14.3.3 Data Concentration

In a p-processor interconnection network assume that there are $d < p$ data items distributed arbitrarily with at most one data item per processor. The problem of data concentration is to move the data into the first d processors of the network one data item per processor. This problem is also known as *packing*. In the case of a p-processor linear array, we have to move the data into the processors $1, 2, \ldots, d$. On a mesh, we might require the data items to move according to any indexing scheme of our choice. For example, the data could be moved into the first $\lceil \frac{d}{\sqrt{p}} \rceil$ rows.

Data concentration on any network is achieved by first performing a prefix computation to determine the destination of each packet and then routing the packets using an appropriate packet routing algorithm.

Let $\mathcal{L}$ be a p-processor linear array with d data items. To find the destination of each data item, we make use of a variable x. If processor i has a data item, then it sets $x_i = 1$; otherwise it sets $x_i = 0$. Let the prefixes of the sequence $x_1, x_2, \ldots, x_p$ be $y_1, y_2, \ldots, y_p$. If processor i has a data item, then the destination of this item is y_i. The destinations for the data items having been determined, they are routed. Prefix computation (Lemma 14.4) as well as packet routing on a linear array (Lemma 14.1) takes p time steps each. Thus the total run time is $2p$.

Example 14.6 Consider a six-processor linear array in which there is an item in processors 1, 3, 4, and 6. Then, $(x_1, x_2, x_3, x_4, x_5, x_6) = (1, 0, 1, 1, 0, 1)$ and $(y_1, y_2, y_3, y_4, y_5, y_6) = (1, 1, 2, 3, 3, 4)$. So, the items will be sent to the processors 1, 2, 3, and 4 as expected (see Figure 14.13). □

On a mesh too, the same strategy of computing prefixes followed by packet routing can be employed. Prefix computation takes $3\sqrt{p}+2$ steps (c.f. Theorem 14.3), whereas packet routing can be done in $3\sqrt{p} + \tilde{O}(p^{1/4} \log p)$ steps (c.f. Theorem 14.1).

Example 14.7 Figure 14.14 shows a mesh in which there are six data items a, b, c, d, e, f, and g. The parallel variable x takes a value of one corresponding to any element and zero otherwise. Prefix sums are computed on $x_1, x_2, \ldots, x_{16}$, and finally the data items are routed to their destinations. We have assumed the row major indexing scheme. □

Theorem 14.4 Data concentration on a p-processor linear array takes $2p$ steps or less. On a $\sqrt{p} \times \sqrt{p}$ mesh, it takes $6\sqrt{p} + \tilde{O}(p^{1/4} \log p)$ steps. □

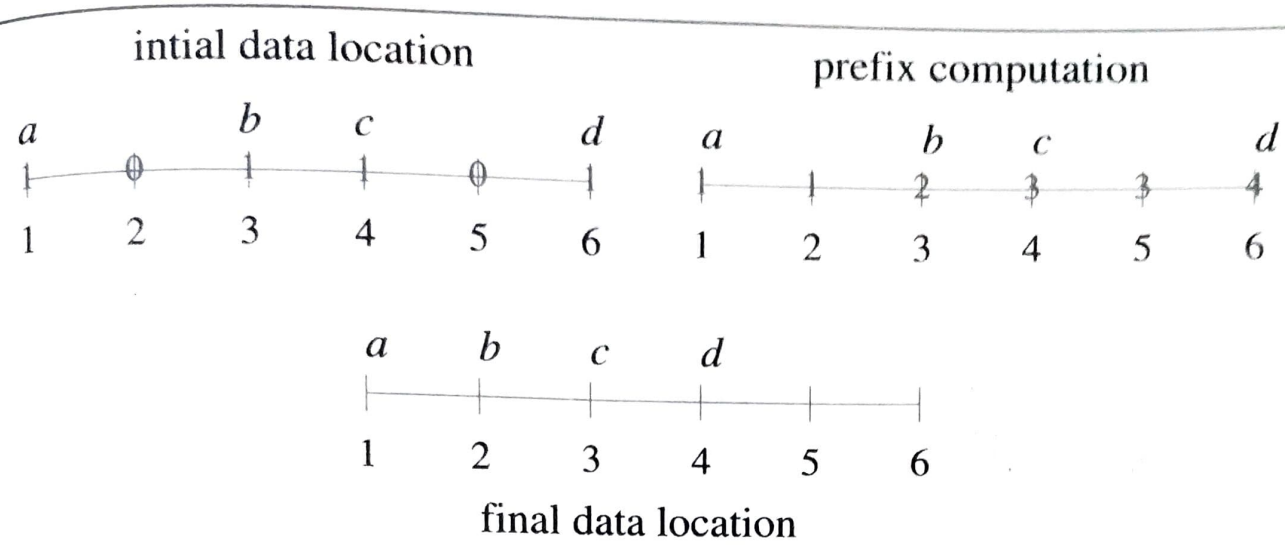

Figure 14.13 Data concentration on a linear array

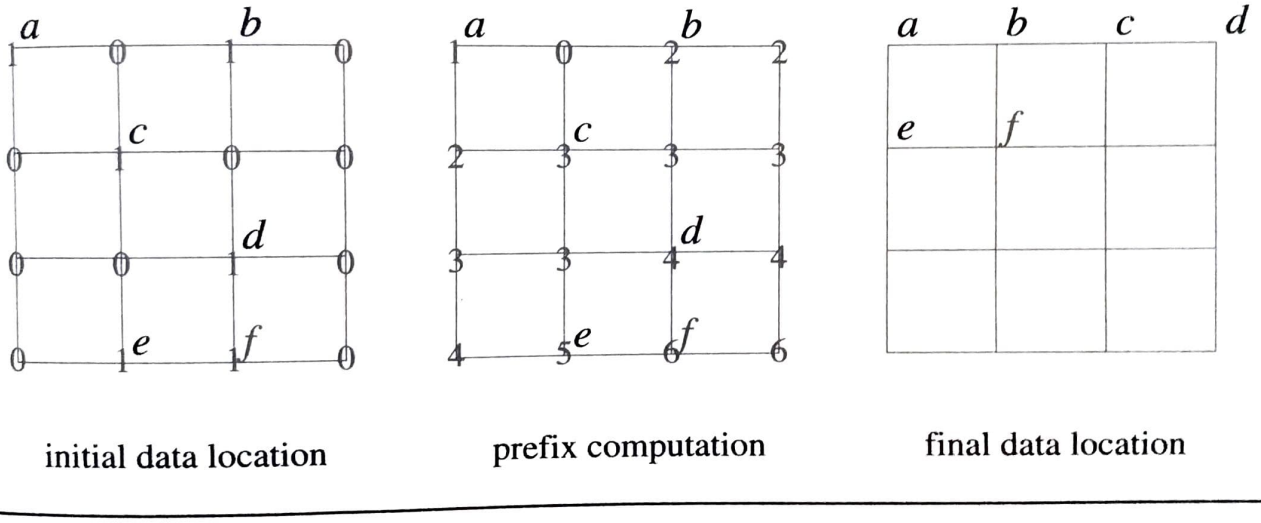

Figure 14.14 Data concentration on a 4×4 mesh

14.3.4 Sparse Enumeration Sort

An instance of sorting in which the number of keys to be sorted is much less than the network size is referred to as the *sparse enumeration sort*. If the network size is p, the number of keys to be sorted is typically assumed to be p^ϵ for some constant $\epsilon \leq \frac{1}{2}$. In the following discussion we assume that $\epsilon = \frac{1}{2}$. On the mesh, sparse enumeration sort can be done by computing the rank of each key and routing the key to its correct position in sorted order (see the proof of Theorem 13.15).

Let the sequence to be sorted be $X = k_1, k_2, \ldots, k_{\sqrt{p}}$. We need to sort

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

The work-optimal algorithm of Section 13.4.5 can be adapted to run optimally on the mesh also. A summary of this algorithm follows. If $X = k_1, k_2, \ldots, k_n$ is the input, the algorithm chooses a random sample (call it S) from X and identifies two elements l_1 and l_2 from S. The elements chosen are such that they bracket the element to be selected with high probability and also the number of input keys that are in the range $[l_1, l_2]$ is small.

After choosing l_1 and l_2, we determine whether the element to be selected is in the range $[l_1, l_2]$. If this is the case, we proceed further and the element to be selected is the $(i - |X_1|)$th element of X_2. If the element to be selected is not in the range $[l_1, l_2]$, we start all over again.

The above process of sampling and elimination is repeated until the number of remaining keys is $\leq n^{0.4}$. After this, we perform an appropriate selection from out of the remaining keys using the sparse enumeration sort (Theorem 14.5). For more details, see Program 13.9.

A *stage* refers to one run of the **while** loop. As shown in Section 13.4.5, there are only $\tilde{O}(1)$ stages in the algorithm.

Step 1 of Program 13.9 takes $O(1)$ time on the mesh. The prefix computations of steps 2 and 5 can be done in a total of $O(\sqrt{p})$ time (c.f. Theorem 14.3). Concentration of steps 3 and 6 takes $O(\sqrt{p})$ time each (see Theorem 14.4). Also, sparse enumeration sort takes the same time in steps 3 and 6 in accordance with Theorem 14.5. The selections of steps 4 and 6 take only $O(1)$ time each since these are selections from sorted sequences. The broadcasts of steps 2, 4, and 5 take $O(\sqrt{p})$ time each (c.f. Theorem 14.2). As a result we arrive at the following theorem.

Theorem 14.6 Selection from $n = p$ keys can be performed in $\tilde{O}(\sqrt{p})$ time on a $\sqrt{p} \times \sqrt{p}$ mesh. □

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

Now we consider the problem of selection when the number of keys is larger than the network size. In particular, assume that $n = p^c$ for some constant $c > 1$. Program 13.9 can be used for this case as well with some minor modifications. 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 of its keys with probability $\frac{1}{N^{1-(1/3c)}}$. So, this step now takes time $\frac{n}{p}$. The number of keys in the sample is $\tilde{O}(N^{1/3c}) = \tilde{o}(\sqrt{p})$. Step 2 remains the same and still takes $O(\sqrt{p})$ time. Since there are only $\tilde{O}(N^{1/3c})$ sample keys, they can be concentrated and sorted in step 3 in time $O(\sqrt{p})$ (c.f. Theorems 14.4 and 14.5). Step 4 takes $O(\sqrt{p})$ time as do steps 5 and 6. So, each stage takes time $O(\frac{n}{p} + \sqrt{p})$.

Lemma 13.3 can be used to show that the number of keys that survive at the end of any stage is $\leq 2\sqrt{\alpha}N^{(1-(1/6c))}\sqrt{\log N} = \tilde{O}(N^{(1-(1/6c))}\sqrt{\log N})$, where N is the number of alive keys at the beginning of this stage. This in turn implies there are only $\tilde{O}(\log\log p)$ stages in the algorithm. In summary, we have the following theorem.

Theorem 14.7 If $n = p^c$ for some constant $c > 1$, selection from n keys can be performed on a $\sqrt{p} \times \sqrt{p}$ mesh in time $\tilde{O}\left((\frac{n}{p} + \sqrt{p})\log\log p\right)$. □

14.4.3 A Deterministic Algorithm For $n > p$

In this section we present a deterministic algorithm for selection whose run time is $O(\frac{n}{p}\log\log p + \sqrt{p}\log n)$. The basic idea behind this algorithm is the same as the one employed in the sequential algorithm of Section 3.7. The sequential algorithm partitions the input into groups (of size, say, 5), finds the median of each group, and computes recursively the median (call it M) of these group medians. Then the rank r_M of M in the input is computed, and as a result, all elements from the input that are either $\leq M$ or $> M$ are dropped, depending on whether $i > r_M$ or $i \leq r_M$, respectively. Finally, an appropriate selection is performed from the remaining keys recursively. We showed that the run time of this algorithm was $O(n)$.

If one has to employ this algorithm on an interconnection network, one has to perform periodic load balancing (i.e., distributing the remaining keys uniformly among all the processors). Load balancing is a time-consuming operation and can be avoided as follows. To begin with, each processor has exactly $\frac{n}{p}$ keys. As the algorithm proceeds, keys get dropped from future consideration. There are always p groups (one group per processor). The remaining keys at each processor constitute its group. We identify the median of each group. Instead of picking the median of these medians as the splitter key M, we choose a *weighted median* of these medians. Each group median is *weighted* with the number of remaining keys in that processor.

Definition 14.1 Let $X = k_1, k_2, \ldots, k_n$ be a sequence of keys, where key k_i has an associated weight w_i, for $1 \leq i \leq n$. Also let $W = \sum_{i=1}^{n} w_i$. The weighted median of X is that $k_j \in X$ which satisfies $\sum_{k_l \in X, k_l \leq k_j} w_{k_l} \geq \frac{W}{2}$ and $\sum_{k_l \in X, k_l \geq k_j} w_{k_l} \geq \frac{W}{2}$. In other words, the total weight of all keys of X that are $\leq k_j$ should be $\geq \frac{W}{2}$ and the total weight of all keys that are $\geq k_j$ also should be $\geq \frac{W}{2}$. □

Example 14.9 Let $X = 9, 15, 12, 6, 5, 2, 21, 17$ and let the respective weights be $1, 2, 1, 2, 3, 1, 7, 5$. Here $W = 22$. The weighted median of X is 17. One way of identifying the weighted median is to sort X; let the sorted sequence

be $k'_1, k'_2, \ldots, k'_n$; let the corresponding weight sequence be $w'_1, w'_2, \ldots, w'_n$; and compute the prefix sums $y_1, y_2, \ldots, y_n$ on this weight sequence. If y_j is the leftmost prefix sum that is $> \frac{W}{2}$, then k'_j is the weighted median.

For X, the sorted order is $2, 5, 6, 9, 12, 15, 17, 21$ and the corresponding weights are $1, 3, 2, 1, 1, 2, 5, 7$. The prefix sums of this weight sequence are $1, 4, 6, 7, 8, 10, 15, 22$. The leftmost prefix sum that exceeds 11 is 15 and hence the weighted median is 17. □

The deterministic selection algorithm makes use of the technique just described for finding the weighted median. To begin with, there are exactly $\frac{n}{p}$ keys at each processor. We need to find the ith smallest key. The detailed description of the algorithm appears as Program 14.5. Here D is a constant.

Example 14.10 Consider a 3×3 mesh where there are three keys at each processor to begin with. Also let $i = 8$. Let the input be $11, 6, 3,\ 18, 2, 14,\ 10, 17, 5,\ 21, 26, 27,\ 12, 7, 25,\ 24, 4, 9,\ 19, 20, 23,\ 15, 8, 22,\ 1, 13, 6$. Figure 14.17 shows the steps in selecting the ith smallest key. It is assumed that parts are of size 1 in step 0 of Program 14.5, to make the discussion simple.

The median of each processor is found in step 1. Since each processor has the same number of keys, the weighted median of these medians is nothing but the median of these. The weighted median M is found to be 14 in step 2. The rank r_M of this weighed median is 14. Since $i < r_M$, all the keys that are greater than or equal to 14 are deleted. The i remains the same. This completes one run of the **while** loop.

In the next run of the **while** loop, the weighted median is found by sorting the local medians. The local medians are $3, 2, 5, 7, 4, 8, 6$. Their corresponding weights are $2, 1, 2, 2, 2, 1, 3$. Sorted order of these medians is $2, 3, 4, 5, 6, 7, 8$, the respective weights being $1, 2, 2, 2, 3, 2, 1$. Thus the weighted median M is found to be 5. The rank r_M of M is 5. So, keys that are less than or equal to 5 get eliminated. The value of i becomes $8 - 5 = 3$.

In the third run of the **while** loop, there are eight keys to begin with. The weighted median is found to be 8, whose rank happens to be 3, which is the same as the value of i. Thus the algorithm terminates and 8 is output as the correct answer. □

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

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

$N = n$;
Step 0. If $\log(n/p)$ is $\leq \log\log p$, then sort the elements at each processor; else partition the keys at each processor into $\log p$ equal parts such that the keys in each part are $\leq$ keys in the parts to the right.

while $(N > D)$ **do**
{

Step 1. In parallel find the median of keys at each processor. Let M_q be the median and N_q be the number of remaining keys at processor q, $1 \leq q \leq p$.

Step 2. Find and broadcast the weighted median of $M_1, M_2, \ldots, M_p$, where key M_q has a weight of N_q, $1 \leq q \leq p$. Let M be the weighted median.

Step 3. Count the rank r_M of M from out of all remaining keys and broadcast it.

Step 4. If $i \leq r_M$, then eliminate all remaining keys that are $> M$; else eliminate all remaining keys that are $\leq M$.

Step 5. Compute and broadcast E, the number of keys eliminated. If $i > r_M$, then $i = i - E$; $N = N - E$;

}

Output the ith smallest key from out of the remaining keys.

Program 14.5 Deterministic selection on a $\sqrt{p} \times \sqrt{p}$ mesh

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, then 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 $O(\sqrt{p})$ time (as we show in Section 14.6). The prefix computation takes $O(\sqrt{p})$ time as well (see Theorem 14.3). Thus M, the weighted median, can be identified in time $O(\sqrt{p})$.

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

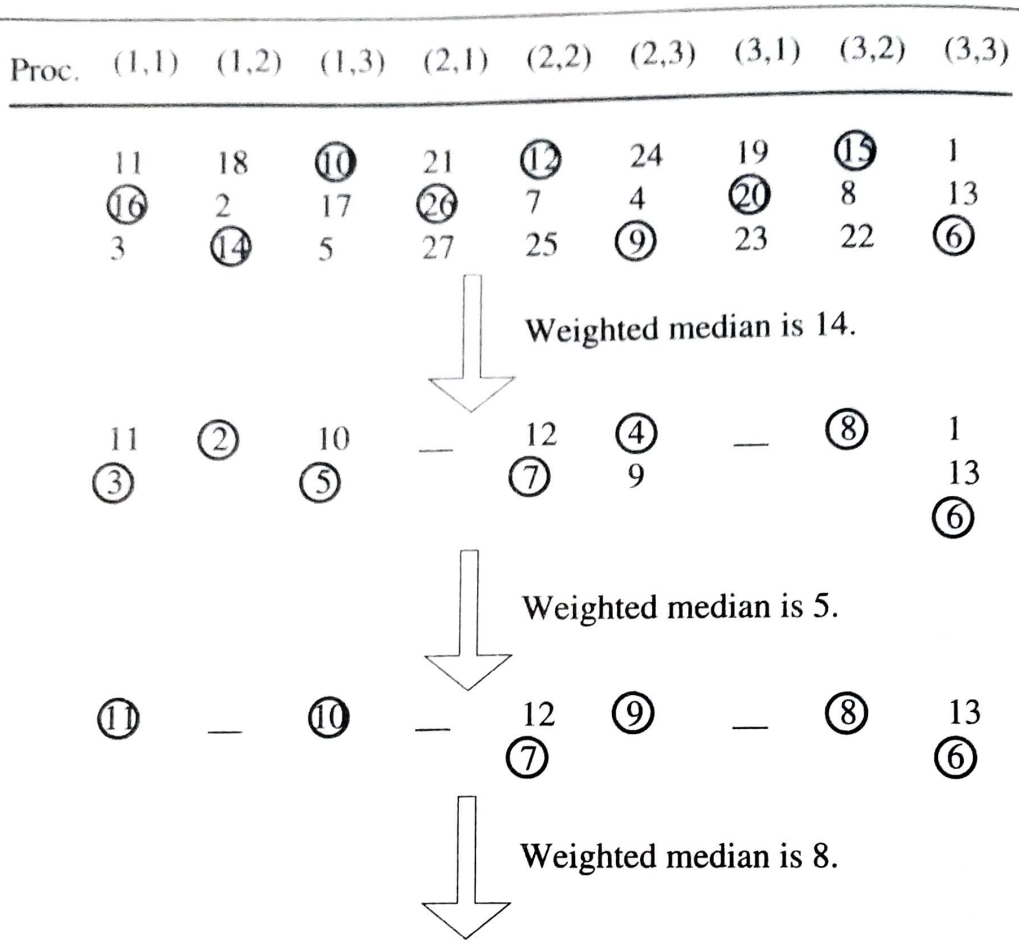

Figure 14.17 Deterministic selection when $n > p$

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, we compare M with the elements of block B to determine the keys to be eliminated. If $i > r_M$ ($i \leq r_M$), of course all blocks to the left (right) of B are eliminated en masse. 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.

Step 5 takes $O(\sqrt{p})$ time, since it involves a prefix computation and a broadcast.

Broadcasting of steps 2, 3, and 4 takes $O(\sqrt{p})$ time each (c.f. Theorem 14.2). Thus each run of the **while** loop takes $O(\frac{n}{p \log p} + \sqrt{p})$ time.

How many keys are eliminated in each run of the **while** loop? Assume that $i \geq r_M$ in a given run. (The other case can be argued similarly.) The

14.6 SORTING

Given a sequence of n keys, recall that the problem of sorting is to rearrange this sequence in either ascending or descending order. In this section we study several algorithms for sorting on both a linear array and a mesh.

14.6.1 Sorting on a Linear Array

Rank sort

The first algorithm, we are going to study, rank sort, computes the rank of each key and then routes the keys to their correct positions. If there are p processors in the linear array with one key per processor, the ranks of all keys can be computed in $O(p)$ time using an algorithm similar to the one employed in the proof of Lemma 14.5. Following this, the key whose rank is r is routed to processor r. This routing also takes $O(p)$ time (Lemma 14.1). Thus we get the following lemma.

Lemma 14.7 A total of p keys can be sorted on a p-processor linear array in $O(p)$ time. □

for $(i = 1; i <= p; i++)$

If i is odd, compare and exchange keys at processors $2j - 1$ and $2j$ for $j = 1, 2, \ldots$; else compare and exchange keys at processors $2j$ and $2j+1$ for $j = 1, 2, \ldots$.

Program 14.7 Odd-even transposition sort

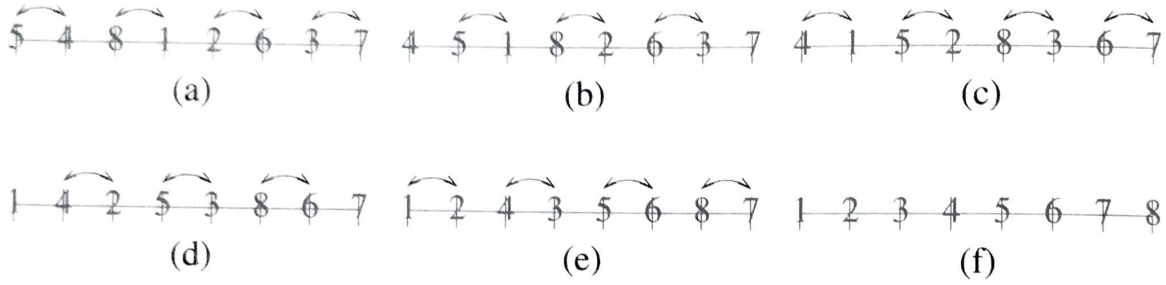

Figure 14.21 Odd-even transposition sort on a linear array

Odd-even transposition sort

An algorithm similar to bubble sort can also be used to sort a linear array in $O(p)$ time. This algorithm (Program 14.7) is also known as the *odd-even transposition sort*. "Compare and exchange" refers to comparing two keys and interchanging them if they are out of order. Each iteration of the **for** loop takes only $O(1)$ time. Thus the whole algorithm terminates in $O(p)$ time steps. The correctness of this algorithm can be proved using the zero-one principle and is left as an exercise.

Lemma 14.8 The odd-even transposition sort runs in $O(p)$ time on a p-processor linear array. □

Example 14.12 Let $p = 8$ and let the keys to be sorted be $4, 5, 1, 8, 2, 6, 3, 7$. Figure 14.21 shows the steps of the odd-even transposition sort. □

Odd-even merge sort

The last algorithm we study on the linear array is based on merge sort; that is, it makes use of a known merging algorithm in order to sort. If there are p

keys on a p-processor linear array, we can recursively sort the first half and the second half at the same time. Once the results are ready, they can be merged using the odd-even merge algorithm (Lemma 14.6). The resultant odd-even merge sort has a run time $T(p) = T(p/2) + O(p)$, which solves to $T(p) = O(p)$.

Lemma 14.9 Odd-even merge sort runs in $O(p)$ time on a p-processor linear array. □

14.6.2 Sorting on a Mesh

We study two different algorithms for sorting on a mesh. The first is called *Shearsort* and takes $O(\sqrt{p}\log p)$ time to sort a $\sqrt{p} \times \sqrt{p}$ mesh. The second is an implementation of odd-even merge sort. This algorithm runs in $O(\sqrt{p})$ time and hence is asymptotically optimal.

Shearsort

This algorithm (Program 14.8) works by alternately sorting the rows and columns. If there is a key at each processor of a $\sqrt{p} \times \sqrt{p}$ mesh, there are $\log p + 1$ phases in the algorithm. At the end, the mesh will be sorted in snakelike row major order. Since a linear array with $\sqrt{p}$ processors can be sorted in $O(\sqrt{p})$ time (c.f. Lemma 14.8), Program 14.8 runs in a total of $O(\sqrt{p}(\log p + 1)) = O(\sqrt{p}\log p)$ time.

Example 14.13 Consider the keys on a 4×4 mesh of Figure 14.22(a). In phase 1, we sort the rows, sorting alternate rows in opposite orders. The result is Figure 14.22(b). The results of the next four phases are shown in Figure 14.22(c), (d), (e), and (f), respectively. At the end of the fifth phase, the mesh is sorted. □

Note that Program 14.8 is comparison based and is also oblivious and hence the zero-one principle can be used to prove its correctness. Assume that the input consists of only zeros and ones. Define a row to be *dirty* if it has both ones and zeros, *clean* otherwise. Note that if the mesh is sorted, there will be only one dirty row and the rest of the rows will either have all ones or all zeros and hence will be clean. To begin with, there could be as many as $\sqrt{p}$ dirty rows; that is, each row could be dirty.

Call a *stage* of the algorithm to be sorting all rows followed by sorting all columns (i.e., a stage consists of two phases). We show that if N is the number of dirty rows at the beginning of any stage, then the number of dirty rows at the end of the stage is no more than $\frac{N}{2}$. This will then imply that after $\log(\sqrt{p})$ stages, there will be at most one dirty row left which can be sorted in an additional row sort. Thus there will be only $2\log(\sqrt{p}) + 1 = \log p + 1$ phases in the algorithm.

(a)

15	12	8	32
7	13	6	17
2	16	19	25
18	11	5	3

(b)

8	12	15	32
17	13	7	6
2	16	19	25
18	11	5	3

(c)

2	11	5	3
8	12	7	6
17	13	15	25
18	16	19	32

(d)

2	3	5	11
12	8	7	6
13	15	17	25
32	19	18	16

(e)

2	3	5	6
12	8	7	11
13	15	17	16
32	19	18	25

(f)

2	3	5	6
12	11	8	7
13	15	16	17
32	25	19	18

Figure 14.22 Shearsort – an example

for (int $i = 1; i <= \log p + 1; i++$)

If i is even, sort the columns in increasing order from top to bottom; else sort the rows. The rows are sorted in such a way that alternate rows are sorted in reverse order. The first row is sorted in increasing order from left to right, the second row is sorted in decreasing order from left to right, and so on.

Program 14.8 Shearsort

Look at two adjacent dirty rows at the beginning of any stage. There are three possibilities: (1) these two rows put together may have an equal number of ones and zeros, (2) the two rows may have more zeros than ones,

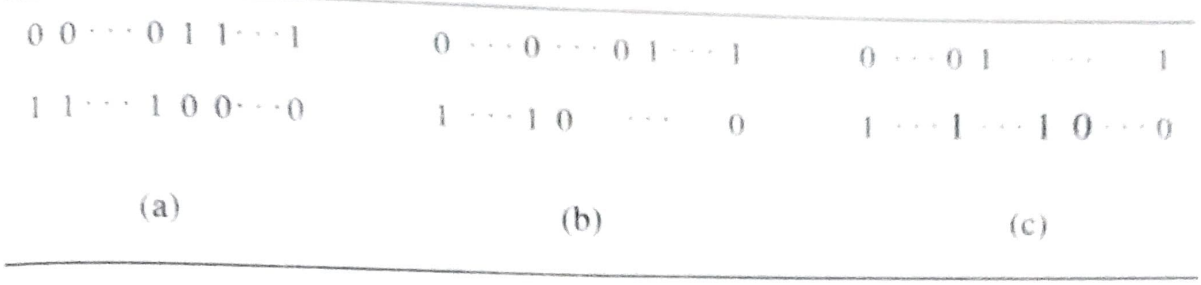

Figure 14.23 Proving the correctness of Program 14.8

and (3) the two rows may have more ones than zeros. In the first phase of this stage the rows are sorted and in the second phase the columns are sorted. In case 1, when the rows are sorted, they will look like Figure 14.23(a). Then, when the columns are sorted, the two rows will contribute two clean rows (one with all ones and the other will all zeros). If case 2 is true, after the row sorting, the two rows will look like Figure 14.23(b). When the columns are sorted, the two rows will contribute one clean row consisting of all zeros. In case 3 also, a clean row (consisting of all ones) will be contributed. In summary, any two adjacent dirty rows will contribute at least one clean row. That is, the number of dirty rows will decrease in any phase by a factor of at least 2.

Theorem 14.10 The Shearsort algorithm (Program 14.8) works correctly and runs in time $O(\sqrt{p}\log p)$ on a $\sqrt{p} \times \sqrt{p}$ mesh. □

Odd-even merge sort

Now we implement the odd-even merge sort method on the mesh. If $X = k_1, k_2, \ldots, k_n$ is the given sequence of n keys, odd-even merge sort partitions X into two subsequences $X_1' = k_1, k_2, \ldots, k_{n/2}$ and $X_2' = k_{n/2+1}, k_{n/2+2}, \ldots, k_n$ of equal length. 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.

We have already seen how the odd-even merge algorithm works on the mesh in $O(\sqrt{p})$ time (Program 14.6). This algorithm can be used in the merging part. Given p keys distributed on a $\sqrt{p} \times \sqrt{p}$ mesh (one key per processor), we can partition them into four equal parts of size $\frac{\sqrt{p}}{2} \times \frac{\sqrt{p}}{2}$ each. Sort each part recursively into snakelike row major order. The result is shown in Figure 14.24(b). Now, merge the top two snakes using Program 14.6. At the same time merge the bottom two snakes using the same algorithm. These mergings take time $O(\sqrt{p})$. After these mergings, the mesh looks like Figure

14.24(c). Finally merge these two snakes by properly modifying Program 14.6. This merging also takes $O(\sqrt{p})$ time. After this merging, the whole mesh is in snakelike row major sorted order (as in Figure 14.24(d)).

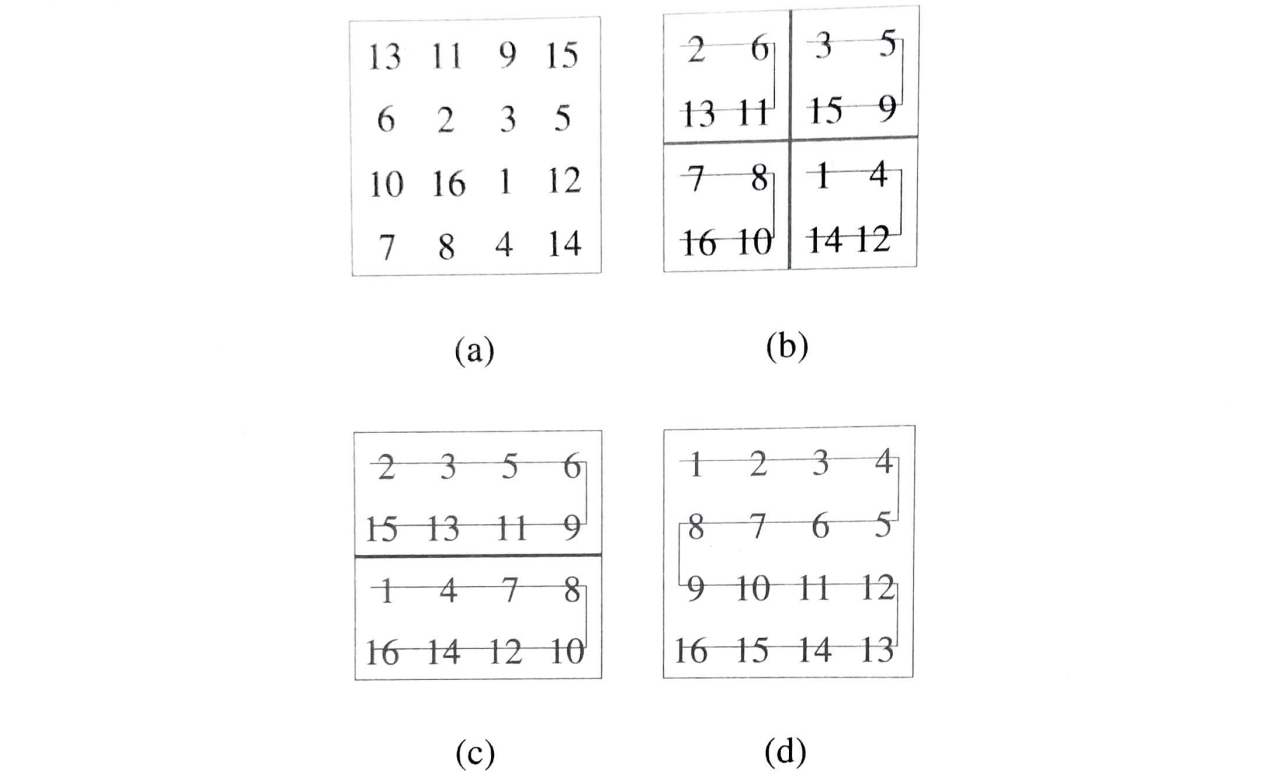

Figure 14.24 Odd-even merge sort on the mesh

If $S(\ell)$ is the time needed to sort an $\ell \times \ell$ mesh using the above divide-and-conquer algorithm, then we have

$$S(\ell) = S\left(\frac{\ell}{2}\right) + O(\ell)$$

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

Theorem 14.11 We can sort p elements in $O(\sqrt{p})$ time on a $\sqrt{p} \times \sqrt{p}$ mesh into snakelike row major order. □

Example 14.14 Figure 14.24(a) shows a 4×4 mesh in which there is a key at each node to begin with. The mesh is partitioned into four quadrants and each quadrant is recursively sorted. The result is Figure 14.24(b). The top two quadrants as well as the bottom two quadrants are merged in parallel (Figure 14.24(c). The resultant two snakes are merged (Figure 14.24(d)). □