15.2 PPR ROUTING

The problem of PPR was defined in Section 14.2 as: Each processor in the network is the origin of at most one packet and is the destination of at most one packet; send the packets to their destinations. In this section we develop PPR algorithms for $\mathcal{H}_d$.

15.2.1 A Greedy Algorithm

We consider the problem of routing on $\mathcal{B}_d$, where there is a packet at each processor of level 0. The destinations of the packets are in level d in such a way that the destination rows form a partial permutation of the origin rows. In $\mathcal{B}_3$, for instance, the origin rows could be all the rows and the destination rows could be $001, 000, 100, 111, 101, 010, 011, 110$. A greedy algorithm for routing any PPR is to let each packet use the greedy path between its origin and destination. The distance traveled by any packet is d using this algorithm. To analyze the run time of this algorithm, we only need to compute the maximum delay any packet suffers.

Let $u = \langle r, \ell \rangle$ be any processor in $\mathcal{B}_d$. Then there are $\leq 2^\ell$ packets that can potentially go through the processor u. This is because u has two neighbors in level $\ell-1$, each one of which has two neighbors in level $\ell-2$, and so on. As an example, the only packets that can go through the processor $\langle 011, 2 \rangle$ have origin rows $001, 011, 101$, and 111 (in Figure 15.3). Similarly, a packet that goes through u can reach only one of $2^{d-\ell}$ possible destinations. This implies that the maximum number of packets that can contend for any link in level ℓ is $\min \{2^{\ell-1}, 2^{d-\ell}\}$. Let π be an arbitrary packet; π can only suffer a delay of $\leq \min \{2^{\ell-1}, 2^{d-\ell}\}$ in crossing a level ℓ link and has to cross a level ℓ link for $\ell = 1, 2, \ldots, d$. Thus the maximum delay the packet can suffer is $D = \sum_{\ell=1}^{d} \min \{2^{\ell-1}, 2^{d-\ell}\}$. Assume without loss of generality that d is even. Then, D can be rewritten as $D = \sum_{\ell=1}^{d/2} 2^{\ell-1} + \sum_{\ell=d/2+1}^{d} 2^{d-\ell}$ $= 2 * 2^{d/2} - 2 = O(2^{d/2})$. The value $O(2^{d/2})$ is also an upper bound on the queue length of the algorithm, since the number of packets going through any processor in level ℓ is $\leq \min \{2^\ell, 2^{d-\ell}\}$. The maximum of this number over all ℓ's is $2^{d/2}$.

Lemma 15.5 The greedy algorithm on $\mathcal{B}_d$ runs in $O(2^{d/2})$ time, the queue length being $O(2^{d/2})$. □

15.2.2 A Randomized Algorithm

We can improve the performance of the preceding greedy algorithm drastically using randomization. Recall that in the case of routing on the mesh, we were able to reduce the queue length of the greedy routing algorithm

with the introduction of an additional phase where the packets are sent to random intermediate processors. The reason for sending packets to random processors is that with high probability a packet does not get to meet many other packets and hence the number of possible link contentions decreases. A similar strategy can be applied on the butterfly also.

The routing problem considered is the same as before; that is, there is a packet at each processor of level zero and the packets have destinations in level d. There are three phases in the algorithm. In the first phase each packet chooses a random intermediate destination in level d and goes there using the greedy path. In the second phase it goes to its actual destination row but in level zero. Finally, in the third phase, the packets go to their actual destinations in level d. In the third phase, each packet has to travel to level d using the direct link at each level. This takes d steps. Figure 15.8 illustrates these three phases. In this figure, r is a random node in level d. The variables u and v are the origin and destination of the packet under concern. The second phase is the reverse of phase 1, and hence it suffices to compute the run time of phase 1 to calculate the run time of the whole algorithm. The following lemma proves helpful in the analysis of phase 1.

Lemma 15.6 [Queueline lemma] Let $\mathcal{P}$ be the collection of paths to be taken by packets in a network. If the paths in $\mathcal{P}$ are *nonrepeating*, then the delay suffered by any packet π is no more than the number of distinct packets that *overlap* with π. A set of paths $\mathcal{P}$ is said to be *nonrepeating* if any two paths in $\mathcal{P}$ that meet, share some successive links, and diverge never meet again. For example, the greedy paths in $\mathcal{B}_d$ are nonrepeating. Two packets are said to *overlap* if they share at least one link in their paths.

Proof: Let π be an arbitrary packet. If π is delayed by each of the packets that overlap with π no more than once, the lemma is proven. Else, if a packet (call it q) overlapping with π delays π twice (say), then q has been delayed by another packet which also overlaps with π and which never gets to delay π. □

Analysis of phase 1

Let π be an arbitrary packet. Also let e_i be the link that π traverses in level i, for $1 \leq i \leq d$. To compute the maximum delay that π can ever suffer, it suffices to compute the number of distinct packets that overlap with π (c.f. the queueline lemma). If n_i is the number of packets that have the link e_i in their paths, then $D = \sum_{i=1}^{d} n_i$ is an upper bound on the number of packets that overlap with π.

Consider the link e_i. The number of packets that can potentially go through this link is 2^{i-1} since there are only 2^{i-1} processors at level zero for which there are greedy paths through the link e_i. Each such packet has a probability of $\frac{1}{2^i}$ of going through e_i. This is because a packet starting at level zero can take either the direct link or the cross link, each with a

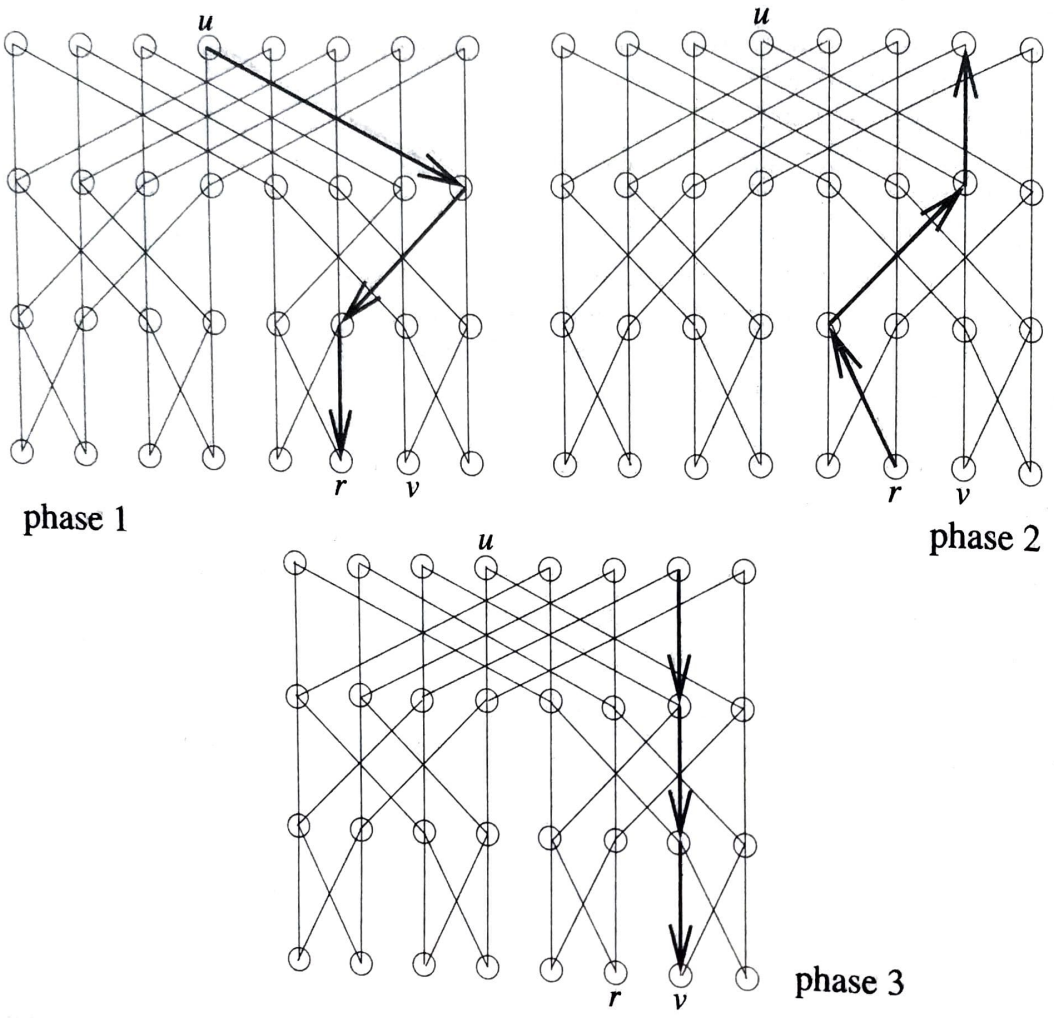

Figure 15.8 Three phases of randomized routing

probability of $\frac{1}{2}$. Once it reaches a processor at level one, it can again take either a cross link or a direct link with probability $\frac{1}{2}$. And so on. If the packet has to go through e_i, it should pick the right link at each level and there are i such links.

Therefore, the number n_i of packets that go though e_i is a binomial $B(2^{i-1}, \frac{1}{2^i})$. The expected value of this is $\frac{1}{2}$. Since the expectation of a sum is the sum of expectations, the expected value of $\sum_{i=1}^{d} n_i$ is $\frac{d}{2}$. Now we show that the total delay is $O(d)$ with high probability. The variable D is upper

bounded by the binomial $B(d, \frac{1}{2})$. Using Chernoff bounds Equation 1.1,

$$\begin{aligned} \text{Prob.}[D > e\alpha d] &\leq \left(\frac{d/2}{e\alpha d}\right)^{e\alpha d} e^{e\alpha d - d/2} < \left(\frac{1}{2e\alpha}\right)^{e\alpha d} e^{e\alpha d} \\ &\leq \left(\frac{1}{2\alpha}\right)^{e\alpha d} < 2^{-e\alpha d} < p^{-\alpha-1} \end{aligned}$$

Here $\alpha \geq 1$ and we have made use of the fact that $d = \Theta(\log p)$. Since there are $< p$ packets, the probability that at least one of the packets has a delay of more than $2e\alpha d$ is $< p^{-\alpha-1}p = p^{-\alpha}$. We arrive at the following theorem.

Theorem 15.1 The randomized algorithm for routing on $\mathcal{B}_d$ runs in time $\tilde{O}(d)$. □

Since the diameter of any network is a lower bound on the worst-case time for PPR in any network, the above algorithm is asymptotically optimal.

Queue length analysis

The queue length of the preceding algorithm is also $\tilde{O}(d)$. Let v_i be any processor in level i (for $1 \leq i \leq d$). The number of packets that can potentially go though this processor is 2^i. Each such packet has a probability of $\frac{1}{2^i}$ of going through v_i. Thus the expected number of packets going through v_i is $2^i \frac{1}{2^i} = 1$. Using Chernoff bounds Equation 1.1, the number of packets going through v_i can be shown to be $\tilde{O}(d)$.

Theorem 15.1 together with Lemma 15.1 yields Theorem 15.2.

Theorem 15.2 Any PPR can be routed on a parallel $\mathcal{H}_d$ in $\tilde{O}(d)$ time, the queue length being $\tilde{O}(d)$. □

EXERCISES

1. Lemma 15.5 proves an upper bound on the run time of the greedy algorithm on $\mathcal{B}_d$. Prove a matching lower bound. (*Hint:* Consider the bit reversal permutation. In this permutation if $b_1 b_2 \cdots b_d$ is the origin row of any packet, its destination row is $b_d b_{d-1} \cdots b_2 b_1$. For this permutation, compute the traffic through any level $\frac{d}{2}$ link.)

2. Assume that d packets originate from every processor of level zero in a $\mathcal{B}_d$. These packets are destined for level d with d packets per processor. Analyze the run time and queue length of the randomized routing algorithm on this problem.

3. If q packets originate from every processor of level zero and each packet has a random destination in level d of a $\mathcal{B}_d$, present a routing algorithm that runs in time $\tilde{O}(q+d)$.

4. In a $\mathcal{B}_d$, at most one packet is destined for any processor in level d. The packet (if any) destined for processor i is at the beginning placed randomly in one of the processors of level zero (each such processor being equally likely). There are only a total of $(2^d)^\epsilon$ packets, for some constant $\epsilon > 0$. If the greedy algorithm is used to route, what is the worst-case run time? What is the queue length?

5. For the routing problem of Exercise 4, what is the run time and queue length if the randomized routing algorithm of Section 15.2.2 is employed?

15.3 FUNDAMENTAL ALGORITHMS

In this section we present hypercube algorithms for such basic operations as broadcasting, prefix sums computation, and data concentration. All these algorithms take $O(d)$ time on $\mathcal{H}_d$. Since the diameter is a lower bound on the solution time of any nontrivial problem in an interconnection network, these algorithms are asymptotically optimal.

15.3.1 Broadcasting

The problem of broadcasting in an interconnection network is to send a copy of a message that originates from a particular processor to a subset of other processors. Broadcasting is quite useful since it is widely used in the design of several algorithms. To perform broadcasting on $\mathcal{H}_d$, we employ the binary tree embedding (see Figure 15.7). Assume that the message M to be broadcast is at the root of the tree (i.e., at the processor $00\cdots0$). The root makes two copies of M and sends a copy to each of its two children in the tree. Each internal processor, on receipt of a message from its parent, makes two copies and sends a copy to each of its children. This proceeds until all the leaves have a copy of M. Note that the height of this tree is d. Thus in $O(d)$ steps, each leaf processor has a copy of M.

In this algorithm, computation happens only at one level of the tree at any given time. Thus each step of this algorithm can be run in one time unit on the sequential $\mathcal{H}_d$.

Lemma 15.7 Broadcasting of a message can be done on the sequential $\mathcal{H}_d$ in $\Theta(d)$ time. □

Example 15.7 Steps involved in broadcasting on a $\mathcal{H}_2$ are shown in Figure 15.9. The algorithm completes in two steps. □

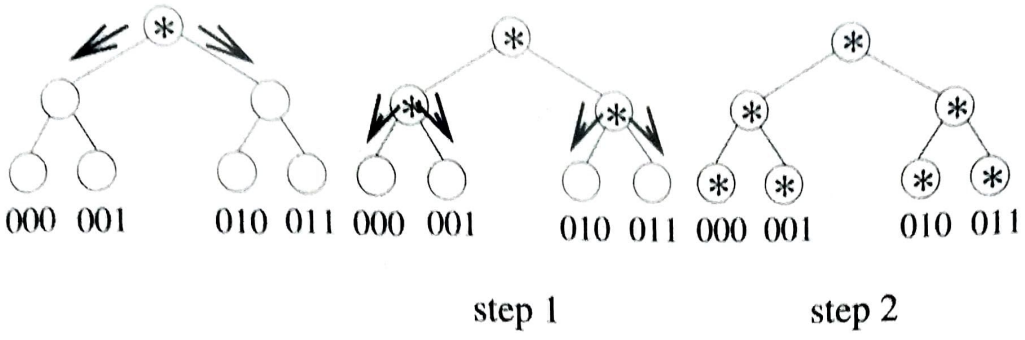

Figure 15.9 Broadcasting on a $\mathcal{H}_2$

15.3.2 Prefix Computation

We again make use of the binary tree embedding to perform prefix computation on $\mathcal{H}_d$. Let x_i be input at the ith leaf of a 2^d-leaf binary tree. There are two phases in the algorithm, namely, the *forward phase* and the *reverse phase*. In the forward (reverse) phase, data items flow from bottom to top (top to bottom). In each step of the algorithm only one level of the tree is active. Algorithm 15.1 gives the algorithm.

Example 15.8 Let Σ be the set of all integers and $\oplus$ be the usual addition. Consider a four-leaf binary tree with the following input: $5, 8, 1, 3$. Figure 15.10 shows the execution of every step of Algorithm 15.1. The datum inside each internal processor is its y-value. In step 1, the leaves send their data up (Figure 15.10(a)). In step 2, the internal processors send 13 and 4, respectively, storing 5 and 1 (Figure 15.10(b)) as their y-values. In step 3, the root sends 0 to the left and 13 to the right (Figure 15.10(c)). In the next step, the leftmost internal processor sends 0 to the left and 5 to the right. The rightmost internal processor sends 13 to the left and 14 to the right (Figure 15.10(d)). In step 5, the prefixes are computed at the leaves. □

In the forward phase of Algorithm 15.1, each internal processor computes the sum of all the data in its subtree. Let v be any internal processor and v' be the leftmost leaf in the subtree rooted at v. Then, in the reverse phase of the algorithm, the datum q received by v can be seen to be $\sum_{i=0}^{v'-1} x_i$. That is, q is the sum of all input data items to the left of v'. The correctness of the algorithm follows. Also both the forward phase and the reverse phase take d steps each. Moreover, at any given time unit, only one level of the tree is active. Thus each step of Algorithm 15.1 can be simulated in one step on $\mathcal{H}_d$.

Forward phase

The leaves start by sending their data up to their parents. Each internal processor on receipt of two items (say y from its left child and z from its right child) computes $w = y \oplus z$, stores a copy of y and w, and sends w to its parent. At the end of d steps, each processor in the tree has stored in its memory the sum of all the data items in the subtree rooted at this processor. In particular, the root has the sum of all the elements in the tree.

Reverse phase

The root starts by sending zero to its left child and its y to its right child. Each internal processor on receipt of a datum (say q) from its parent sends q to its left child and $q \oplus y$ to its right child. When the ith leaf gets a datum q from its parent, it computes $q \oplus x_i$ and stores it as the final result.

Program 15.1 Prefix computation on a binary tree

Lemma 15.8 Prefix computation on a 2^d-leaf binary tree as well as $\mathcal{H}_d$ can be performed in $\Theta(d)$ time steps. □

Note: The problem of *data sum* is to compute $x_1 \oplus x_2 \oplus \cdots \oplus x_n$, given the x_i's. The forward phase of Algorithm 15.1 suffices to compute the data sum. Thus the time to compute the data sum is only one-half the time taken to compute all the prefixes.

15.3.3 Data Concentration

On $\mathcal{H}_d$ assume that there are $k < p$ data items distributed arbitrarily with at most one datum per processor. The problem of data concentration is to move the data into the processors $0, 1, \ldots, k-1$ of $\mathcal{H}_d$ one data item per processor. If we can compute the final destination address for each data item, then the randomized routing algorithm of Section 15.2 can be employed to route the packets in time $\tilde{O}(d)$. Note that the randomized routing algorithm assumes the parallel hypercube.

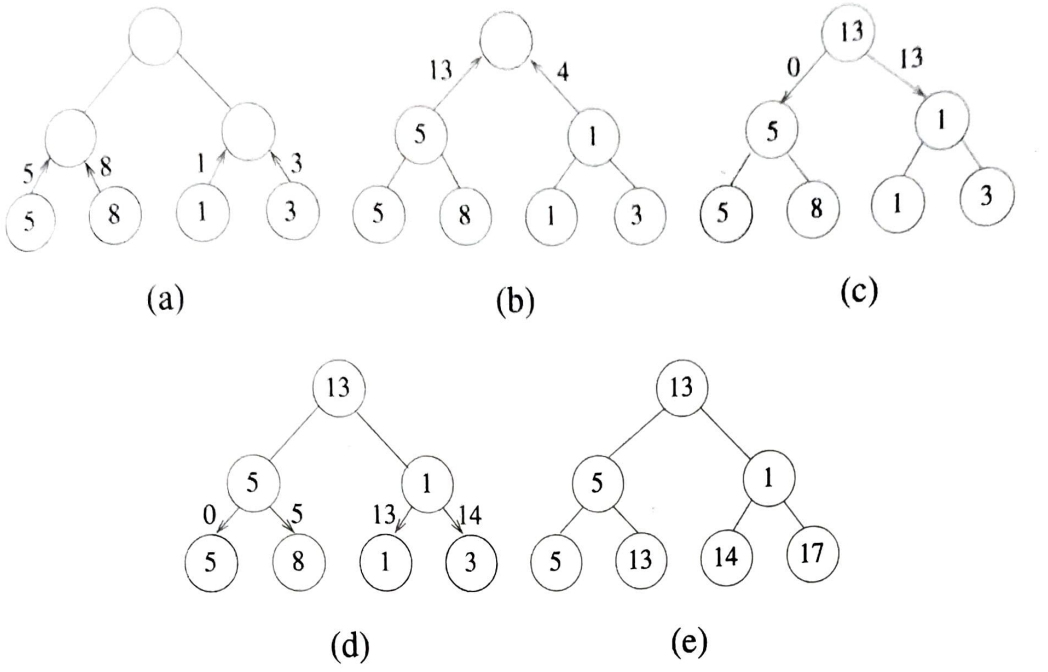

Figure 15.10 Prefix computation on a binary tree

There is a much simpler deterministic algorithm that runs in the same asymptotic time on the sequential hypercube. In fact we present a normal butterfly algorithm with the same run time and then invoke Lemma 15.2. We list some properties of the butterfly network that are needed in the analysis of the algorithm.

Property 1 If the level d processors and incident links are eliminated from $\mathcal{B}_d$, two copies of $\mathcal{B}_{d-1}$ result. As an example, in Figure 15.11, removal of level 3 processors and links results in two independent $\mathcal{B}_2$'s. One of these butterflies consists of only even rows (shown with thick lines) and the other consists of only odd rows. Call the former the even subbutterfly and the latter the odd subbutterfly.

Property 2 All processors at level d are connected by a full binary tree. For example, if we trace all the descendants of the processor $00\cdots0$ of level zero, the result is a full binary tree with the processors of level d as its leaves. In fact this is true for each processor at level zero.

Now we are ready to describe the algorithm for data concentration. Assume that the $k \leq 2^d$ data items are arbitrarily distributed in level d of $\mathcal{B}_d$.

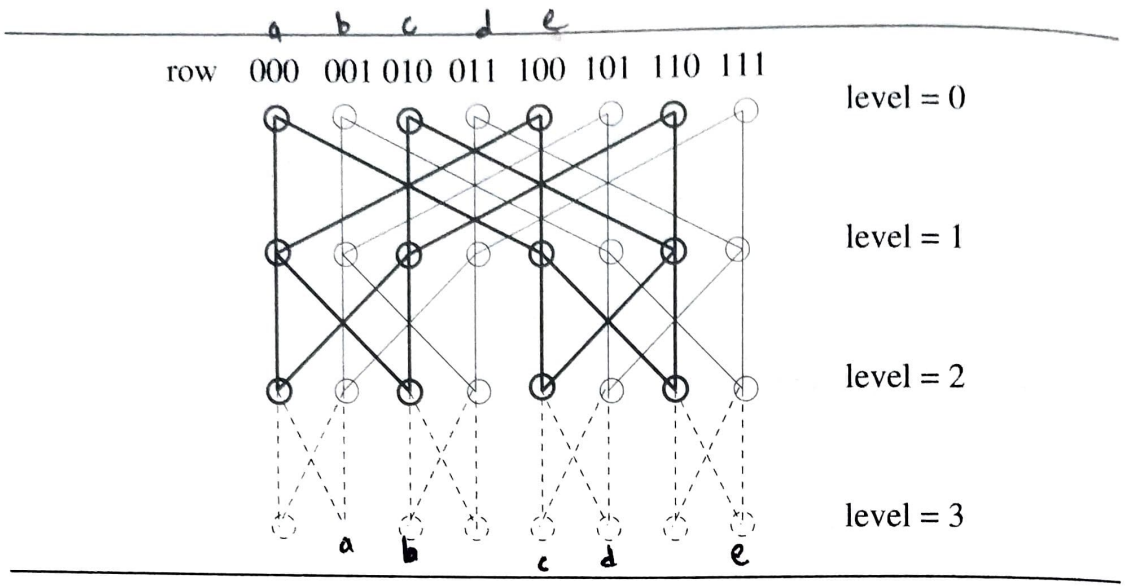

Figure 15.11 Removal of level d processors and links

At the end, these data items have to be moved to successive rows of level zero. For example, if there are five items in level 3 of $\mathcal{B}_3$, row 001 $\rightarrow a$ (this notation means that the processor $\langle 001, 3\rangle$ has the item a), row 010 $\rightarrow b$, row 100 $\rightarrow c$, row 101 $\rightarrow d$, and row 111 $\rightarrow e$, then at the end, these items will be at level zero and row 000 $\rightarrow a$, row 001 $\rightarrow b$, row 010 $\rightarrow c$, row 011 $\rightarrow d$, and row 100 $\rightarrow e$. There are two phases in the algorithm. In the first phase a prefix sums operation is performed to compute the destination address of each data item. In the second phase each packet is routed to its destination using the greedy path from its origin to its destination.

The prefix computation can be done using any of the trees mentioned in property 2 and Lemma 15.8. The prefix sums are computed on a sequence, $x_0, x_1, \ldots, x_{2^d-1}$, of zeros and ones. Leaf i sets x_i to one if it has a datum, otherwise to zero. In accordance with Lemma 15.8, this phase takes $O(d)$ time.

In the second phase packets are routed using the greedy paths. The claim is that no packet gets to meet any other and hence there is no possibility of link contentions. Consider the first step in which the packets travel from level d to level $d-1$. If two packets meet at level $d-1$, it could be only because they originated from two successive processors of level d. If two packets originate from two successive processors, then they are also destined for two successive processors. In particular, one has an odd row as its destination and the other has an even row. That is, one belongs to the odd subbutterfly and the other belongs to the even subbutterfly (see Figure 15.11). Without loss of generality assume that the packets that meet at level $d-1$ meet at a processor of the odd subbutterfly. Then it is impossible for one of these two

to reach any processor of the even subbutterfly. In summary, no two packets can meet at level $d-1$.

After the first step, the problem of concentration reduces to two subproblems: concentrating the packets in the odd subbutterfly and concentrating the items on the even subbutterfly. But these subbutterflies are of dimension $d-1$. Thus by induction it follows that there is no possibility of any two packets' meeting in the whole algorithm.

The first phase as well as the second phase of this algorithm takes $\Theta(d)$ time each. Also note that the whole algorithm is normal. We get this lemma.

Lemma 15.9 Data concentration can be performed on $\mathcal{B}_d$ as well as the sequential $\mathcal{H}_d$ in $\Theta(d)$ time. □

Definition 15.4 The problem of *data spreading* is there are $k \leq 2^d$ items in successive processors at level zero of $\mathcal{B}_d$ (starting from row zero). The problem is to route them to some k specified processors at level d (one item per processor). The destinations can be arbitrary except that the order of the items must be preserved (that is, the packet originating from row zero must be the leftmost packet at level d, the packet originating from row one must be the next packet at level d, and so on). □

Definition 15.5 The problem of *monotone routing* is there are $k \leq 2^d$ packets arbitrarily distributed, at most one per processor, at level d of $\mathcal{B}_d$. They are destined for some k arbitrary processors at level zero such that the order of packets is preserved. □

Data spreading is just the reverse of data concentration. Also, monotone routing can be performed by performing a data concentration followed by a data spreading. Thus each can be done in time $\Theta(d)$.

Lemma 15.10 Data spreading as well as monotone routing takes $\Theta(d)$ time on $\mathcal{B}_d$ and the sequential $\mathcal{H}_d$. □

15.3.4 Sparse Enumeration Sort

The problem of sparse enumeration sort was introduced in Section 14.3.4. Let the sequence to be sorted on $\mathcal{H}_d$ be $X = k_0, k_2, \ldots, k_{\sqrt{p}-1}$, where $p = 2^d$. Without loss of generality assume that d is even. Let the input be given one key per processor in the subcube defined by fixing the first $\frac{d}{2}$ bits of a d-bit number to zeros (and varying the other bits). The sorted output also should appear in this subcube one key per processor.

Let v be any processor of $\mathcal{H}_d$. Its label is a d-bit binary number. The same label can be thought of as a tuple $\langle i, j \rangle$, where i is the first $\frac{d}{2}$ bits and j is the next $\frac{d}{2}$ bits. All processors whose first $\frac{d}{2}$ bits are the same and equal

to i form a $\mathcal{H}_{d/2}$ (for each $0 \leq i \leq 2^{d/2} - 1$). Call this subcube row i. Also all processors of $\mathcal{H}_d$ whose last $\frac{d}{2}$ bits are the same and equal to j form a subcube $\mathcal{H}_{d/2}$. Call this subcube column j. The $\sqrt{p}$ numbers to be sorted are input in row zero. The output also should appear in row zero. To be specific, the key whose rank is r should be output at the processor $\langle 0, r-1 \rangle$ (for $1 \leq r \leq 2^{d/2}$).

To begin with, k_j is broadcast in column j (for $0 \leq j \leq \sqrt{p} - 1$) so that each row has a copy of X. In row i compute the rank of k_i. This is done by broadcasting k_i to all the processors in row i followed by a comparison of k_i with every key in the input and a data sum computation. The rank of k_i is broadcast to all the processors in row i. If the rank of k_i is r_i, then the processor $\langle i, r_i - 1 \rangle$ broadcasts k_i along the column $r_i - 1$ so that at the end of this broadcasting the processor $\langle 0, r_i - 1 \rangle$ gets the key that it is supposed to output.

The preceding algorithm is a collection of operations local to the columns or the rows. The operations involved are prefix computation, broadcast, and comparison each of which can be done in $O(d)$ time. Thus the whole algorithm runs in time $\Theta(d)$.

Lemma 15.11 Sparse enumeration sort can be completed in $\Theta(d)$ time on a sequential $\mathcal{H}_d$, where the number of keys to be sorted is at most $\sqrt{p}$. □

EXERCISES

1. The broadcasting algorithm of Section 15.3.1 assumes that the message originates from processor $00\cdots0$. How can you broadcast in $\Theta(d)$ time on $\mathcal{H}_d$ if the origin is an arbitrary processor?

2. Prove Lemma 15.10.

3. In a sequential $\mathcal{H}_d$, every processor has a packet to be sent to every other processor. Present an $O(pd)$ algorithm for this routing problem, where $p = 2^d$.

4. Present an $O(p)$-time algorithm for the problem of Exercise 3.

5. If we fix the first $d - k$ bits of a d-bit binary number and vary the last k bits, the corresponding processors form a subcube $\mathcal{H}_k$ in $\mathcal{H}_d$. There are 2^{d-k} such subcubes. There is a message in each of these subcubes. Present an algorithm for every subcube to broadcast its message locally (known as *window broadcast*). What is the run time of your algorithm?

15.4 SELECTION

Given a sequence of n keys and an integer i, $1 \leq i \leq n$, the problem of selection is to find the ith-smallest key from the sequence. We have seen sequential algorithms (Section 3.7), PRAM algorithms (Section 13.4), and mesh algorithms (Section 14.4) for selection. Like in the case of the mesh, we consider two different versions of selection on $\mathcal{H}_d$. In the first version we assume that $p = n$, p being the number of processors and n the number of input keys. In the second version we assume that $n > p$. It is necessary to handle the second version separately, since no general slow-down lemma (like the one for PRAMs) exists for $\mathcal{H}_d$.

15.4.1 A Randomized Algorithm for $n = p$ (*)

The work optimal algorithm (Algorithm 13.9) of Section 13.4.5 can be adapted to run optimally on $\mathcal{H}_d$ as well. There are $\tilde{O}(1)$ stages in this algorithm. Step 1 of Algorithm 13.9 can be implemented on $\mathcal{H}_d$ in $O(1)$ time. In steps 2 and 5 prefix computations can be done in a total of $O(d)$ time (c.f. Lemma 15.8). Concentration in steps 3 and 6 takes $O(d)$ time each (see Lemma 15.9). Also, sparse enumeration sort takes the same time in steps 3 and 6 in accordance with Lemma 15.11. Selections in steps 4 and 6 take only $O(1)$ time each since these are selections from sorted sequences. Broadcasts in steps 2, 4, and 5 take $O(d)$ time each (c.f. Lemma 15.7). We arrive at this theorem.

Theorem 15.3 Selection from $n = p$ keys can be performed in $\tilde{O}(d)$ time on $\mathcal{H}_d$. □

15.4.2 Randomized Selection for $n > p$ (*)

Now we consider the problem of selection when $n = p^c$ for some constant $c > 1$. Algorithm 13.9 can be used for this case as well with some modifications. The modifications are the same as the ones we did for the mesh (see Section 14.4.2). Each processor has $\frac{n}{p}$ keys to begin with. The condition for the **while** statement is changed to $(N > D)$ (where D is a constant). In step 1 a processor includes each one of its keys with probability $\frac{1}{N^{1-(1/3c)}}$. Thus this step now takes time $\frac{n}{p}$. Step 2 remains the same and still takes $O(d)$ time. The concentration and sparse enumeration sort of step 3 can be performed in time $O(d)$ (c.f. Lemmas 15.9 and 15.11). Step 4 takes $O(d)$ time and so do steps 5 and 6. Thus each stage takes time $O(\frac{n}{p} + d)$. There are only $\tilde{O}(\log \log p)$ stages in the algorithm. The final result is this theorem.

Theorem 15.4 Selection from out of $n = p^c$ keys can be performed on $\mathcal{H}_d$ in time $\tilde{O}\left((\frac{n}{p} + d) \log \log p\right)$. □

15.4.3 A Deterministic Algorithm for $n > p$

The deterministic mesh selection algorithm (Algorithm 14.5) can be adapted to a hypercube. The correctness of this algorithm has already been established in Section 14.4.3. We only have to compute the run time of this algorithm when implemented on a hypercube.

In step 0, the elements can be partitioned into $\log p$ parts in $\frac{n}{p} \log \log p$ time (see Section 13.4, Exercise 5) and the sorting can be done in time $O(\frac{n}{p} \log \frac{n}{p})$. Thus step 0 takes $\frac{n}{p}$ min $\{\log(n/p), \log \log p\}$ time. At the end of step 0, the keys in any processor have been partitioned into approximately $\log p$ nearly equal parts. Call each such part a block.

In step 1, we can find the median at any processor as follows. First determine the block the median is in and then perform an appropriate selection in that block (using Algorithm 3.19). The total time is $O(\frac{n}{p \log p})$.

In step 2, we can sort the medians to identify the weighted median. If $M'_1, M'_2, \ldots, M'_p$ is the sorted order of the medians, we need to identify j such that $\sum_{k=1}^{j} N'_k \geq \frac{N}{2}$ and $\sum_{k=1}^{j-1} N'_k < \frac{N}{2}$. Such a j can be computed with an additional prefix computation. Sorting can be done in $O(d^2)$ time (as is shown in Section 15.6). The prefix computation takes $O(d)$ time (see Lemma 15.8). Thus M, the weighted median, can be identified in $O(d^2)$ time.

In step 3, each processor can identify the number of remaining keys in its queue and then all the processors can perform a prefix sums computation. Therefore, this step takes $O(\frac{n}{p \log p} + d)$ time.

In step 4, the appropriate keys in any processor can be eliminated as follows. First identify the block B that M falls in. This can be done in $O(\log p)$ time. After this, compare M with the elements of block B to determine the keys to be eliminated. If $i > r_M$ ($i \leq r_M$), all blocks to the left (right) of B are eliminated en masse. The total time needed is $O(\log p + \frac{n}{p \log p})$, which is $O(\frac{n}{p \log p})$ since $n = p^c$ for some constant $c > 1$.

Step 5 takes $O(d)$ time as it involves a prefix computation and a broadcast.

The broadcasting in steps 2, 3, and 4 takes $O(d)$ time each (c.f. Lemma 15.7). Thus each run of the **while** loop takes $O(\frac{n}{p \log p} + d^2)$ time. (Note that $d = \log p$.) The **while** loop is executed $O(\log n)$ times (see the proof of Theorem 14.8). Thus we get the following theorem (assuming that $n = p^c$ and hence $\log n$ is asymptotically the same as $\log p$).

Theorem 15.5 Selection on $\mathcal{H}_d$ can be performed in time $O(\frac{n}{p} \log \log p + d^2 \log n)$. □

Example 15.9 On a $\mathcal{H}_3$, let each processor have five keys to begin with. Consider the selection problem in which $i = 32$. The input keys are shown

in Figure 15.12. For simplicity neglect step 0 of Algorithm 14.3; that is, assume that the parts are of size 1.

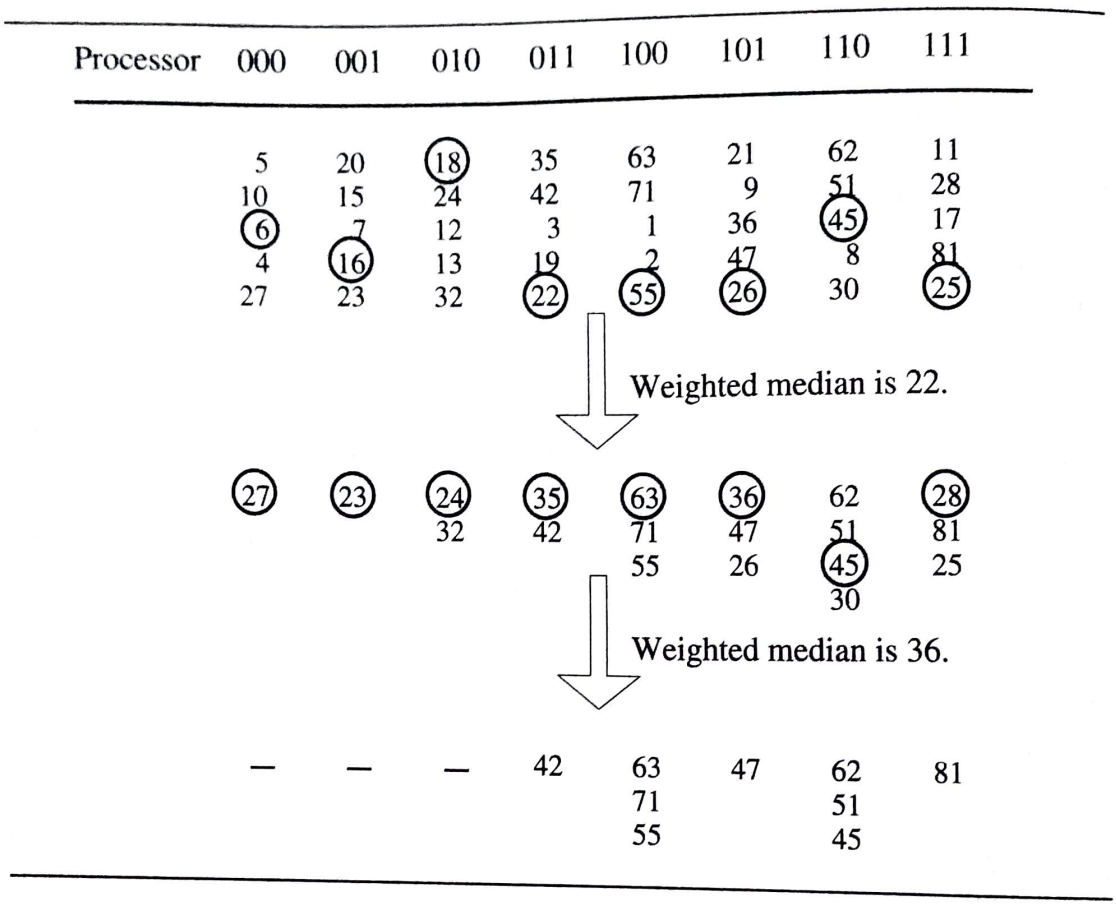

Figure 15.12 Deterministic selection

In step 1, local medians are found. These medians are circled in the figure. The sorted order of these medians is $6, 16, 18, 22, 25, 26, 45, 55$. Since at the beginning each processor has the same number of keys, the weighted median is the same as the regular median. A median M of these medians is 22. In step 3, the rank of M is determined as 21. Since $i > 21$, all the keys that are less than or equal to 22 are eliminated. We update i to $32 - 21 = 11$. This completes one run of the **while** loop.

In the second run of the **while** loop, there are 19 keys to begin with. The local medians are $27, 23, 24, 35, 63, 36, 45, 28$ with corresponding weights of $1, 1, 2, 2, 3, 3, 4, 3$, respectively. The sorted order of these medians is $23, 24, 27, 28, 35, 36, 45, 63$ with corresponding weights of $1, 2, 1, 3, 2, 3, 4, 3$, respectively. The weighted median M is 36. Its rank is 10. Thus all the keys that are less than or equal to 36 are eliminated. We update i to $11 - 10 = 1$, and this completes the second run of the **while** loop.

The rest of the computation proceeds similarly to finally output 42 as the answer. □

EXERCISES

1. Complete Example 15.9.
2. Present an efficient algorithm for finding the kth quantiles of any given sequence of n keys on $\mathcal{H}_d$. Consider the cases $n = p$ and $n > p$.
3. Given an array A of n elements, the problem is to find any element of A that is greater than or equal to the median. Present a simple Monte Carlo algorithm for this problem on $\mathcal{H}_d$. You cannot use the selection algorithm of this section. Your algorithm should run in time $O(d)$. Show that the output of your algorithm is correct with high probability. Assume that $p = n$.

15.5 MERGING

The problem of merging is to take two sorted sequences as input and produce a sorted sequence of all the elements. This problem was studied in Chapters 3, 10, 13, and 14. If the two sequences to be merged are of length m each, they can be merged in $O(\log m)$ time using a hypercube with $O(m^2)$ processors (applying the sparse enumeration sort). In this section we are interested in merging on a hypercube with only $2m$ processors (assuming that m is an integral power of 2). The technique employed is the odd-even merge.

15.5.1 Odd-Even Merge

Let $X_1 = k_0, k_1, \ldots, k_{m-1}$ and $X_2 = k_m, k_{m+1}, \ldots, k_{2m-1}$ be the two sorted sequences to be merged, where $2m = 2^d$. We show that there exists a normal butterfly algorithm that can merge X_1 and X_2 in $O(d)$ time.

We use a slightly different version of the odd-even merge. First we separate the odd and even parts of X_1 and X_2. Let them be O_1, E_1, O_2, and E_2. Then we recursively merge E_1 with O_2 to obtain $A = a_0, a_1, \ldots, a_{m-1}$. Also we recursively merge O_1 with E_2 to obtain $B = b_0, b_1, \ldots, b_{m-1}$. After this, A and B are shuffled to form $C = a_0, b_0, a_1, b_1, \ldots, a_{m-1}, b_{m-1}$. Now we compare a_i with b_i (for $0 \leq i \leq m-1$) and interchange them if they are out of order. The resultant sequence is in sorted order. The correctness of this algorithm can be established using the zero-one principle and is left as an exercise.

Example 15.10 Let $X_1 = 8, 12, 25, 31$ and $X_2 = 3, 5, 28, 46$. For this case, $O_1 = 12, 31$ and $E_1 = 8, 25$. $O_2 = 5, 46$ and $E_2 = 3, 28$. Merging E_1 with O_2

we get $A = 5, 8, 25, 46$. Similarly we get $B = 3, 12, 28, 31$. Shuffling A with B gives $C = 5, 3, 8, 12, 25, 28, 46, 31$. Next we interchange 5 and 3, and 46 and 31 to get $3, 5, 8, 12, 25, 28, 31, 46$. □

It turns out that the modified algorithm is very easy to implement on $\mathcal{B}_d$. For example, partitioning X_1 and X_2 into their odd and even parts can be easily done in one step on a $\mathcal{B}_d$. The shuffling operation can also be performed easily on a $\mathcal{B}_d$. On a $\mathcal{B}_d$, assume that both X_1 and X_2 are input in level d. Let X_1 be input in the first m rows (i.e., rows $0, 1, 2, \ldots, m-1$) and X_2 in the next m rows. The first step of the algorithm is to separate X_1 and X_2 into their odd and even parts. After this, we recursively merge E_1 with O_2, and O_1 with E_2. To do this, route the keys in the first m rows using direct links and route the other keys using cross links (see Figure 15.13).

After this routing we have E_1 and O_2 in the even subbutterfly and O_1 and E_2 in the odd subbutterfly. In particular, E_1 is in the first half of the rows of the even subbutterfly, O_2 is in the second half of the rows of the even subbutterfly, and so on (see Figure 15.13(a)). The parts E_1 and O_2 are recursively merged in the even subbutterfly. At the same time, O_1 and E_2 are recursively merged in the odd subbutterfly. Once the recursive calls are over, A will be ready in the even subbutterfly (at level $d-1$) and B will be ready in the odd subbutterfly (see Figure 15.13(b)). What remains to be done is a shuffle and a compare-exchange. They can be done as follows. Each processor at level $d-1$ sends its result along the cross link as well as the direct link. When the processor in row i at level d receives two data from above, it keeps the minimum of the two if i is even; otherwise it keeps the maximum. For example, the processor $\langle 0, d \rangle$ keeps the minimum of a_0 and b_0. The processor $\langle 1, d \rangle$ keeps the maximum of a_0 and b_0. And so on.

If $T(\ell)$ is the run time of the above algorithm on a butterfly of dimension ℓ, then the time needed to partition X_1 and X_2 is $O(1)$. The time taken to recursively merge E_1 with O_2 and O_1 with E_2 is $T(\ell-1)$ since these mergings happen in the even and odd subbutterflies which are of dimension one less than ℓ. Once the recursive merges are ready, shuffling A and B and performing the compare-exchange operation also take a total of $O(1)$ time. So, $T(\ell)$ satisfies $T(\ell) = T(\ell-1) + O(1)$, which solves to $T(\ell) = O(\ell)$.

There are two phases in the overall algorithm. In the first phase data flow from bottom to top and in the second phase data flow from top to bottom. In the first phase, when any data item progresses toward level zero, it enters subbutterflies of smaller and smaller dimensions. In particular, if it is in level ℓ, it is in a $\mathcal{B}_\ell$. In the $\mathcal{B}_\ell$ the datum is in, if it is in the first half of the rows, it takes the direct link; otherwise it takes the cross link. When all the data reach level zero, the first phase is complete. In the second phase, data flow from top to bottom one level at a time. When the data are at level ℓ, each processor at level ℓ sends its datum both along the direct link and along the cross link. In the next time step, processors in level $\ell+1$ keep either the minimum or the maximum of the two items received from above

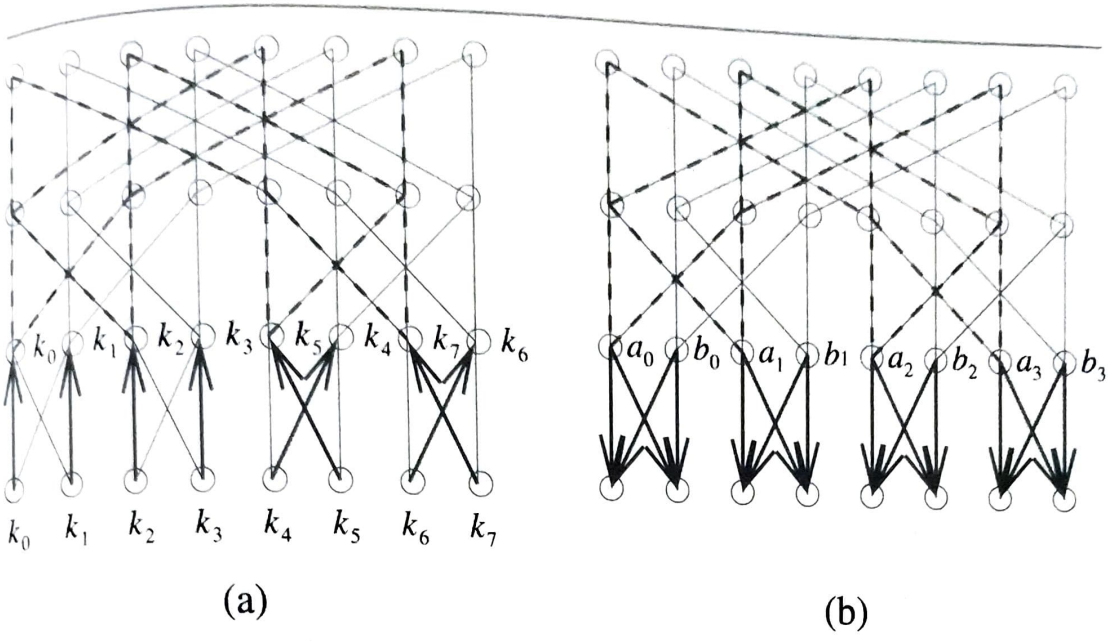

Figure 15.13 Odd-even merge on the butterfly

depending on whether the processor is in an even row or an odd row. When the data reach level d, the final result is computed. This also verifies that the algorithm takes only time $O(d)$ on any $\mathcal{B}_d$. Note also that this algorithm is indeed normal.

Theorem 15.6 Two sorted sequences of length m each can be merged on a $\mathcal{B}_d$ in $O(d)$ time, given that $2m = 2^d$. Using Lemma 15.2, merging can also be done on a sequential $\mathcal{H}_d$ in $O(d)$ time. □

15.5.2 Bitonic Merge

In Section 13.5, Exercise 2, the notion of a bitonic sequence was introduced. To recall, a sequence $K = k_0, k_1, \ldots, k_{n-1}$ is said to be *bitonic* either (1) if there is a $0 \leq j \leq n-1$ such that $k_0 \leq k_1 \leq \cdots \leq k_j \geq k_{j+1} \geq \cdots \geq k_{n-1}$ or (2) if a cyclic shift of K satisfies condition 1. If K is a bitonic sequence with n elements (for n even), let $a_i = \min\{k_i, k_{i+n/2}\}$ and $b_i = \max\{k_i, k_{i+n/2}\}$, $0 \leq i \leq n/2 - 1$. Also let $L(K) = a_0, a_1, \ldots, a_{n/2-1}$ and $H(K) = b_0, b_1, \ldots, b_{n/2-1}$. A bitonic sequence has the following properties (which you have already proved):

1. $L(K)$ and $H(K)$ are both bitonic.
2. Every element of $L(K)$ is smaller than any element of $H(K)$. In other

words, to sort K, it suffices to sort $L(K)$ and $H(K)$ separately and output one followed by the other.

Given two sorted sequences of m elements each, we can form a bitonic sequence out of them by following one by the other in reverse order. For example, if we have the sequences $5, 12, 16, 22$ and $6, 14, 18, 32$, the bitonic sequence $5, 12, 16, 22, 32, 18, 14, 6$ can be formed. If we have an algorithm that takes as input a bitonic sequence and sorts this sequence, then that algorithm can be used to merge two sorted sequences. The resultant algorithm is called *bitonic merge*.

We show how to sort a bitonic sequence on a butterfly using a normal algorithm. Let the bitonic sequence $X = k_0, k_1, \ldots, k_{n-1}$ with $n = 2^d$ be input at level zero of $\mathcal{B}_d$. We make use of the fact that if we remove all the zero level processors and incident links of a $\mathcal{B}_d$, then two copies of $\mathcal{B}_{d-1}$ result (see Figure 15.14). Call the subbutterfly with rows $0, 1, \ldots, 2^{d-1} - 1$ the *left subbutterfly* (shown with dotted thick lines in the figure) and the other subbutterfly the *right subbutterfly*.

In the first step, each level zero processor sends its key along the direct link and the cross link as shown in Figure 15.14. A level one processor, on receipt of two keys from above, keeps the minimum of the two if it is in the left subbutterfly. Otherwise it keeps the maximum. At the end of this step, the left subbutterfly has $L(K)$ and the right subbutterfly has $H(K)$. The $L(K)$ and $H(K)$ are recursively sorted in the left and right subbutterflies, respectively. Once the recursive calls are complete, the sorted sequence is at level d.

If $T(\ell)$ is the run time of this algorithm, we have $T(\ell) = T(\ell - 1) + O(1)$; that is, $T(\ell) = O(\ell)$.

Theorem 15.7 A bitonic sequence of length 2^d can be sorted on a $\mathcal{B}_d$ in $O(d)$ time. In accordance with Lemma 15.2, sorting can also be done on a sequential $\mathcal{H}_d$ in $O(d)$ time. □

EXERCISE

1. Prove the correctness of the modified version of odd-even merge using the zero-one principle.

15.6 SORTING

15.6.1 Odd-Even Merge Sort

The first algorithm is an implementation of the odd-even merge sort. If $X = k_0, k_1, \ldots, k_{n-1}$ is the given sequence of n keys, odd-even merge sort partitions X into two subsequences $X'_1 = k_0, k_1, \ldots, k_{n/2-1}$ and $X'_2 = k_{n/2}, k_{n/2+1}$,

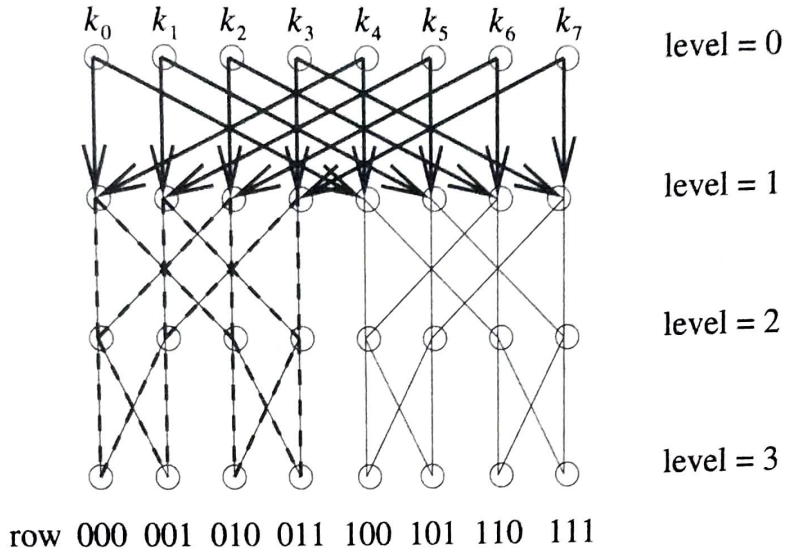

Figure 15.14 Bitonic merge on $\mathcal{B}_d$

$\ldots, k_{n-1}$ of equal length. The subsequences X_1' and X_2' are sorted recursively assigning $n/2$ processors to each. The two sorted subsequences (call them X_1 and X_2, respectively) are then finally merged using the odd-even merge algorithm.

Given 2^d keys on level d of $\mathcal{B}_d$ (one key per processor), we can partition them into two equal parts of which the first part is in rows $0, 1, \ldots, 2^{d-1} - 1$ and the second part is in the remaining rows. Sort each part recursively. Specifically, sort the first part using the left subbutterfly and at the same time sort the second part using the right subbutterfly. At the end of sorting, the sorted sequences appear in level d. Now merge them using the odd-even merge algorithm (Theorem 15.6).

If $S(\ell)$ is the time needed to sort on a $\mathcal{B}_\ell$ using the above divide-and-conquer algorithm, then we have

$$S(\ell) = S(\ell - 1) + O(\ell)$$

which solves to $S(\ell) = O(\ell^2)$.

Theorem 15.8 We can sort $p = 2^d$ elements in $O(d^2)$ time on a $\mathcal{B}_d$. As a consequence, the same can be done in $O(d^2)$ time on a sequential $\mathcal{H}_d$ as well (c.f. Lemma 15.2). □

15.6.2 Bitonic Sort

The idea of merge sort can also be applied in conjunction with the bitonic merge algorithm (Theorem 15.7). In this case, we have p numbers input at level zero of $\mathcal{B}_d$. We send the data to level $d-1$, so that the first half of the input is in the left subbutterfly and the next half is in the right subbutterfly. The left half of the input is sorted recursively using the left subbutterfly in increasing order. At the same time the right half of the input is sorted using the right subbutterfly in decreasing order. The sorted sequences are available at level $d-1$. They are now sent back to level d, so that at level d we have a bitonic sequence. This sequence is then sorted using the algorithm of Theorem 15.7.

Again, if $S(\ell)$ is the time needed to sort on a $\mathcal{B}_\ell$ using the above bitonic sort method, then we have

$$S(\ell) = S(\ell - 1) + O(\ell)$$

which solves to $S(\ell) = O(\ell^2)$.

Theorem 15.9 We can sort $p = 2^d$ elements in $O(d^2)$ time on a $\mathcal{B}_d$ using bitonic sort. As a result, applying Lemma 15.2, sorting can also be done in $O(d^2)$ time on a sequential $\mathcal{H}_d$ using bitonic sort. □

EXERCISES

1. Each processor of a sequential $\mathcal{H}_d$ is the origin of exactly one packet and each processor is the destination of exactly one packet. Present an $O(d^2)$ time $O(1)$-queue-sized deterministic algorithm for this routing problem.

2. Making use of the idea of Exercise 1, devise an $O(d^2)$ time $O(1)$-queue-length deterministic algorithm for the PPR problem.

3. You are given 2^d k-bit keys. Present an $O(kd)$ time algorithm to sort these keys on a $\mathcal{B}_d$.

4. Array A is an almost-sorted array of $p = 2^d$ elements. It is given that the position of each key is at most a distance q away from its final sorted position. How fast can you sort this sequence on the sequential $\mathcal{H}_d$ and the $\mathcal{B}_d$? Express the run time of your algorithm as a function of d and q. Prove the correctness of your algorithm using the zero-one principle.