either the random or the … heme works well, but in others neither scheme delivers good pe … l pivot selection schemes are examined in Problems 9.20 and 9.21.

9.5 Bucket and Sample Sort

A popular serial algorithm for sorting an array of n elements whose values are uniformly distributed over an interval $[a, b]$ is the ***bucket sort*** algorithm. In this algorithm, the interval $[a, b]$ is divided into m equal-sized subintervals referred to as ***buckets***, and each element is placed in the appropriate bucket. Since the n elements are uniformly distributed over the interval $[a, b]$, the number of elements in each bucket is roughly n/m. The algorithm then sorts the elements in each bucket, yielding a sorted sequence. The run time of this algorithm is $\Theta(n \log(n/m))$. For $m = \Theta(n)$, it exhibits linear run time, $\Theta(n)$. Note that the reason that bucket sort can achieve such a low complexity is because it assumes that the n elements to be sorted are uniformly distributed over an interval $[a, b]$.

Parallelizing bucket sort is straightforward. Let n be the number of elements to be sorted and p be the number of processes. Initially, each process is assigned a block of n/p elements, and the number of buckets is selected to be $m = p$. The parallel formulation of bucket sort consists of three steps. In the first step, each process partitions its block of n/p elements into p sub-blocks, one for each of the p buckets. This is possible because each process knows the interval $[a, b)$ and thus the interval for each bucket. In the second step, each process sends sub-blocks to the appropriate processes. After this step, each process has only the elements belonging to the bucket assigned to it. In the third step, each process sorts its bucket internally by using an optimal sequential sorting algorithm.

Unfortunately, the assumption that the input elements are uniformly distributed over an interval $[a, b]$ is not realistic. In most cases, the actual input may not have such a distribution or its distribution may be unknown. Thus, using bucket sort may result in buckets that have a significantly different number of elements, thereby degrading performance. In such situations an algorithm called ***sample sort*** will yield significantly better performance. The idea behind sample sort is simple. A sample of size s is selected from the n-element sequence, and the range of the buckets is determined by sorting the sample and choosing $m - 1$ elements from the result. These elements (called ***splitters***) divide the sample into m equal-sized buckets. After defining the buckets, the algorithm proceeds in the same way as bucket sort. The performance of sample sort depends on the sample size s and the way it is selected from the n-element sequence.

Consider a splitter selection scheme that guarantees that the number of elements ending up in each bucket is roughly the same for all buckets. Let n be the number of elements to be sorted and m be the number of buckets. The scheme works as follows. It divides the n elements into m blocks of size n/m each, and sorts each block by using quicksort. From each sorted block it chooses $m - 1$ evenly spaced elements. The $m(m - 1)$ elements selected from all the blocks represent the sample used to determine the buckets. This

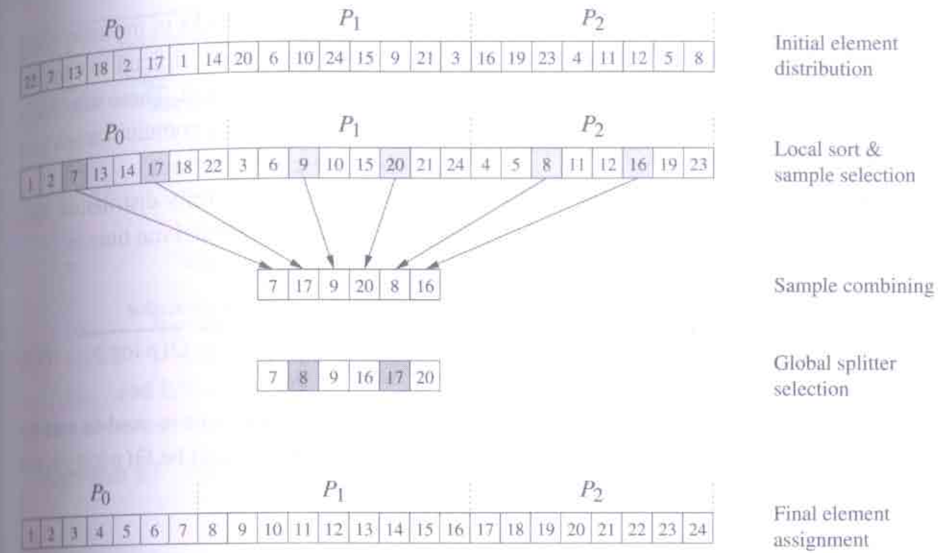

Figure 9.20 An example of the execution of sample sort on an array with 24 elements on three processes.

scheme guarantees that the number of elements ending up in each bucket is less than $2n/m$ (Problem 9.28).

How can we parallelize the splitter selection scheme? Let p be the number of processes. As in bucket sort, set $m = p$; thus, at the end of the algorithm, each process contains only the elements belonging to a single bucket. Each process is assigned a block of n/p elements, which it sorts sequentially. It then chooses $p - 1$ evenly spaced elements from the sorted block. Each process sends its $p - 1$ sample elements to one process – say P_0. Process P_0 then sequentially sorts the $p(p - 1)$ sample elements and selects the $p - 1$ splitters. Finally, process P_0 broadcasts the $p - 1$ splitters to all the other processes. Now the algorithm proceeds in a manner identical to that of bucket sort. This algorithm is illustrated in Figure 9.20.

Analysis We now analyze the complexity of sample sort on a message-passing computer with p processes and $O(p)$ bisection bandwidth.

The internal sort of n/p elements requires time $\Theta((n/p)\log(n/p))$, and the selection of $p-1$ sample elements requires time $\Theta(p)$. Sending $p-1$ elements to process P_0 is similar to a gather operation (Section 4.4); the time required is $\Theta(p^2)$. The time to internally sort the $p(p-1)$ sample elements at P_0 is $\Theta(p^2 \log p)$, and the time to select $p-1$ splitters is $\Theta(p)$. The $p-1$ splitters are sent to all the other processes by using one-to-all broadcast (Section 4.1), which requires time $\Theta(p \log p)$. Each process can *insert* these $p-1$ splitters in its local sorted block of size n/p by performing $p - 1$ binary searches. Each process thus partitions its block into p sub-blocks, one for each bucket. The time required for this partitioning is $\Theta(p \log(n/p))$. Each process then sends sub-blocks to the appropriate processes (that is, buckets). The communication time for this step is difficult to compute precisely, as it depends on the size of the sub-blocks to be communicated. These sub-blocks can vary arbitrarily between 0 and n/p. Thus, the upper bound on the communication time is $O(n) + O(p \log p)$.

If we assume that the elements stored in each process are uniformly distributed, then each sub-block has roughly $\Theta(n/p^2)$ elements. In this case, the parallel run time is

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(p^2 \log p\right)}^{\text{sort sample}} + \overbrace{\Theta\left(p \log\frac{n}{p}\right)}^{\text{block partition}} + \overbrace{\Theta(n/p) + O(p \log p)}^{\text{communication}}. \quad (9.9)$$

In this case, the isoefficiency function is $\Theta(p^3 \log p)$. If bitonic sort is used to sort the $p(p-1)$ sample elements, then the time for sorting the sample would be $\Theta(p \log p)$, and the isoefficiency will be reduced to $\Theta(p^2 \log p)$ (Problem 9.30).

9.6 Other Sorting Algorithms

As mentioned in the introduction to this chapter, there are many sorting algorithms, and we cannot explore them all in this chapter. However, in this section we briefly present two additional sorting algorithms that are important both practically and theoretically. Our discussion of these schemes will be brief. Refer to the bibliographic remarks (Section 9.7) for references on these and other algorithms.

9.6.1 Enumeration Sort

All the sorting algorithms presented so far are based on compare-exchange operations. This section considers an algorithm based on ***enumeration sort***, which does not use compare-exchange. The basic idea behind enumeration sort is to determine the rank of each element. The ***rank*** of an element a_i is the number of elements smaller than a_i in the sequence to be sorted. The rank of a_i can be used to place it in its correct position in the sorted sequence. Several parallel algorithms are based on enumeration sort. Here we present one such algorithm that is suited to the CRCW PRAM model. This formulation sorts n elements by using n^2 processes in time $\Theta(1)$.

Assume that concurrent writes to the same memory location of the CRCW PRAM result in the sum of all the values written being stored at that location (Section 2.4.1). Consider the n^2 processes as being arranged in a two-dimensional grid. The algorithm consists of two steps. During the first step, each column j of processes computes the number of elements smaller than a_j. During the second step, each process $P_{1,j}$ of the first row places a_j in its proper position as determined by its rank. The algorithm is shown in Algorithm 9.7. It uses an auxiliary array $C[1 \ldots n]$ to store the rank of each element. The crucial steps of this algorithm are lines 7 and 9. There, each process $P_{i,j}$, writes 1 in $C[j]$

```
1.   procedure ENUM_SORT (n)
2.   begin
3.       for each process P_{1,j} do
4.           C[j] := 0;
5.       for each process P_{i,j} do
6.           if (A[i] < A[j]) or (A[i] = A[j] and i < j) then
7.               C[j] := 1;
8.           else
9.               C[j] := 0;
10.      for each process P_{1,j} do
11.          A[C[j]] := A[j];
12.  end ENUM_SORT
```

Algorithm 9.7 Enumeration sort on a CRCW PRAM with additive-write conflict resolution.

if the element $A[i]$ is smaller than $A[j]$ and writes 0 otherwise. Because of the additive-write conflict resolution scheme, the effect of these instructions is to count the number of elements smaller than $A[j]$ and thus compute its rank. The run time of this algorithm is $\Theta(1)$. Modifications of this algorithm for various parallel architectures are discussed in Problem 9.26.

9.6.2 Radix Sort

The ***radix sort*** algorithm relies on the binary representation of the elements to be sorted. Let b be the number of bits in the binary representation of an element. The radix sort algorithm examines the elements to be sorted r bits at a time, where $r < b$. Radix sort requires b/r iterations. During iteration i, it sorts the elements according to their i^{th} least significant block of r bits. For radix sort to work properly, each of the b/r sorts must be stable. A sorting algorithm is ***stable*** if its output preserves the order of input elements with the same value. Radix sort is stable if it preserves the input order of any two r-bit blocks when these blocks are equal. The most common implementation of the intermediate b/r radix-2^r sorts uses enumeration sort (Section 9.6.1) because the range of possible values $[0 \ldots 2^r - 1]$ is small. For such cases, enumeration sort significantly outperforms any comparison-based sorting algorithm.

Consider a parallel formulation of radix sort for n elements on a message-passing computer with n processes. The parallel radix sort algorithm is shown in Algorithm 9.8. The main loop of the algorithm (lines 3–17) performs the b/r enumeration sorts of the r-bit blocks. The enumeration sort is performed by using the *prefix_sum()* and *parallel_sum()* functions. These functions are similar to those described in Sections 4.1 and 4.3. During each iteration of the inner loop (lines 6–15), radix sort determines the position of the elements with an r-bit value of j. It does this by summing all the elements with the same value and then assigning them to processes. The variable *rank* holds the position of each