3.4 Packet-Routing Algorithms

One of the most important components of any large-scale general-purpose parallel computer is its packet-routing algorithm. This is because most large-scale general-purpose parallel machines spend a large portion of their resources making sure that the right data gets to the right place within a reasonable amount of time.

We already studied packet-routing algorithms for arrays in Chapter 1 and meshes of trees in Chapter 2. Although the algorithms described in these chapters are optimal for arrays and meshes of trees, they are not especially efficient in a general setting. For example, the routing algorithms for arrays use few processors but are relatively slow. The routing algorithms for meshes of trees, on the other hand, are fast but use an excessive number of processors.

We have also studied the packet-routing problem for hypercubic networks in Sections 3.2 and 3.3. In particular, we showed how to solve any fixed N-packet permutation routing problem in $O(\log N)$ steps on an N-processor butterfly or shuffle-exchange graph in Theorems 3.12 and 3.16. The solution to a routing problem found by this approach is fast and efficient, but suffers from the limitation that there is no $O(\log N)$-step algorithm known for finding the routing paths on-line. In other words, we proved in Theorems 3.12 and 3.16 that there is a fast and efficient solution to any permutation routing problem on a hypercubic network, but we don't know how to find the solution quickly in parallel. For some applications, this constraint doesn't matter, since we can afford to precompute the solution (off-line) and then store the routing information in the network. For many applications, however, the limitation is crucial, since we may not know the routing problem ahead of time. For such applications, we will need to develop *on-line* routing algorithms (i.e., algorithms for which the local routing decisions are made without precomputation and without knowledge of the global routing problem).

In this section, we describe several on-line algorithms for routing on hypercubic networks. For the most part, the algorithms will perform quickly (taking $\Theta(\log N)$ steps) and efficiently (using N processors to route N packets), although all of the algorithms described in this section can perform very badly in the worst case. In Section 3.5, we will describe algorithms for sorting that can be used to construct routing algorithms that are guaranteed to always perform well, but the sorting-based algorithms are quite

complicated and are often not as useful in practice.

We begin our discussion of packet-routing algorithms with some definitions and a description of some of the most common routing models in Subsection 3.4.1. We then define the greedy routing algorithm, and analyze its worst-case performance in Subsection 3.4.2. Unfortunately, we will find that the worst-case performance of the greedy algorithm is very poor and that several important routing problems exhibit worst-case performance.

On the other hand, there are also large classes of important problems for which the greedy algorithm performs very well. For example, we will show in Subsection 3.4.3 that the greedy algorithm performs optimally for packing, spreading, and monotone routing problems. These special classes of routing problems arise in many applications, and we will use them frequently throughout the remainder of Chapter 3. For example, we show in Subsection 3.4.3 how to decompose an arbitrary routing problem into a sorting problem and a monotone routing problem. Since any monotone routing problem can be solved in $O(\log N)$ steps using the greedy algorithm, this gives us an automatic way to convert any sorting algorithm into a packet-routing algorithm. Even though sorting N items quickly on a hypercubic network is a challenging task, this means that all of the sorting algorithms that are described in Section 3.5 can be converted into packet-routing algorithms with very little additional effort.

Greedy algorithms also perform well for average-case routing problems. In fact, we will show in Subsection 3.4.4 that almost all N-packet-routing problems can be solved in $O(\log N)$ steps by using the greedy algorithm on an N-processor hypercubic network. Hence, we will find that the greedy algorithm is optimal (up to constant factors) for random routing problems in a hypercubic network. This fact is quite important, since so many parallel machines use variations of the greedy algorithm to solve routing problems. In addition, we can use this fact to design a randomized algorithm for solving worst-case problems. In particular, we will show in Subsection 3.4.5 how to use randomness to convert any worst-case one-to-one routing problem into two average-case problems, thereby solving *any* one-to-one routing problem in $O(\log N)$ steps with high probability.

One problem with the naive greedy algorithm is that it allows packets to pile up at certain nodes in the network, resulting in queues which (for most routing problems) can grow as large as $\Theta(\log N)$ in size. In Subsection 3.4.6, we show how to modify the naive greedy algorithm so that all the queues stay small, and so that the overall performance remains good.

We also show how to generalize the result to apply to a much larger class of networks (including arrays).

Another problem with the naive greedy algorithm is that it doesn't always work well for some many-to-one routing problems, even if randomization is used. In Subsection 3.4.7, we show how to modify the naive greedy algorithm to handle many-to-one routing problems. We also describe an effective strategy for combining packets that are headed for the same destination, if that is desired.

In Subsection 3.4.8, we describe a variation of the greedy routing algorithm known as the information dispersal algorithm. The information dispersal approach to routing makes use of coding theory to partition a packet into many subpackets, only some fraction of which need to be successfully routed in order for the contents of the packet to be reconstructed at the destination. As a consequence, some packet components that get stuck in a congested area or that encounter a faulty component can be discarded without harm. As we will see in Section 3.6, information dispersal is also a useful tool when it comes to organizing data in a distributed memory.

We conclude our study of packet routing with a discussion of circuit-switching and bit-serial routing algorithms in Subsection 3.4.9. The algorithms described in this subsection differ from those discussed in Subsections 3.4.4–3.4.8 in that each packet needs to have a dedicated, uncongested path through the network from its source to its destination in order for the message to be transmitted. Even in this more restricted routing model, however, we find that the greedy algorithm performs fairly well for most (i.e., random) routing problems.

3.4.1 Definitions and Routing Models

As was mentioned in Section 1.7, there are many different types of routing models. For the most part, we will focus our attention on the *store-and-forward* model (also known as the *packet-switching* model) of packet routing in Section 3.4. In the store-and-forward model, each packet is maintained as an entity that is passed from node to node as it moves through the network and a single packet can cross each edge during each step of the routing. Depending on the algorithm, we may or may not allow packets to pile up in queues located at each node. When queues are allowed, we will generally make efforts to keep them from getting very large.

In Subsection 3.4.9, we consider the *circuit-switching* (or *path-lockdown*)

model of routing. In the circuit-switching model of routing, the entire path from the source of a packet to its destination must be dedicated to the packet in order for the packet's data to be transmitted.

For the most part, we will focus our attention on *static* routing problems (i.e., those for which the packets to be routed are present in the network when the routing commences) in Section 3.4. Many of the results that we obtain can also be applied to *dynamic* routing problems (in which packets arrive at the network at arbitrary times and the routing proceeds in a continuous fashion), although we will only specifically discuss the case of dynamic routing problems in Subsection 3.4.4.

There are many different types of static routing problems. Generally, we will assume that each processor starts with at most one packet, and, for the most part, we will focus our attention on the simplest case of one-to-one routing problems. A routing problem is said to be *one-to-one* if at most one packet is destined for any processor and if each packet has precisely one destination. We will also consider many-to-one and one-to-many routing problems. A routing problem is said to be *many-to-one* if more than one packet can have the same destination. It is said to be *one-to-many* if a single packet can have multiple destinations (i.e., if copies of one packet need to be sent to more than one destination).

When many packets are headed for the same destination, the usual problems with congestion in the network can become even more severe. For example, if at most one packet can be delivered to its destination during each step, then most of the packets that are headed for a common destination will experience significant delays due to (if for no other reason) the bottleneck at the destination. Such bottlenecks are often referred to as *hot spots*. Needless to say, hot spots can be a serious problem since they can also cause packets which are headed for other destinations to become delayed.

We will describe many methods for overcoming or minimizing the effects of hot spots and bottlenecks resulting from multiple packets having the same destination. One approach to dealing with such problems is to allow packets that are headed for the same destination to be combined. When *combining* is allowed, we can merge two packets P_1 and P_2 into a single (possibly larger) packet provided that P_1 and P_2 are headed for the same destination and that P_1 and P_2 are contained in the same node at the same time. Packet-routing algorithms that make use of combining will be described in Subsections 3.4.3 and 3.4.7.

Throughout Section 3.4, we will insist that our algorithms be *on-line*. This means that each processor (or switch) must decide what to do with the packets that pass through it based only on its local control and the information carried with the packets. In particular, we will not allow a global controller to precompute routing paths as was done in the proofs of Theorems 3.12 and 3.16. As a consequence, our algorithms will be able to handle any packet-routing problem immediately using only local control.

As was mentioned previously in the text, the development of an efficient routing algorithm for a network enables that network to efficiently emulate any other network. More generally, it will enable us to get the right data to the right place at the right time. As a consequence of the on-line feature of the routing algorithm, we will also be able to emulate abstract parallel machines such as a parallel random access machine (PRAM). Methods for simulating PRAMs on hypercubic networks will be studied extensively in Section 3.6.

3.4.2 Greedy Routing Algorithms and Worst-Case Problems

We begin our study of packet routing algorithms on hypercubic networks by considering the problem of routing N packets from level 0 to level $\log N$ in a $\log N$-dimensional butterfly. In particular, we assume that each node $\langle u, 0\rangle$ on level 0 of the butterfly contains a packet that is destined for node $\langle \pi(u), \log N\rangle$ on level $\log N$ where $\pi : [1, N] \rightarrow [1, N]$ is a permutation. For example, we have illustrated an 8-packet routing problem in Figure 3-48. In this example, we have selected π to be the *bit-reversal permutation* (i.e., $\pi(u_1 \cdots u_{\log N}) = u_{\log N} \cdots u_1$, where $u_1 \cdots u_{\log N}$ denotes the binary representation of u).

At first glance, the routing problem shown in Figure 3-48 does not seem particularly difficult. Indeed, any of the packets in the problem can be easily routed to its destination simply by sending the packet along the unique path of length $\log N$ through the butterfly to its destination. For example, we have illustrated this path for the packet destined for node $\langle 001, 3\rangle$ in Figure 3-48.

In general, the unique path of length $\log N$ from a level 0 node $\langle u, 0\rangle$ to a level $\log N$ node $\langle v, \log N\rangle$ in the butterfly is known as the *greedy path* from $\langle u, 0\rangle$ to $\langle v, \log N\rangle$. In the *greedy routing algorithm*, each packet is constrained to follow its greedy path. When there is only one packet to route, it is easy to see that the greedy algorithm performs very well. Trouble can arise when many packets have to be routed in parallel, however.

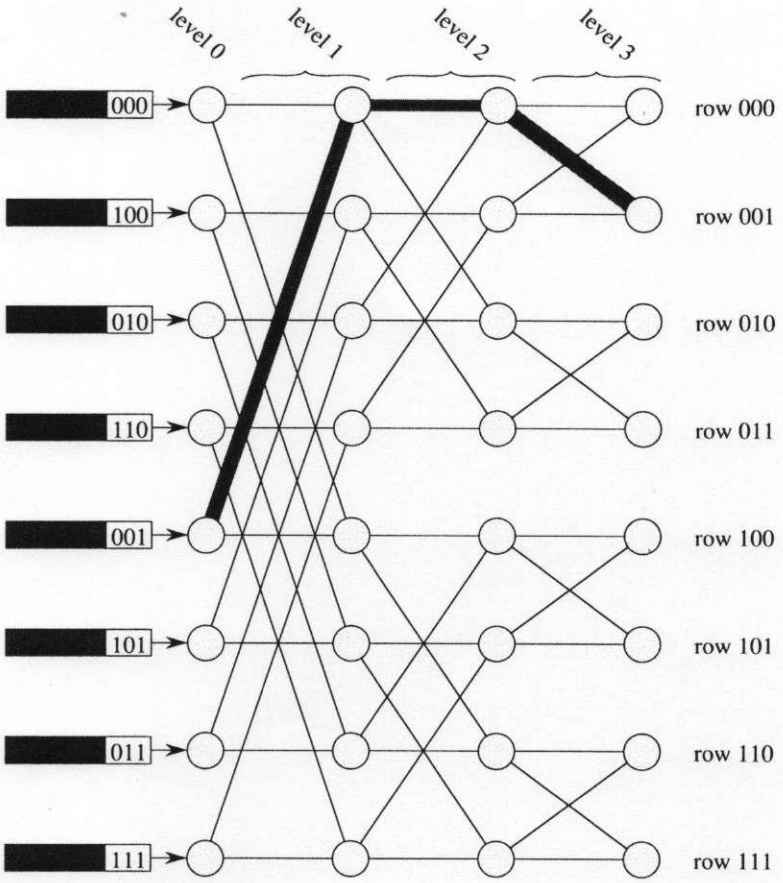

Figure 3-48 *An 8-packet routing problem on the three-dimensional butterfly. In this problem, the packet starting at node* $\langle u_1u_2u_3, 0\rangle$ *wants to go to node* $\langle u_3u_2u_1, 3\rangle$ *for each* $u_1u_2u_3$. *The greedy path for the packet starting at node* $\langle 100, 0\rangle$ *is shown in boldface.*

The problem is that many greedy paths might pass through a single node or edge. For example, the packets starting at nodes $\langle 000, 0\rangle$ and $\langle 100, 0\rangle$ in Figure 3-48 both must pass through edge $(\langle 000, 1\rangle, \langle 000, 2\rangle)$ on the way to their destinations. Since only one of these packets can use the edge at a time, one of them must be delayed before crossing the edge.

It is not difficult to check that the congestion problem arising in the example illustrated in Figure 3-48 is not overly serious. At most two packets will ever contend for a middle-level edge, and every packet can reach its destination in a total of 4 steps using the greedy algorithm.

When N is larger, however, the problem can be much more serious. In fact, a total of

$$2^{\frac{\log N-1}{2}} = \sqrt{N/2}$$

greedy paths will use the edge $\left(\langle 0\cdots 0, \frac{\log N-1}{2}\rangle, \langle 0\cdots 0, \frac{\log N+1}{2}\rangle\right)$ in a $\log N$-dimensional butterfly when the greedy algorithm is used to route N packets according to the bit-reversal permutation. (Here, we have assumed for simplicity that $\log N$ is odd. A similar result holds when $\log N$ is even. For example, see Problem 3.179.) The reason is that the packet which starts at node $\langle u_1\cdots u_{\frac{\log N-1}{2}}00\cdots 0, 0\rangle$ must travel to node $\langle 0\cdots 00u_{\frac{\log N-1}{2}}\cdots u_1, \log N\rangle$ along the path

$$\begin{aligned}
\langle u_1\cdots u_{\frac{\log N-1}{2}}00\cdots 0, 0\rangle &\rightarrow \left\langle 0u_2\cdots u_{\frac{\log N-1}{2}}00\cdots 0, 1\right\rangle\\
&\rightarrow \cdots\\
&\rightarrow \left\langle 0\cdots 0u_{\frac{\log N-1}{2}}00\cdots 0, \frac{\log N-3}{2}\right\rangle\\
&\rightarrow \left\langle 0\cdots 000\cdots 0, \frac{\log N-1}{2}\right\rangle\\
&\rightarrow \left\langle 0\cdots 000\cdots 0, \frac{\log N+1}{2}\right\rangle\\
&\rightarrow \left\langle 0\cdots 00u_{\frac{\log N-1}{2}}0\cdots 0, \frac{\log N+3}{2}\right\rangle\\
&\rightarrow \cdots\\
&\rightarrow \langle 0\cdots 00u_{\frac{\log N-1}{2}}\cdots u_1, \log N\rangle.
\end{aligned}$$

Note that this path contains the edge

$$e = \left\langle 0\cdots 000\cdots 0, \frac{\log N-1}{2}\right\rangle \rightarrow \left\langle 0\cdots 000\cdots 0, \frac{\log N+1}{2}\right\rangle,$$

since the middle bit of both $u_1 \cdots u_{\frac{\log N-1}{2}} 00 \cdots 0$ and $0 \cdots 00 u_{\frac{\log N-1}{2}} \cdots u_1$ is 0. There are $2^{\frac{\log N-1}{2}} = \sqrt{N/2}$ possible values of $u_1 \cdots u_{\frac{\log N-1}{2}}$, and thus $\sqrt{N/2}$ packets must traverse e when the greedy algorithm is used to route the packets. This means that at least one of the packets will be delayed by $\sqrt{N/2}-1$ steps, and that the greedy algorithm will take at least $\sqrt{N/2}+\log N-1$ steps to route all of the packets to their destinations. In fact, the greedy algorithm takes precisely $\sqrt{N/2}+\log N-1$ steps to route the bit-reversal permutation when $\log N$ is odd. (See Problem 3.181.)

Unfortunately, the bit-reversal permutation is not the only permutation that requires $\Theta(\sqrt{N})$ steps to route using the greedy algorithm. Indeed, many natural permutations exhibit poor performance with the greedy algorithm. For example, the commonly used *transpose permutation*

$$\pi\left(u_1 \cdots u_{\frac{\log N}{2}} u_{\frac{\log N}{2}+1} \cdots u_{\log N}\right) = u_{\frac{\log N}{2}+1} \cdots u_{\log N} u_1 \cdots u_{\frac{\log N}{2}}$$

also requires $\Theta(\sqrt{N})$ steps using the greedy algorithm. (See Problem 3.182.)

In fact, the bit-reversal permutation and the transpose permutation are (up to constant factors) worst-case permutations for greedy routing on the butterfly. This is because every one-to-one routing problem can be solved in $O(\sqrt{N})$ steps on a $\log N$-dimensional butterfly. We will prove this fact in the following theorem.

THEOREM 3.22 *Given any routing problem on a $\log N$-dimensional butterfly for which at most one packet starts at each level 0 node and at most one packet is destined for each level $\log N$ node, the greedy algorithm will route all the packets to their destinations in $O(\sqrt{N})$ steps.*

Proof. For simplicity, we will assume that $\log N$ is odd. The case when $\log N$ is even is handled in a similar fashion.

Let e be any edge in level i of the $\log N$-dimensional butterfly ($0 < i \leq \log N$), and define n_i to be the number of greedy paths that traverse e. We first observe that $n_i \leq 2^{i-1}$ for every i. This is because there are at most 2^{i-1} nodes at level 0 which can reach e using a path through levels $1, 2, \ldots, i-1$. For example, only the packets starting at nodes $\langle 000, 0\rangle$ and $\langle 100, 0\rangle$ can use the edge $(\langle 000, 1\rangle, \langle 000, 2\rangle)$ in a three-dimensional butterfly, no matter where each packet is destined.

Similarly, $n_i \leq 2^{\log N - i}$ for every i. This is because there are at most $2^{\log N - i}$ nodes at level $\log N$ which can be reached from e using a path through levels $i+1, i+2, \ldots, \log N$. For example, only the packets

ending at nodes $\langle 000, 3\rangle$ and $\langle 001, 3\rangle$ can use the edge $(\langle 000, 1\rangle, \langle 000, 2\rangle)$, no matter where the packets originate.

Since any packet crossing e can only be delayed by the other $n_i - 1$ packets that want to cross the edge, the total delay encountered by any packet as it traverses levels 1, 2, ..., $\log N$ can be at most

$$\begin{aligned}\sum_{i=1}^{\log N}(n_i - 1) &\leq \sum_{i=1}^{\frac{\log N+1}{2}} 2^{i-1} + \sum_{i=\frac{\log N+3}{2}}^{\log N} 2^{\log N - i} - \log N \\ &= 2^{\frac{\log N+1}{2}} + 2^{\frac{\log N-1}{2}} - \log N - 2 \\ &= \frac{3\sqrt{N}}{\sqrt{2}} - \log N - 2.\end{aligned}$$

Hence, the total time to complete any one-to-one routing problem is at most $O(\sqrt{N})$, as claimed. ■

The preceding analysis does not specifically deal with the problem of packets piling up in queues. Indeed, the queues at nodes in the middle levels of the butterfly might grow to be as large as $\Theta(\sqrt{N})$ if we do not limit their size. (See Problem 3.183.) We can restrict the growth of the queues by not allowing any packet to move forward across an edge if there are too many packets (say q) in the queue at the other end of the edge. The problem with limiting the queue sizes, however, is that packets can be delayed even further. In fact, if we restrict queue sizes to be $O(1)$ in the butterfly, then the greedy algorithm can be forced to use $\Theta(N)$ steps to route some permutations. (See Problem 3.185.)

For small values of N, the worst-case performance of the greedy routing algorithm is not so bad. This is because $\sqrt{N}$ and $\log N$ are not all that different when N is small (say, less than 100). For large N, the worst-case performance of the greedy algorithm becomes more of a problem, however, particularly since so many of the natural permutations (such as bit-reversal and transpose) exhibit worst-case performance for the greedy algorithm.

Of course, we know from Theorem 3.11 that every permutation can be routed in $2 \log N$ steps on the butterfly with queues of size 1, provided that we are allowed to use off-line precomputation and that we can make two passes through the butterfly. Hence, it makes sense to use a special set of precomputed routing paths (instead of the greedy algorithm) whenever we encounter one of the known worst-case permutations. As a consequence, we really don't have to worry about the worst-case performance of the

bit-reversal and transpose permutations since we don't have to route them using the greedy algorithm.

Unfortunately, there are many bad permutations for the greedy algorithm, and it is not feasible to precompute special routing paths for all of them using Theorem 3.11. In an attempt to overcome this problem, special-purpose routing algorithms have been developed that work well for large classes of permutations that are not handled efficiently by the greedy algorithm. For example, see Problems 3.188–3.191. While such special-purpose algorithms can efficiently handle several natural permutations such as the transpose and bit-reversal permutations, they are still not sufficient to efficiently handle all of the permutations for which the greedy algorithm performs poorly. Indeed, we will have to cover a lot more material before we are ready to describe routing algorithms that perform well for all permutations.

In the preceding discussion, we concentrated on the problem of routing packets from one end of the butterfly to the other. (Such routing problems are sometimes called *end-to-end* routing problems.) In practice, the butterfly is often used to route packets in exactly this fashion. It is also sometimes used to route packets between all of the nodes of the network. When each node of the $\log N$-dimensional wrapped butterfly starts and finishes with one packet, and each of the $\log N$ packets is greedily routed (in the same direction) first to the correct row and then to the correct node, then each of the $N \log N$ packets will be routed to the correct destination within $\Theta(\sqrt{N \log N})$ steps in the worst case. (See Problems 3.192–3.193.)

For simplicity, we will continue to focus our study of hypercubic routing algorithms on the problem of routing packets from one end of the butterfly to the other. The results that we obtain for this particular problem can usually be extended to hold for most other hypercubic routing problems of interest. For example, results for end-to-end routing on a $\log N$-dimensional butterfly can be immediately extended to hold for arbitrary routing problems (i.e., routing problems where every node may start with a packet) on an N-node hypercube, since there is such a close relationship between the edges of the $\log N$-dimensional butterfly and the edges of the N-node hypercube. Moreover, if the packets move through the hypercube in a normal fashion, then the results can also be extended to hold for arbitrary routing problems on any N-node hypercubic network.

Results for end-to-end routing on a butterfly can also be extended to hold for arbitrary butterfly routing problems by first routing each packet

to the level 0 node in its row, and then routing the packet to the level $\log N$ node in its destination row, before routing the packet to its correct destination. The hard part of the routing is the end-to-end routing, since routing packets within their rows can usually be accomplished in $O(\log N)$ additional steps. In other words, solving an arbitrary routing problem on a $\log N$-dimensional butterfly is often not much harder than solving $\log N$ end-to-end routing problems on the butterfly. In addition, by using pipelining, we will often find that solving $\log N$ end-to-end routing problems on a butterfly is not much harder than solving a single end-to-end routing problem on the butterfly.

Despite the fact that the greedy routing algorithm performs poorly in the worst case, the greedy algorithm is very useful. In fact, we will show that the greedy algorithm often performs exceptionally well. For example, for many useful classes of permutations, the greedy algorithm runs in $\log N$ steps, which is optimal. And, for most permutations, the greedy algorithm runs in $\log N + o(\log N)$ steps. (We prove these important facts in Subsections 3.4.3 and 3.4.4, respectively.) As a consequence, the greedy algorithm is widely used in practice.

In what follows, we digress briefly from our study of greedy routing algorithms on hypercubic networks in order to prove a general lower bound on the time required for any greedylike algorithm to route a worst-case permutation on an arbitrary network.

A General Lower Bound for Oblivious Routing ⋆

A routing algorithm is said to be *oblivious* if the path travelled by each packet depends only on the origin and destination of the packet (and not on the origins and destinations of the other packets nor on congestion encountered during the routing). For example, the greedy routing algorithm on the butterfly is oblivious, since each packet follows the greedy path to its destination.

In what follows, we will show that for any N-node, degree-d network and any oblivious routing algorithm, there is an N-packet one-to-one routing problem for which the algorithm will take $\Omega(\sqrt{N}/d)$ steps to complete the routing. This means that the worst-case running time of any oblivious or greedy routing algorithm on the butterfly will be $\Omega(\sqrt{N})$, a far cry from the desired bound of $O(\log N)$. In fact, this means that the worst-case running time of the greedy algorithm on any N-node bounded-degree network is $\Omega(\sqrt{N})$. For the hypercube, the worst-case bound will be $\Omega(\sqrt{N}/\log N)$.