 01063

  Header file for Queue Management routines
*****************************************************************/
#ifndef BUFQUEUE_H
#define BUFQUEUE_H

#include "packet.h"
#include "globals.h"

typedef struct QueueType QUEUE_TYPE;
struct  QueueType
{
     int  MaxBuffer;
     char *Buffer;
     int  Front;
     int  Back;
     char Full;
     char Empty;
     char Size;
     Semaphore Sem;
     int       VchanNo;   /* virtual channel #    */
     Channel   *link;     /* virtual channel      */
     Channel   *signal;   /* soft channel to awake
                                output manager */
};

QUEUE_TYPE *InitQueue(int MaxPackets, int VchanNo,
            Channel *link);
void EnqueuePacket(QUEUE_TYPE *Q, PACKET_TYPE *Packet);
int  EnqueuePacketSkip(QUEUE_TYPE *Q,
     PACKET_TYPE *Packet);
void QueueOutManager(Process *P, QUEUE_TYPE *Q);
void QueueInManager(Process *P, Channel *Vchan,
     QUEUE_TYPE *Q,QUEUE_TYPE *Buffer[] );
int  CountQPopulation(QUEUE_TYPE *Buffer[]);
#endif
```

```
/*****************************************************************
  bufqueue.c

  bufqueue.c                                                1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                        Smith College
  thiebaut@sophia.smith.edu               Northampton, MA 01063

  This module contains functions implementing typical queue
  operations for managing the queues attached to the virtual
  channels.

*****************************************************************/
#define  DEBUG_BUF
#undef   DEBUG_BUF

#include "packet.h"
#include "bufqueue.h"
/*****************************************************************/
/* SAFEMALLOC: Allocate dynamic storage. If memory is not       */
/* available, return error message.                             */
/*****************************************************************/
void *safemalloc(int size)
{
     void *p;
     p = malloc(size);
     if (p==NULL)
     {
          printf("Out of dynamic memory\n");
          exit(1);
     }
     return p;
}

/*****************************************************************/
/* INITQUEUE                                                     */
/*****************************************************************/
QUEUE_TYPE *InitQueue(int MaxPackets, int VchanNo,
                      Channel *Link)
{
     QUEUE_TYPE *Q;
```

```
     Q = (QUEUE_TYPE *) safemalloc(sizeof(QUEUE_TYPE));
     Q->VchanNo   = VchanNo;
     Q->MaxBuffer = MaxPackets * PACKET_SIZE;
     Q->Buffer    = (char *) safemalloc((Q->MaxBuffer+1)
                    *sizeof(char));
     Q->Full      = 0;
     Q->Empty     = 1;
     Q->Front     = 0;
     Q->Back      = 0;
     Q->Size      = 0;
     Q->Sem.use   = 0;
     Q->Sem.head  = (int *) NOPROCESS;
     Q->Sem.tail  = (int *) NOPROCESS;
     Q->link      = Link;
     Q->signal    = ChanAlloc();
     if (Q->signal==NULL)
     {
        printf("Out of dyn memory\n");
        exit(1);
     }

     return Q;
}
/*****************************************************************/
/* QUEUESTATUS                                                   */
/*****************************************************************/
void QueueStatus(QUEUE_TYPE *Q, char *msg)
{
     #ifdef DEBUG_BUF
     printf("[%s F%d E%d Fr%d Bk%d Sem%d]\n",
                msg,Q->Full,Q->Empty,Q->Front,
                Q->Back,Q->Sem.use);
     #endif
}
/*****************************************************************/
/* COUNT QUEUE POPULATION: counts the number of                  */
/*   packets in all the queues for a node                        */
/*****************************************************************/
int  CountQPopulation(QUEUE_TYPE *Buffer[])
{
     int i;
     int population = 1;
     int Current_Priority;
```

```
     Current_Priority = ProcToHigh();

     for (i = 0; i < DIM; i++)
        population+= Buffer[i]->Size;

     if (Current_Priority == 1)
        ProcToLow();

     return population;
}
/*****************************************************************/
/* ENQUEUE: Enqueues x in Q.  Blocks if Q is full .Works on a    */
/*          packet basis, i.e. Buffer becomes not empty only     */
/*          after a full packet is entered.                      */
/*****************************************************************/
void EnqueuePacket(QUEUE_TYPE *Q, PACKET_TYPE *Packet)
{
     int i, AwakeManager;
     int CurrentPriority;

     if (Q->Full)
     {
        CurrentPriority = ProcToLow();
        while (Q->Full)
            ProcReschedule();
        if (CurrentPriority == 0)
            ProcToHigh();
     }
     (Q->Size)++;
     memcpy( (void *)&(Q->Buffer[Q->Front]),
             (void *)Packet, PACKET_SIZE);
     Q->Front = ( (Q->Front+PACKET_SIZE)%Q->MaxBuffer);
     Q->Full = (Q->Size >= Q->MaxBuffer/PACKET_SIZE);
     Q->Empty = 0;
}
/*****************************************************************/
/* QUEUEINMANAGER:  Parallel task managing input                 */
/*                  into queue to a channel.                     */
/*****************************************************************/
void QueueInManager(Process *P, Channel *VChan,
                    QUEUE_TYPE *Q , QUEUE_TYPE *Buffer[])
{
```

```
     PACKET_TYPE packet;
     int  node;

     /*--- run forever! ---*/
     while (1)
     {
          VChanIn(VChan, &packet, PACKET_SIZE);

          node = packet.NoHops;
          if (node < DIMSTAT)
          {
             packet.QueueSize[node]  = (Q->Size);
             /*
             # of packets not including itself.
             ie. QSize when the packet comes in
             */

             packet.NodePopulation[node] =
                              CountQPopulation(Buffer);
          }
          EnqueuePacket(Q, &packet);
     }
}
/*****************************************************************/
/* QUEUEOUTMANAGER:  Parallel task managing output               */
/*                   from queue to a channel.                    */
/*****************************************************************/
void QueueOutManager(Process *P, QUEUE_TYPE *Q)
{
     char temp;
     PACKET_TYPE Packet;
     int CurrentPriority, i;

     /*--- run forever! ---*/
     while (1)
     {
       if (Q->Empty)
       {
          CurrentPriority = ProcToLow();
          while (Q->Empty)
              ProcReschedule();
          if (CurrentPriority == 0)
              ProcToHigh();
```

```
        }
        VChanOut(Q->link,&(Q->Buffer[Q->Back]),PACKET_SIZE);
        (Q->Size)--;
        Q->Back  = (Q->Back+PACKET_SIZE) % (Q->MaxBuffer);
        Q->Empty = (Q->Size == 0);
        Q->Full  = 0;
      }
}
```

```
/*************************************************************
  params.h                                              1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

  params.h contains headers for params.c
*************************************************************/
enum PARAM_TYPES  {INTEGER, LONG, CHAR, STRING,
                   FLOAT, DOUBLE};

void GetParams(int argc, char *argv[], char *Switch,
               int type, void *result);
```

```
/*************************************************************
  params.c                                                1/28/93

  S. Mitra and D. Thiebaut               Dept. Computer Science
  smitra@sophia.smith.edu                         Smith College
  thiebaut@sophia.smith.edu               Northampton, MA 01063

  Module containing functions to access and parse command
  line arguments.
*************************************************************/

/*
 Examples:
 ---------
 GetParams(argc, argv, "-i", INTEGER, &i);
 GetParams(argc, argv, "-s", STRING,  s);
 GetParams(argc, argv, "-l", LONG,   &l);
 GetParams(argc, argv, "-f", FLOAT,  &f);
 GetParams(argc, argv, "-c", CHAR,   &c);
 GetParams(argc, argv, "-d", DOUBLE, &d);
*/
/*************************************************************/
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include "params.h"

/*
  Examples:
  ---------
  GetParams(argc, argv, "-s", INTEGER, &i);
  GetParams(argc, argv, "-s", STRING,  s);
  GetParams(argc, argv, "-s", LONG,   &l);
  GetParams(argc, argv, "-s", FLOAT,  &f);
  GetParams(argc, argv, "-s", CHAR,   &c);
  GetParams(argc, argv, "-s", DOUBLE, &d);
*/
/*************************************************************/
/* GETPARAMS: get the parameters                             */
/*************************************************************/
void GetParams(int argc, char *argv[], char *Switch,
               int type, void *result)
```

```
{
     int    i, tempi;
     long   templ;
     char   tempc;
     char   temps[200];
     float  tempf;
     double tempd;

     for (i=0; i<argc; i++)
     {
       if (strcmp(argv[i],Switch)==0)
       {
         switch (type)
         {
           case INTEGER:*((int *) result) = atoi(argv[i+1]);
                        return;
           case STRING: strcpy((char *) result,argv[i+1]);
                        return;
           case LONG:   *((long *) result)   = atol(argv[i+1]);
                        return;
           case FLOAT:  *((float *) result) = (float)atof(argv[i+1]);
                        return;
           case CHAR:   *((char *) result) = argv[i+1][0];
                        return;
           case DOUBLE: *((double *) result) = atof(argv[i+1]);
                        return;
         }
       }
     }
}
```

```
/*************************************************************
  sync.h                                                1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

  sync.h contains headers for synchronization of several
  transputer nodes such that all the processors maintain the same
  high priority time.
*************************************************************/
#ifndef SYNC_H
#define SYNC_H

#include "globals.h"

#define PING_PONG 0      /* Delay "computation" packet */
#define SET_TIME  1      /* packet sent with time to which
                            remote node should adjust its
                            clock to */
#define SYNC_OVER 2      /* packet sent by root when
                            synchronization is done */
#define TIMER   1000000  /* 1 second of HP time */

typedef struct sync_packet SYNC_PACKET_TYPE;
struct sync_packet
{
     char      source;
     char      dest;
     char      type;
     int       time;
};
#define SYNC_PACKET_SIZE  (sizeof(struct sync_packet))

void Sync(int lnn);
int  Ping(int node, int lnn);
void Relay(SYNC_PACKET_TYPE packet, int lnn);
void RemoteTimeSet(int node, int delay);
void SyncDoneInform(int node);

#endif
```

```
/*****************************************************************
  sync.c                                                  1/28/93

  S. Mitra and D. Thiebaut               Dept. Computer Science
  smitra@sophia.smith.edu                         Smith College
  thiebaut@sophia.smith.edu               Northampton, MA 01063

  sync.c contains code for synchronization of several transputer
  nodes such that all the processors maintain the same high
  priority time.
*****************************************************************/
#include "sync.h"
/*****************************************************************/
/* SYNCHRONIZE                                                   */
/*****************************************************************/
void Sync(int lnn)
{
     int node, i, index;
     int Done = 0;
     int delay = 0;
     Channel *VChanList[DIM+1];
     SYNC_PACKET_TYPE  packet;
     int increment = LMAX/PMAX;

     ProcToHigh();
     srand(lnn+3*Time());
     /*
      ====================================
      ROOT
      ====================================
     */
     if (lnn == 0)
     {
        SetTime(10);      /*Sets the HP Timer */
        for (node = 0+increment; node < LMAX; node+=increment)
        {
            delay = 0;
            for (i = 0; i < 20; i++)
            {
                delay+= Ping(node,lnn);
                #ifdef DEBUG_SYNC
                printf("time = %d\n", time1);
                #endif
```

```
        }
        delay = delay/10/2;
        RemoteTimeSet(node, delay);
    }
    ProcWait(TIMER);
    for (node = 1; node < LMAX; node++)
        SyncDoneInform(node);
    ProcWait(TIMER);
    SwitchSS16();
    printf("Sync Done\n");
    SwitchSS16();

}
/*
 ====================================
 NON ROOT NODES
 ====================================
*/

else
{
    for (i = 0; i < DIM; i++)
        VChanList[i] = VChan(VChanNumber(lnn, i));
    VChanList[DIM] = 0;

    while (1)
    {
        if (Done)
            index = VProcTimerAltList(Time()+TIMER, VChanList);
        else
            index = VProcAltList(VChanList);

        if (index == -1)
            break;

        VChanIn(VChanList[index], &packet, SYNC_PACKET_SIZE);

        if ((packet.dest==lnn) && (packet.type==PING_PONG))
        {
            packet.dest = 0;
            Relay(packet, lnn);
        }
```

```
            else if ((packet.dest==lnn) && (packet.type==SET_TIME))
               SetTime(packet.time);

            else if ((packet.dest==lnn) && (packet.type==SYNC_OVER))
               Done = 1;

            else if (packet.dest != lnn)
               Relay(packet, lnn);
         }
      }
      ProcToLow();
}
/*************************************************************/
/* REMOTE TIME SET                                           */
/*************************************************************/
void RemoteTimeSet(int node, int delay)
{
     SYNC_PACKET_TYPE packet;

     packet.type   = SET_TIME;
     packet.source = 0;
     packet.dest   = node;
     packet.time   = Time()+delay;

     Relay(packet, 0);
}
/*************************************************************/
/* SYNC DONE INFORM                                          */
/*************************************************************/
void SyncDoneInform(int node)
{
     SYNC_PACKET_TYPE packet;

     packet.type   = SYNC_OVER;
     packet.source = 0;
     packet.dest   = node;
     packet.time   = 0;

     Relay(packet, 0);
}
/*************************************************************/
/* RELAY: Relays a packet during synchronization phase       */
/*************************************************************/
```

```
void Relay(SYNC_PACKET_TYPE packet, int lnn)
{
     int i, base;
     Channel *VChannel;
     int VChanNo;

     base = 1;
     for (i = 0; i < DIM; i++, base*=2)
     {
         if ( (lnn^packet.dest) & base)
             break;
     }

     VChanNo  = VChanNumber(lnn,i);
     VChannel = VChan(VChanNo);

     VChanOut(VChannel, &packet, SYNC_PACKET_SIZE);
}
/*************************************************************/
/* PING: Send packet to a node and get it back. Compute time   */
/*       taken in travelling back and forth.                   */
/*************************************************************/
int Ping(int node, int lnn)
{
     SYNC_PACKET_TYPE packet;
     int TimeSent, TimeRcvd;
     int Delay;
     int i;
     int index;
     Channel *VChanList[DIM+1];


     for (i = 0; i < DIM; i++)
         VChanList[i] = VChan(VChanNumber(lnn, i));
     VChanList[DIM] = 0;

     packet.source = 0;
     packet.dest   = node;
     packet.type   = PING_PONG;

     TimeSent      = Time();

     Relay(packet, 0);
```

```
    index = VProcAltList(VChanList);
    VChanIn(VChanList[index], &packet, SYNC_PACKET_SIZE);

    TimeRcvd = Time();
    Delay = TimeRcvd - TimeSent;

    return Delay;
}
```

```
/*****************************************************************
  print.h                                                  1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

  Header for the print module
*****************************************************************/
#ifndef PRINT_H
#define PRINT_H

#include "packet.h"
#include "globals.h"


void PrintStart(int lnn, PACKET_PTR Packet_Lst,
     FILE *fp,int PACKET_PRINT_MODE);
void printpacket(PACKET_PTR Packet_Lst, FILE *fp);

#endif
```

```
/*****************************************************************
  print.c                                                  1/28/93

  S. Mitra and D. Thiebaut               Dept. Computer Science
  smitra@sophia.smith.edu                        Smith College
  thiebaut@sophia.smith.edu               Northampton, MA 01063

  print.c contains the functions to print the statistics.
*****************************************************************/
#include "print.h"
/*****************************************************************/
/*  PRINTSTART                                                   */
/*****************************************************************/
void PrintStart(int lnn, PACKET_PTR Packet_Lst, FILE *fp,
                int PACKET_PRINT_MODE)
{
   if (PACKET_PRINT_MODE)
   {
       printpacket(Packet_Lst, fp); /* will free list */
       Packet_Lst = NULL;
   }
}
/*****************************************************************/
/* PRINT PACKET                                                  */
/*****************************************************************/
void printpacket(PACKET_PTR Packet_Lst, FILE *fp)
{
  int  i,j;
  PACKET_PTR Head=Packet_Lst, p, q;

  while (1)
  {
      /*--- find last packet in list (since it's the oldest) ---*/
      for (p = Head, q = NULL; p!=NULL; q=p, p=p->next);
      if (q==NULL)
           break;

      fprintf(fp,"[%3d ----> %3d]  %8d us    %2d hops  ",
            q->StoredPacket.source, q->StoredPacket.dest,
            (q->StoredPacket.TimeReceived)-
            (q->StoredPacket.TimeSent),
            q->StoredPacket.NoHops);
      fprintf(fp,"-> %3d   Q= %3d  Pop= %3d\n",
```

```
            q->StoredPacket.NodeID[0],
            q->StoredPacket.QueueSize[0],
            q->StoredPacket.NodePopulation[0]);
      for (i = 1; i < q->StoredPacket.NoHops; i++)
      {
         if (q->StoredPacket.NoHops < DIMSTAT)
            fprintf(fp,"%41s-> %3d   Q= %3d  Pop= %3d\n"," ",
            q->StoredPacket.NodeID[i],q->StoredPacket.QueueSize[i],
            q->StoredPacket.NodePopulation[i]);
         else
            break;
      }

      /*--- Remove last packet. ---*/
      if (q!=Head)
      {
          for (p = Head; p->next!=q; p=p->next);
          free(q);
          p->next = NULL;
      }
      else
      {
          free(q);
          break;
      }
  }
}
```

```
/*************************************************************
  Greenberg:packet.h                                    1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

  Packet.h contains the definition of the packets that are
  passed around.

*************************************************************/

#ifndef PACKET_H
#define PACKET_H
#include "globals.h"

#define TRUE_TAKEN    2
#define FALSE_TAKEN   3

#define INFO        0       /* "computation" packet */
#define NODE_DONE   1       /* packet sent when node is done */
#define ROOT_DONE   2       /* packet sent by root when
                               everybody's done */
#define PRINT_STATS 3       /* command for printing          */
#define LAST_NODE_PRINTED 4 /* when last node has printed the*/
                            /* statistics, it will send thi  */
                            /* packet to Root, which will    */
                               close down the network        */

typedef struct packet PACKET_TYPE;
struct packet
{
     char      source;
     char      dest;
     char      type;
     int       TimeSent;
     int       TimeReceived;
     int       NoHops;
     char      QueueSize[DIMSTAT];
     char      NodePopulation[DIMSTAT];
     char      NodeID[DIMSTAT];
};
#define PACKET_SIZE  (sizeof(struct packet))
```

```
typedef struct packet_node PACKET_NODE;
typedef PACKET_NODE *PACKET_PTR;
struct packet_node
{
     PACKET_TYPE StoredPacket;
     PACKET_PTR  next;
};
#define PACKET_NODE_SIZE  (sizeof(struct packet_node))

#endif
```

```
/*************************************************************
  greenber.c                                          1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

  This  program  implements a logical Hypercube of transputers
  using  virtual  channels  for inter-node communication. Each
  logical  node  generates a  random  destination. A  packet
  containing  the  source, destination, time received and time
  sent is prepared and  sent  out  by each node in a direction
  determined  by  XORing  the  logical  node number with the
  destination node number.

  The algorithm  followed is Greenberg's deflection routing, in
  which all packets arriving at a node are considered in random
  order. When  a  packet is considered, the node examines which
  outgoing  links  are  on  shortest  paths  to  the  packet's
  destination (preferred links) and sees if any of those are
  free ie. not already assigned to another packet. If at least
  one  preferred  link  is  free, the  packet  is assigned to a
  preferred, free  link at random. Otherwise,  the  packet  is
  assigned to a free link (also randomly chosen), and the packet
  is said to be deflected. If all preferred and free links are
  taken, then the excess arrivals are queued. (One of the options
  suggested by Greenberg in "Deflection Routing in Hypercube
  Networks" (pp 1077).
*************************************************************/
/* PICK RANDOM DIM: Out of all the links that are on the      */
/* shortest route to the destination, pick one with an empty  */
/* queue. If none with an empty queue is found, pick any link */
/* with empty queue. Else buffer packets.                     */
/*************************************************************/
int PickRandomDim(int lnn, PACKET_TYPE packet,
                  QUEUE_TYPE *Buffer[] )
{
    int array[DIM];
    int i, base;
    int All_True_Taken=TRUE; /* Flag to indicate that none of
                             the preferred links is empty */
    int All_False_Taken=TRUE; /* Flag to indicate that none of
                         the not-preferred links is empty */
```

```
    base = 1;
    for (i = 0; i < DIM; i++, base*=2)
    {
        if ((lnn^packet.dest) & base)
        {
            if (Buffer[i]->Empty)
            {
                array[i]       = TRUE;
                All_True_Taken = FALSE;
            }
            else
                array[i]       = TRUE_TAKEN;
        }
        else
        {
            if (Buffer[i]->Empty)
            {
                array[i]        = FALSE;
                All_False_Taken = FALSE;
            }
            else
                array[i]        = FALSE_TAKEN;
        }
    }
    i = rand() % DIM;

    if (!All_True_Taken)      /*If all preferred links are not*/
    {                         /*taken, then randomly find an  */
      while (array[i] != TRUE)/*available preferred link      */
        i = rand() % DIM;
    }
    else if (!All_False_Taken)/*If all preferred links are    */
    {                         /*taken, see which not-preffered*/
      while (array[i]!=FALSE) /*link is free                  */
        i = rand() % DIM;
    }                         /*If all preferred and not-     */
                              /*preffered links are taken,    */
                              /*just choose a random link.    */
    return i;
}
```

```
/*************************************************************
  Valiant:packet.h                                      1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

  Packet.h contains the definition of the packets that are
  passed around.

*************************************************************/
#ifndef PACKET_H
#define PACKET_H
#include "globals.h"


#define INFO        0       /* "computation" packet */
#define NODE_DONE   1       /* packet sent when node
                               is done */
#define ROOT_DONE   2       /* packet sent by root when
                               everybody's done */
#define PRINT_STATS 3       /* command for printing
                               statistics  */
#define LAST_NODE_PRINTED 4 /* when last node has printed
                               the statistics, it will send
                               this packet to Root, which will
                               close down the network
                            */

typedef struct packet PACKET_TYPE;
struct packet
{
     char      source;
     char      dest;
     char      type;
     char      Phase1dest;
     char      Actualdest;
     char      PhaseFlag;
     int       TimeSent;
     int       TimeReceived;
     int       NoHops;
     char      QueueSize[DIMSTAT];
     char      NodePopulation[DIMSTAT];
```

```
     int       NodeID[DIMSTAT];
};
#define PACKET_SIZE  (sizeof(struct packet))

typedef struct packet_node PACKET_NODE;
typedef PACKET_NODE *PACKET_PTR;
struct packet_node
{
     PACKET_TYPE StoredPacket;
     PACKET_PTR  next;
};
#define PACKET_NODE_SIZE  (sizeof(struct packet_node))

#endif
```

```
/*************************************************************
  valiant.c                                            1/28/93

  S. Mitra and D. Thiebaut              Dept. Computer Science
  smitra@sophia.smith.edu                       Smith College
  thiebaut@sophia.smith.edu              Northampton, MA 01063

  This program implements a logical Hypercube of transputers
  using virtual channels for inter-node communication. Each
  logical node generates a random destination. A packet containing
  the source, destination, time received and time sent is prepared
  and sent out by each node to a random destination in Phase 1,
  and is routed to its Actual destination in Phase 2. The
  algorithm is due to Les Valiant.
*************************************************************/
int  GeneratePhase1Dest(int source);
/*************************************************************/
/* TASK1:  Sends a packet to a random destination             */
/*************************************************************/
void Task1(Process *p, int lnn, Channel *T1Rsoft)
{
   PACKET_TYPE  packet;
   int          i, j;
   int          StartTime;

   /*--- send a packet(s) to a random destination ---*/
   packet.source   = lnn;
   packet.type     = INFO;
   packet.NoHops   = -1;
   packet.PhaseFlag  = ONE;
   for (j = 0; j < DIMSTAT; j++)
   {
        packet.QueueSize[j]      = 0;
        packet.NodePopulation[j] = 0;
        packet.NodeID[j]         = 0;
   }
   ProcToHigh();
   ProcAfter(BARRIER_4);
   ProcToLow();
   for (i=0; i<NO_RANDOM_PACKETS; i++)
   {
      packet.Phase1dest = GeneratePhase1Dest(packet.source);
      packet.Actualdest = GenerateRandomDest(packet.source);
```

```
      packet.dest       = packet.Phase1dest;

      if (FlipCoin())
           ProcReschedule();

      packet.TimeSent = TimeHigh();
      ChanOut(T1Rsoft, &packet, PACKET_SIZE);
   }

   /*--- Now tell the root that we are done ---*/
   if (lnn != 0)
   {
      packet.dest       = 0;
      packet.type       = NODE_DONE;
      packet.Phase1dest = lnn;
      packet.Actualdest = 0;
      packet.PhaseFlag  = TWO;
      ChanOut(T1Rsoft, &packet, PACKET_SIZE);
   }
}
/*****************************************************************/
/* TASK2:                                                        */
/*****************************************************************/
void Task2(Process *p, int lnn, Channel *T2Rsoft,
           Channel *R2Tsoft, FILE *fp)
{
   float  thrput = 0;
   PACKET_TYPE packet;
   int i, index, j;
   int Done      = 0;
   int NodesDone = 1;/*1 because the root will be done first!*/
   PACKET_PTR Packet_Lst = NULL;
   int max = 0;
   int min = 9000000;
   int FLAG = 0;
   float Delay = 0.0;
   int DelayCounter = 0;
   int Hop = 0;
   int Shortest = 0;
   int QS = 0;
   int Pop = 0;
   int QCnt = 0;
   int PCnt = 0;
```

```
while (1)
{
    if (Done)
         index = ProcTimerAlt(Time()+TIMEOUT, R2Tsoft, 0);
    else
         index = ProcAlt(R2Tsoft, 0);

    if (index == -1)
         break;

    ChanIn(R2Tsoft, &packet, PACKET_SIZE);

    if ((packet.type==INFO) || (packet.type==NODE_DONE))
         packet.TimeReceived = TimeHigh();
    if (packet.type != NODE_DONE)
    {
         min = Min(min, packet.TimeSent);
         max = Max(max, packet.TimeReceived);
    }
    switch (packet.type)
    {
      case INFO:
           if (PACKET_PRINT_MODE)
                Packet_Lst = StorePacket(packet, Packet_Lst);
           Delay+= 1.0*(packet.TimeReceived-packet.TimeSent);
           DelayCounter++;
           Hop+= packet.NoHops;
           Shortest+= ShortestPath(packet.source, packet.dest);
           for (i = 0; i < packet.NoHops; i++)
           {
                if (packet.NoHops >= DIMSTAT)
                    break;
                QS+= packet.QueueSize[i];
                QCnt++;
                Pop+= packet.NodePopulation[i];
                PCnt++;
           }
           break;
     case NODE_DONE:
           /* we are the logical root: Increment count */
           NodesDone++;
```

```
        /*    if all the other nodes have responded, not
              including the root, tell them to start printing
              */
        if (NodesDone == LMAX)
        {
           SwitchSS16();
           PrintStart(lnn,Packet_Lst, fp,PACKET_PRINT_MODE);
           if (DelayCounter)
           {
               fprintf(fp, "\"Node\" %d %3f %d %d %d %d %d
               %d %d \n", lnn, Delay, DelayCounter, Hop,
               Shortest, QS, QCnt, Pop, PCnt);
           }
           /*--- root prints its own stats ---*/
           /*--- then it passes the token around ---*/
           packet.type       = PRINT_STATS;
           packet.source     = lnn;
           packet.PhaseFlag  = TWO;
           packet.Phase1dest = lnn;
           packet.dest       = 1;
           packet.Actualdest = 1;
           packet.NoHops     = 0;
           packet.TimeSent  = min;
           packet.TimeReceived = max;
           ChanOut(T2Rsoft, &packet, PACKET_SIZE);
        }
        break;
case PRINT_STATS:
        PrintStart(lnn,Packet_Lst, fp,PACKET_PRINT_MODE);
        if (DelayCounter)
        {
           fprintf(fp, "\"Node\" %d %3f %d %d %d %d
           %d %d %d \n", lnn, Delay, DelayCounter, Hop,
           Shortest, QS, QCnt, Pop, PCnt);
        }
        packet.TimeSent  = min;
        packet.TimeReceived = max;

        if (lnn < (LMAX-1) )
        {
           packet.type       = PRINT_STATS;
           packet.dest       = lnn+1;
           packet.Actualdest = lnn+1;
```

```
                }
                else if (lnn == (LMAX-1) )
                {
                   packet.type      = LAST_NODE_PRINTED;
                   packet.dest      = 0;
                   packet.Actualdest= 0;
                }
                else
                {
                    printf("\n*** ERROR! *** lnn = %d\n\n",lnn);
                    ProcWait(1000000/64);
                }
                packet.source    = lnn;
                packet.PhaseFlag = TWO;
                packet.Phase1dest= lnn;
                packet.NoHops     = 0;

                ChanOut(T2Rsoft, &packet, PACKET_SIZE);
                Done = 1;
                break;

        case LAST_NODE_PRINTED:
                thrput=((1.0*NO_RANDOM_PACKETS*LMAX)/(max-min))
                         *1000000;
                fprintf(fp,"-------------------------------------
                -----------\n\n");
                fprintf(fp,"  THROUGHPUT          = %10.3f\n",thrput);
                fprintf(fp,"-------------------------------------
                -----------\n\n");
                printf("END OF COMPUTATION (%d)\n",lnn);
                fprintf(fp,"END OF COMPUTATION (%d)\n",lnn);
                Done = 1;
                break;

        default:
                printf("\n\n*** ERROR *** Packet received of
                       type %d\n\n", packet.type);
                ProcWait(1000000/64);
                break;
       } /* end switch */
   } /* end while */
}
/*************************************************************/
```

```
/*   ROUTE: Valiant's Two Phase routing algorithm              */
/*************************************************************/
void Route(Process *P, int lnn, QUEUE_TYPE *Buffer[],
           Channel *T1Rsoft, Channel *T2Rsoft, Channel *R2Tsoft,
           Channel *SoftcArray[])
{
     int i;
     PACKET_TYPE packet;
     int index = 0;
     int node;

     while (1)
     {
         index = ProcAltList(SoftcArray);

         ChanIn(SoftcArray[index], &packet, PACKET_SIZE);

         node = ++(packet.NoHops);

         if (packet.PhaseFlag == ONE)
              packet.NodeID[node]         = (0 - lnn);
         else if (packet.PhaseFlag == TWO)
              packet.NodeID[node]         = lnn;

         /*if packet reached its actual destination, then preempt*/
         /*algorithm.  This could happen in either phases        */
         if (packet.Actualdest==lnn)
         {
            EnqueuePacket(Buffer[DIM], &packet);
            continue;
         }
         if (packet.dest == lnn)
         {
            /*this happens only in Phase 1, and we are not the*/
            /*Actual destination                              */
            packet.dest = packet.Actualdest;
            packet.PhaseFlag = TWO;
         }
         i = PickRandomDim(lnn, packet);
         EnqueuePacket(Buffer[i], &packet);
     }
}
/*************************************************************/
```

```
/* GENERATE PHASE 1 DEST: generates a random destination       */
/*************************************************************/
int  GeneratePhase1Dest(int source)
{
   int i;

   i = rand() % LMAX;
   while (i == source)
      i = rand() % LMAX;
   return i;
}
/*************************************************************/
/* PICK RANDOM DIM: Randomly pick one of the preferred links   */
/*************************************************************/
int PickRandomDim(int lnn, PACKET_TYPE packet)
{
    int array[DIM];
    int i, base;

    base = 1;
    for (i = 0; i < DIM; i++, base*=2)
    {
        if ((lnn^packet.dest) & base)
            array[i] = TRUE;
        else
            array[i] = FALSE;
    }

    i = rand() % DIM;

    while (!(array[i]))
        i = rand() % DIM;

    return i;
}
```