Chapter 10

Sorting Algorithms

This chapter is concerned with ways to sort numbers. Sorting is usually studied in sequential programming classes, and there are a number of well-known sorting methods. Here we select some sequential sorting algorithms for conversion to a parallel implementation. We also describe sorting algorithms specifically invented for parallel implementation. Finally, some sorting algorithms that have received recent attention for implementation on clusters are considered.

10.1 GENERAL

10.1.1 Sorting

Sorting numbers — that is, rearranging a list of numbers into increasing (or decreasing) order — is a fundamental operation that appears in many applications. If numbers have duplicates in their sequence, sorting is more properly defined as placing the numbers in *nondecreasing* (or nonincreasing) order. Sorting is also applicable to non-numerical values; for example, rearranging strings into alphabetical order. Sorting is often done because it makes searches and other operations easier. Books in a library are sorted into subject/name for ease of searching for specific books.

Two sorting algorithms have already been presented to demonstrate specific parallelizing techniques. Bucket sort demonstrated divide and conquer in Chapter 4, Section 4.2.1, and insertion sort demonstrated a pipeline in Chapter 5, Section 5.2.2. In this chapter, we will look at other sorting algorithms and apply the most suitable parallelizing techniques to them. Many parallel sorting algorithms and parallel implementations of sequential sorting algorithms are synchronous algorithms, in that a group of actions taking place on the numbers must be completed before the next group of actions takes place. Thus, the synchronous iteration techniques described in Chapter 6 are applicable.

10.1.2 Potential Speedup

Quicksort and mergesort are popular sequential sorting algorithms. They are in the family of "comparison-based" sorting algorithms that sort only on the basis of comparing pairs of numbers. The worst-case time complexity of mergesort and the average time complexity of quicksort are both $O(n\log n)$, where there are n numbers. $O(n\log n)$ is, in fact, optimal for any comparison-based sequential sorting algorithm without using any special properties of the numbers. Therefore, the best parallel time complexity we can expect, based upon a sequential sorting algorithm but using p processors, is

$$\text{Optimal parallel time complexity} = \frac{O(n\log n)}{p} = O(\log n) \quad \text{if } p = n$$

An $O(\log n)$ sorting algorithm with n processors was demonstrated by Leighton (1984), based upon an algorithm by Ajtai, Komlós, and Szemerédi (1983), but the constant hidden in the order notation is extremely large. An $O(\log n)$ sorting algorithm is also described by Leighton (1994) for an n-processor hypercube using random operations. Akl (1985) describes 20 different parallel sorting algorithms, several of which achieve the lower bound for a particular interconnection network. But, in general, a realistic $O(\log n)$ algorithm with n processors is a goal that is not easy to achieve with comparison-based sorting algorithms. It may be that the number of processors will be greater than n. Having said that, let us start with traditional comparison-based algorithms.

10.2 COMPARE-AND-EXCHANGE SORTING ALGORITHMS

10.2.1 Compare and Exchange

An operation that can form the basis of several, if not most, classical sequential sorting algorithms is the compare-and-exchange (or compare-and-swap) operation. In a compare-and-exchange operation, two numbers, say A and B, are compared. If $A > B$, A and B are exchanged; that is, the content of the location holding A is moved into the location holding B, and the content of the location holding B is moved into the location holding A; otherwise, the contents of the locations remain unchanged. Compare and exchange is illustrated in the sequential code

```
if (A > B) {
   temp = A;
   A = B;
   B = temp;
}
```

Compare and exchange is well suited to a message-passing system. Suppose the compare and exchange is between the two numbers A and B, where A is held in process P_1 and B is held in process P_2. One simple way of implementing the compare and exchange is for P_1 to send A to P_2, which then compares A and B and sends back B to P_1 if A is larger than B (otherwise it sends back A to P_1). It is not strictly necessary to send A back to P_1, since it already has A, but it does make it easier to have the same number of sends/receives in either case. The method is illustrated in Figure 10.1. The code could be

Process P_1

```
send(&A, P2);
recv(&A, P2);
```

Process P_2

```
recv(&A, P1);
if (A > B) {
   send(&B, P1);
   B = A;
} else
   send(&A, P1);
```

An alternative way is for P_1 to send A to P_2, and P_2 to send B to P_1. Then both processes perform compare operations. P_1 keeps the smaller of A and B, and P_2 keeps the larger of A and B, as illustrated in Figure 10.2. The code for this approach is

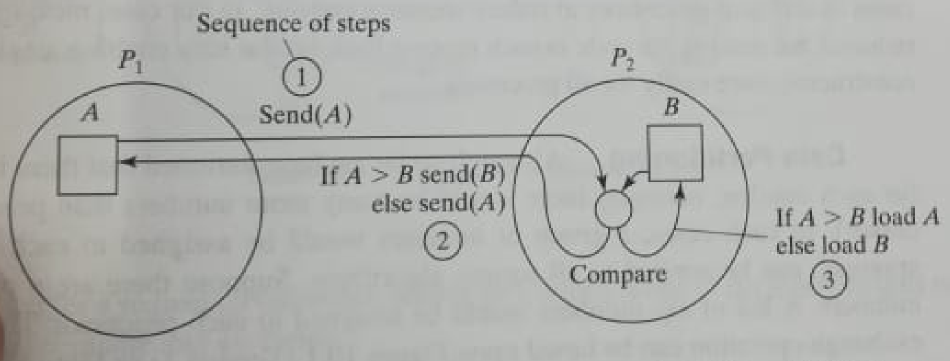


Figure 10.1 Compare and exchange on a message-passing system — Version 1.

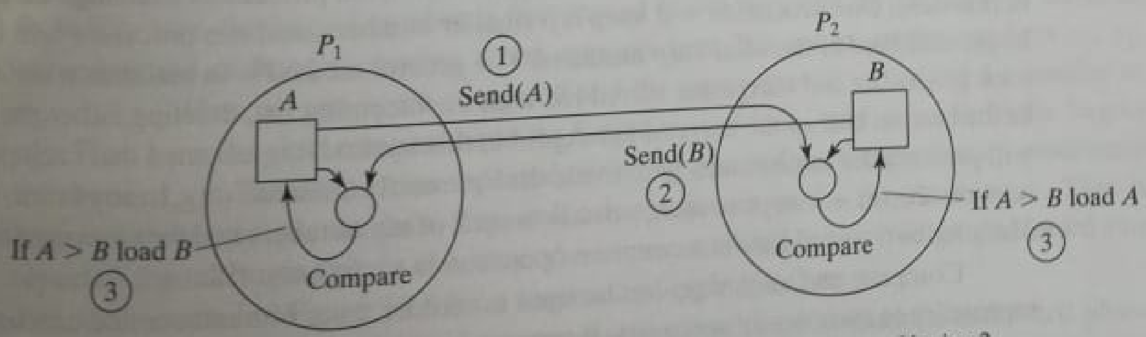


Figure 10.2 Compare and exchange on a message-passing system — Version 2.

Process P_1

```
send(&A, P2);
recv(&B, P2);
if (A > B) A = B;
```

Process P_2

```
recv(&A, P1);
send(&B, P1);
if (A > B) B = A;
```

Process P_1 performs the `send()` first, and process P_2 performs the `recv()` first to avoid deadlock. (In the first version, `send()`s and `recv()`s are naturally in nondeadlock order.) Alternatively, both P_1 and P_2 could perform `send()` first if locally blocking (asynchronous) sends are used and sufficient buffering is guaranteed to exist. Then both processes could initiate their message transfers simultaneously to overlap the message transfer overhead. This is not safe programming in the MPI sense, because deadlock would occur if sufficient buffering were not present.

Note on Precision of Duplicated Computations. The preceding code assumes that the `if` condition, `A > B`, will return the same Boolean answer in both processors; different processors operating at different precisions could conceivably produce different answers if real numbers are being compared. This situation applies wherever computations are duplicated in different processors to reduce message-passing. In our case, message-passing is not reduced, but making the code in each process look similar may enable a single program to be constructed more easily for all processes.

Data Partitioning. Although so far we have assumed that there is one processor for each number, normally there would be many more numbers than processors (or processes). In such cases, a group of numbers would be assigned to each processor. This approach can be applied to all sorting algorithms. Suppose there are p processors and n numbers. A list of n/p numbers would be assigned to each processor. The compare-and-exchange operation can be based upon Figure 10.1 (Version 1) or Figure 10.2 (Version 2). Version 1 is as illustrated in Figure 10.3. Only one processor sends its partition to the other, which then performs the merge operation and returns the lower half of the list back to the first process. Version 2 is illustrated in Figure 10.4. Both processors exchange their groups in this case. One processor will keep n/p smaller numbers, and one processor will keep n/p larger numbers of the total $2n/p$ numbers. The general method is to maintain a sorted list in each processor and merge the stored list with the incoming list, deleting either the top half or the bottom half of the merged list. Again in this method it is assumed that each processor will produce the same results and divide the list exactly the same way. In any event, merging requires $2(n/p) - 1$ steps to merge two lists each of n/p numbers and two message transfers. Merging two sorted lists is a common operation in sorting algorithms.

Compare and exchange can be used to reorder pairs of numbers and can be applied repeatedly to completely sort a list. It appears in quicksort and mergesort. First, though, let us consider a compare-and-exchange algorithm that is less attractive than quicksort or

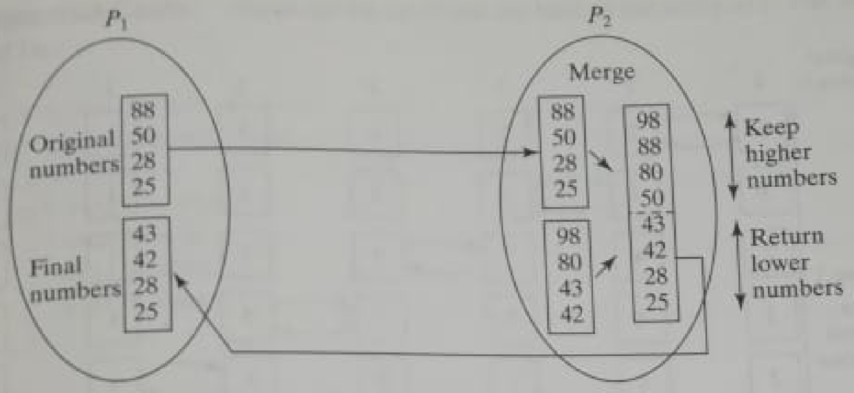


Figure 10.3 Merging two sublists — Version 1.

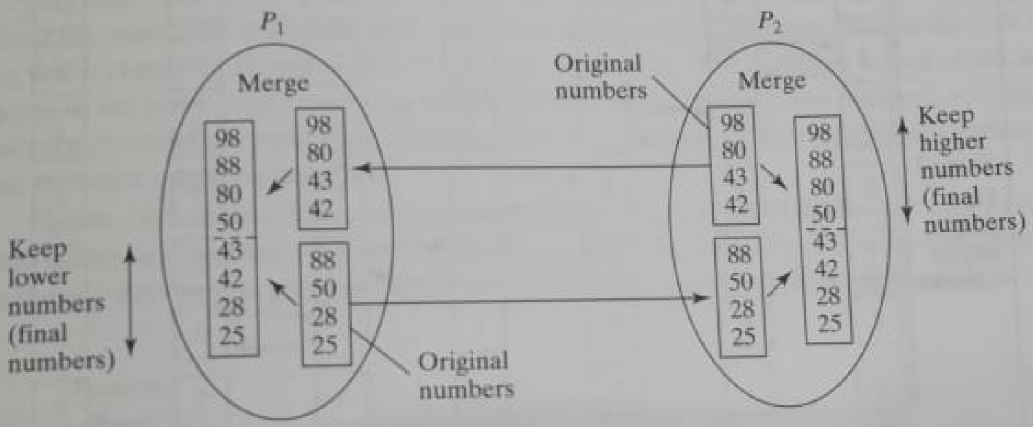


Figure 10.4 Merging two sublists — Version 2.

mergesort for a sequential computer, bubble sort, which is one of the most obvious methods for using compare and exchange.

10.2.2 Bubble Sort and Odd-Even Transposition Sort

In *bubble sort*, the largest number is first moved to the very end of the list by a series of compares and exchanges, starting at the opposite end. We are given the numbers x_0, x_1, x_2, …, x_{n-1}. First, x_0 and x_1 are compared and the larger moved to x_1 (and the smaller to x_0). Then x_1 and x_2 are compared and the larger moved to x_2, and so on until the largest number is x_{n-1}. The actions are repeated, stopping just before the previously positioned largest number, to get the next-largest number adjacent to the largest number. This is repeated for each number. In this way, the larger numbers move ("bubble") toward one end, as illustrated in Figure 10.5 for a sequence of eight numbers.

With n numbers, there are $n - 1$ compare-and-exchange operations in the first phase of positioning the largest number at the end of the list. In the next phase of positioning the

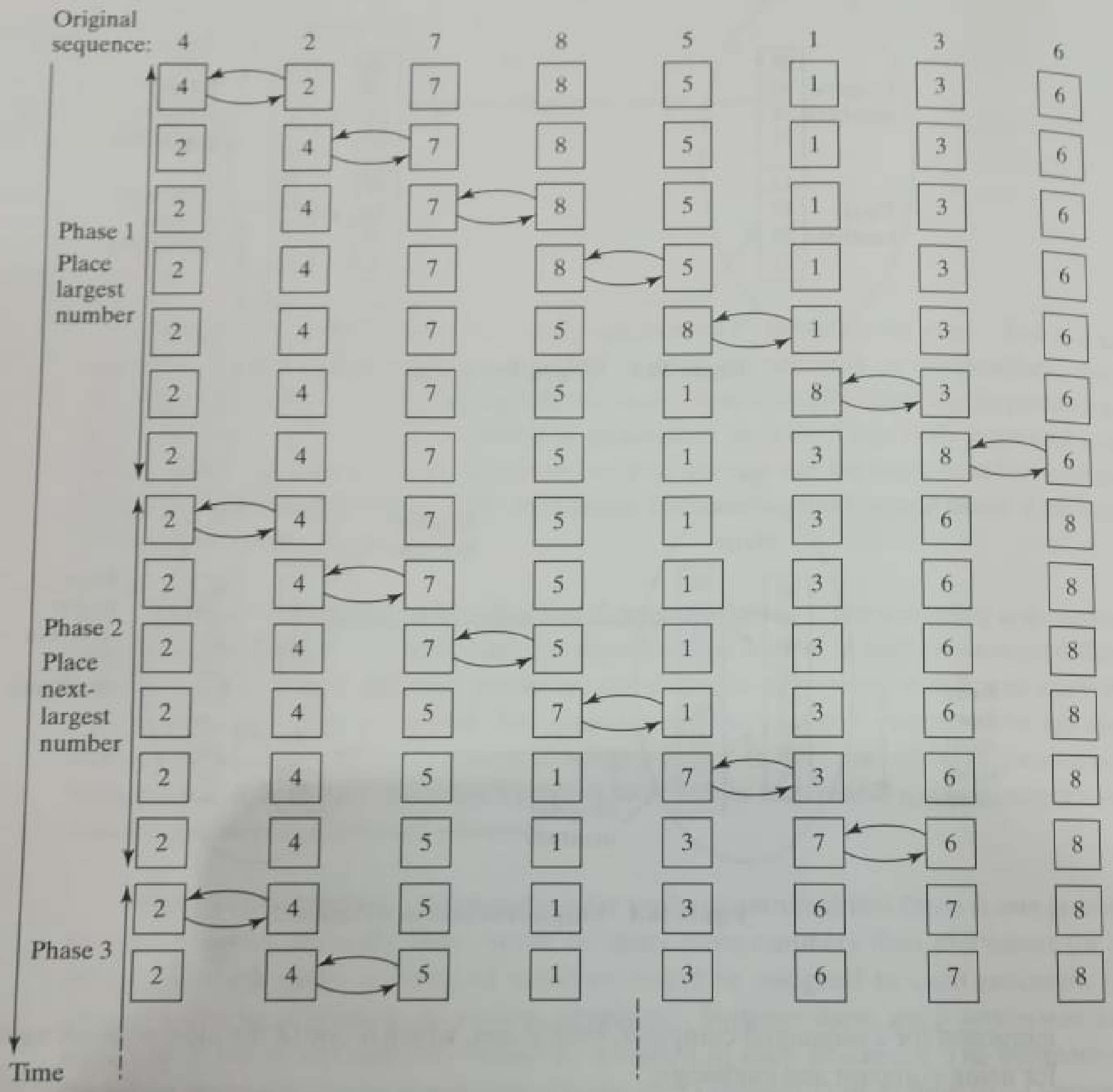


Figure 10.5 Steps in bubble sort.

next largest number, there are $n - 2$ compare-and-exchange operations, and so on. Therefore, the total number of operations is given by

$$\text{Number of compare and exchange operations} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

which indicates a time complexity of $O(n^2)$ given that a single compare-and-exchange operation has a constant complexity, $O(1)$.

Sequential Code. Suppose the numbers are held in the array a[]. The sequential code could be

```
for (i = n - 1; i > 0; i--)
  for (j = 0; j < i; j++) {
    k = j + 1;
    if (a[j] > a[k]) {
      temp = a[j];
      a[j] = a[k];
      a[k] = temp;
    }
  }
```

Parallel Code — Odd-Even Transposition Sort. Bubble sort, as written, is a purely sequential algorithm. Each step in the inner loop takes place before the next, and the whole inner loop is completed before the next iteration of the outer loop. However, just because the sequential code uses statements that depend upon previous statements does not mean that it cannot be reformulated as a parallel algorithm. The bubbling action of the next iteration of the inner loop could start before the preceding iteration has finished, so long as it does not overtake the preceding bubbling action. This suggests that a pipeline implementation structure might be beneficial.

Figure 10.6 shows how subsequent exchange actions of bubble sort can operate behind others in a pipeline. We see that iteration 2 can operate behind iteration 1 at the same time if separated by one process. Similarly, iteration 3 can operate behind iteration 2.

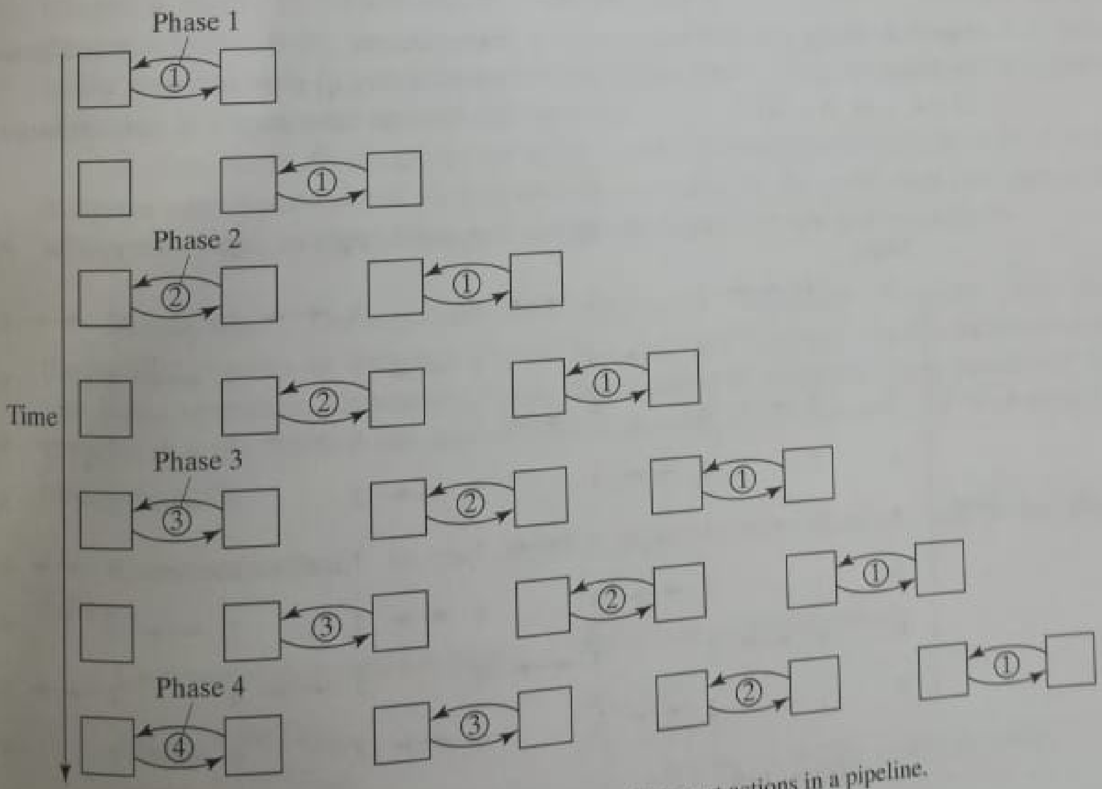


Figure 10.6 Overlapping bubble sort actions in a pipeline.

This idea leads to a variation of bubble sort called *odd-even (transposition) sort*, which operates in two alternating phases, an *even* phase and an *odd* phase. In the even phase, even-numbered processes exchange numbers with their right neighbors. Similarly, in the odd phase, odd-numbered processes exchange numbers with their right neighbors. Odd-even transposition sort normally would not be discussed for sequential programming as it has no particular advantage over normal bubble sort when programmed that way. However, the parallel implementation reduces the time complexity to $O(n)$. Odd-even transposition sort can be implemented on a line network and is optimal for that network (since it requires n steps to reposition a number in the worst case). Odd-even transposition sort applied to a sequence of eight numbers, one number stored in each process, is shown in Figure 10.7.

First let us consider the two distinct alternating phases separately. In the even phase, we have the compare and exchanges: $P_0 \leftrightarrow P_1$, $P_2 \leftrightarrow P_3$, etc. Using the same form of compare and exchange as Version 2 in Section 10.2.1, the code could be

P_i, $i = 0, 2, 4, \ldots, n-2$ (even) $\qquad$ P_i, $i = 1, 3, 5, \ldots, n-1$ (odd)

```
recv(&A, Pi+1);                    send(&A, Pi-1);        /* even phase */
send(&B, Pi+1);                    recv(&B, Pi-1);
if (A < B) B = A;                  if (A < B) A = B;      /* exchange */
```

where the number stored in P_i (even) is B, and the number stored in P_i (odd) is A. In the odd phase, we have the compare and exchanges: $P_1 \leftrightarrow P_2$, $P_3 \leftrightarrow P_4$, etc. This could be coded as

P_i, $i = 1, 3, 5, \ldots, n-3$ (odd) $\qquad$ P_i, $i = 2, 4, 6, \ldots, n-2$ (even)

```
send(&A, Pi+1);                    recv(&A, Pi-1);        /* odd phase */
recv(&B, Pi+1);                    send(&B, Pi-1);
if (A > B) A = B;                  if (A > B) B = A;      /* exchange */
```

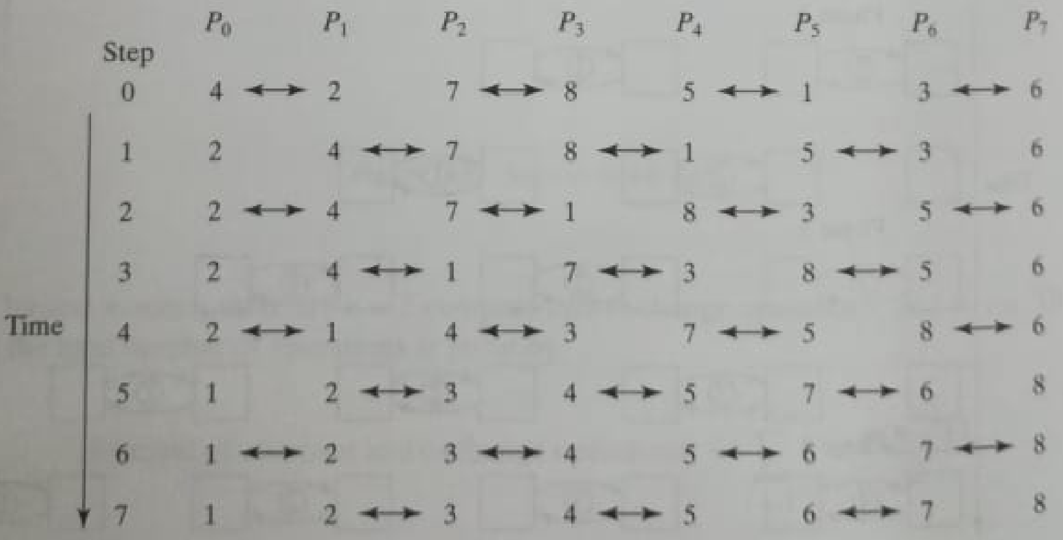


Figure 10.7 Odd-even transposition sort sorting eight numbers.

In both cases, odd-numbered processes execute their `send()` routines first, and even-numbered processes execute their `recv()` routines first. Combining,

P_i, $i = 1, 3, 5, \ldots, n-3$ (odd) P_i, $i = 0, 2, 4, \ldots, n-2$ (even)

```
send(&A, Pi-1);
recv(&B, Pi-1);                          recv(&A, Pi+1);        /* even phase */
if (A < B) A = B;                        send(&B, Pi+1);
if (i <= n-3) {                          if (A < B) B = A;
   send(&A, Pi+1);                       if (i >= 2) {          /* odd phase */
   recv(&B, Pi+1)                           recv(&A, Pi-1);
   if (A > B) A = B;                        send(&B, Pi-1);
}                                           if (A > B) B = A;
                                         }
```

These code segments could be combined into an SPMD form in which the identity of the process is used to select the part of the program that a particular processor will execute (Problem 10-5).

10.2.3 Mergesort

Mergesort is a classical sequential sorting algorithm using a divide-and-conquer approach. The unsorted list is first divided in half. Each half is divided in two. This is continued until individual numbers are obtained. Then pairs of numbers are combined (merged) into sorted lists of two numbers each. Pairs of these lists of four numbers are merged into sorted lists of eight numbers. This is continued until the one fully sorted list is obtained. Described in this manner, it is clear that the algorithm will map perfectly into the tree structures of Chapter 4. Figure 10.8 illustrates the algorithm operating on eight numbers. We can see that this construction is the same as the tree structure to divide a problem (Figure 4.3) followed by the tree structure to combine a problem (Figure 4.4). It also follows that the processor allocation in Figures 4.3 and 4.4 can be made.

A significant disadvantage of using a tree construction is that the load is not well balanced among processors. At first one process is active, then two, then four, and so on. At subsequent steps, as the sublists get smaller the processor has less work to do.

Analysis. The sequential time complexity is $O(n \log n)$. There are $2 \log n$ steps in the parallel version, as shown in Figure 10.8, but each step may need to perform more than one basic operation, depending upon the number of numbers being processed. Let us suppose that the sublists are sent to their respective processors and that the merging takes place internally.

Communication. In the division phase, communication only takes place as follows:

Communication at each step	Processor communication
$t_{\text{startup}} + (n/2)t_{\text{data}}$	$P_0 \rightarrow P_4$
$t_{\text{startup}} + (n/4)t_{\text{data}}$	$P_0 \rightarrow P_2$; $P_4 \rightarrow P_6$
$t_{\text{startup}} + (n/8)t_{\text{data}}$	$P_0 \rightarrow P_1$; $P_2 \rightarrow P_3$; $P_4 \rightarrow P_5$; $P_6 \rightarrow P_7$
$\vdots$	

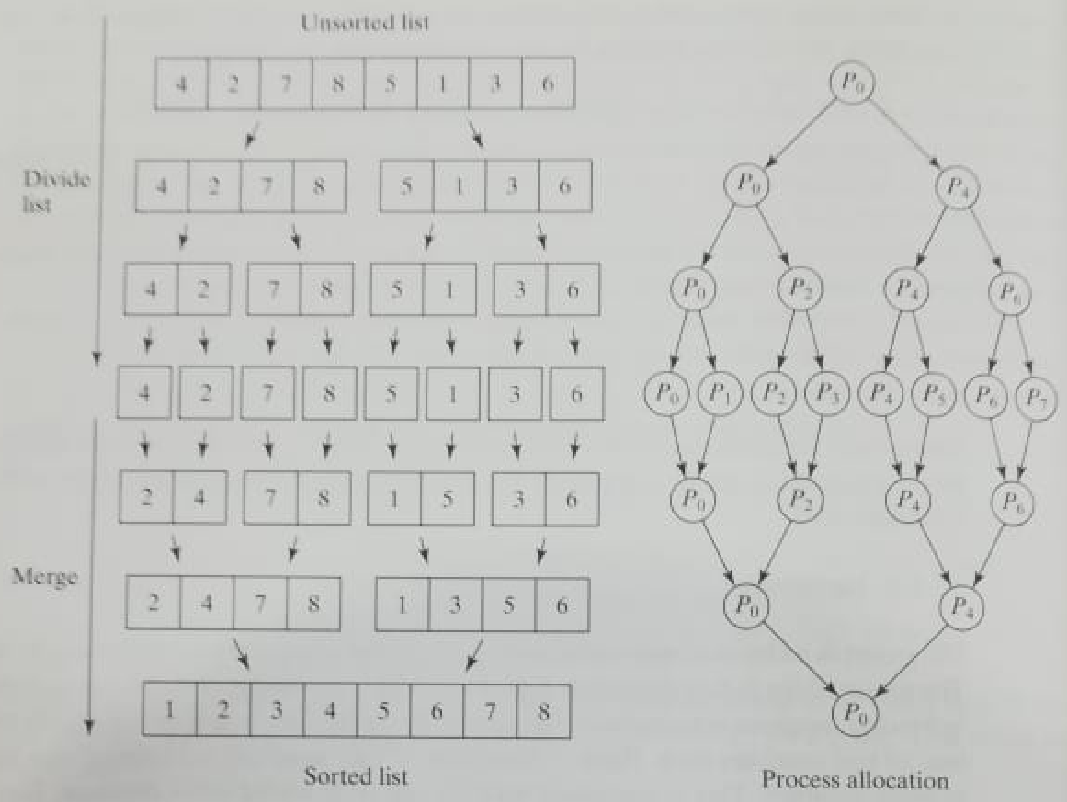


Figure 10.8 Mergesort using tree allocation of processes.

with log p steps, given p processors. In the merge phase, the reverse communications take place:

$$\vdots$$

$t_{\text{startup}} + (n/8)t_{\text{data}}$	$P_0 \leftarrow P_1;\ P_2 \leftarrow P_3;\ P_4 \leftarrow P_5;\ P_6 \leftarrow P_7$
$t_{\text{startup}} + (n/4)t_{\text{data}}$	$P_0 \leftarrow P_2;\ P_4 \leftarrow P_6$
$t_{\text{startup}} + (n/2)t_{\text{data}}$	$P_0 \leftarrow P_4$

again log p steps. This leads to the communication time being

$$t_{\text{comm}} = 2(t_{\text{startup}} + (n/2)t_{\text{data}} + t_{\text{startup}} + (n/4)t_{\text{data}} + t_{\text{startup}} + (n/8)t_{\text{data}} + \ldots)$$

or

$$t_{\text{comm}} \approx 2(\log p)t_{\text{startup}} + 2nt_{\text{data}}$$

Computation. Computations only occurs in merging the sublists. Merging can be done by stepping through each list, moving the smallest found into the final list first. In the worst case, it takes $2n - 1$ steps to merge two sorted lists, each of n numbers, into one sorted list in this manner. For eight numbers, the computation consists of

$t_{\text{comp}} = 1$	$P_0;\ P_2;\ P_4;\ P_6$
$t_{\text{comp}} = 3$	$P_0;\ P_4$
$t_{\text{comp}} = 7$	P_0

Hence:

$$t_{\text{comp}} = \sum_{i=1}^{\log p} (2^i - 1)$$

The parallel computational time complexity is $O(p)$ using p processors and one number in each processor. As with all sorting algorithms, normally we would partition the list into groups, one group of numbers for each processor.

10.2.4 Quicksort

Quicksort (Hoare, 1962) is a very popular sequential sorting algorithm that performs well with an average sequential time complexity of $O(n \log n)$. The question to answer is whether a direct parallel version can achieve the time complexity of $O(\log n)$ with n processors. We did not manage this with mergesort according to our previous analysis. Now let us examine quicksort as a basis for a parallel sorting algorithm.

To recall from sequential programming, quicksort sorts a list of numbers by first dividing the list into two sublists, as in mergesort. All the numbers in one sublist are arranged to be smaller than all the numbers in the other sublist. This is achieved by first selecting one number, called a *pivot*, against which every other number is compared. If the number is less than the pivot, it is placed in one sublist. Otherwise, it is placed in the other sublist. The pivot could be any number in the list, but often the first number in the list is chosen. The pivot could be placed in one sublist, or it could be separated and placed in its final position. We shall separate the pivot.

The procedure is repeated on the sublists, creating four sublists, essentially putting numbers into regions, as in bucket sort (Chapter 4, Section 4.2.1). The difference is that the regions are determined by the pivots selected at each step. By repeating the procedure sufficiently, we are left with sublists of one number each. With proper ordering of the sublists, a sorted list is obtained.

Sequential Code. Quicksort is usually described by a recursive algorithm. Suppose an array, `list[]`, holds the list of numbers, and `pivot` is the index in the array of the final position of the pivot. We could have code of the form

```
quicksort(list, start, end)
{
  if (start < end) {
    partition(list, start, end, pivot)
    quicksort(list, start, pivot-1);     /* recursively call on sublists*/
    quicksort(list, pivot+1, end);
  }
}
```

`Partition()` moves numbers in the list between `start` and `end` so that those less than the pivot are before the pivot and those equal or greater than the pivot are after the pivot. The pivot is in its final position in the sorted list.

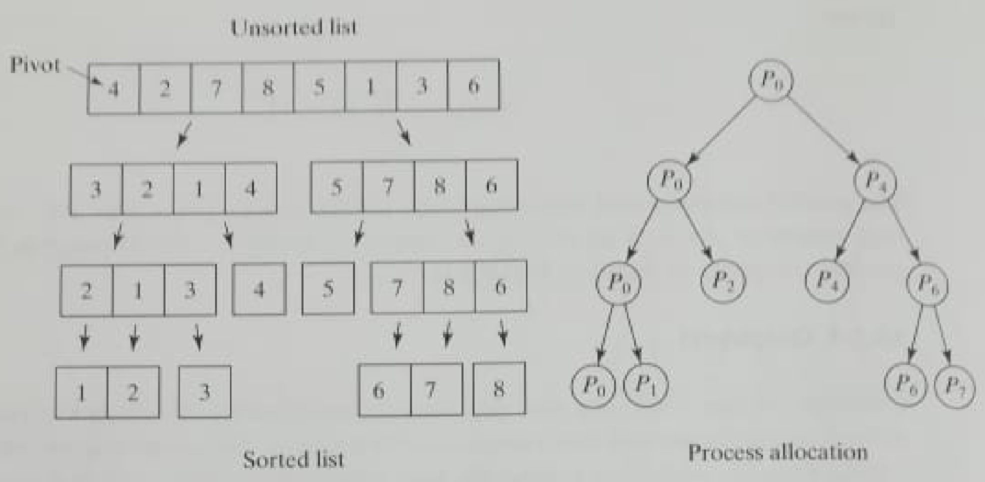


Figure 10.9 Quicksort using tree allocation of processes.

Parallelizing Quicksort. One obvious way to parallelize quicksort is to start with one processor and pass on one of the recursive calls to another processor while keeping the other recursive call to perform. This will create the now familiar tree structure we have seen with mergesort, as illustrated in Figure 10.9. In this example, the pivot is carried with the left list until the final sorting action. Note that the placement of the pivot between two sublists is the final placement of this number, and thus the number need not be considered further in subsequent sorting actions. We could redraw the tree to show the pivot being withheld. Then the sorted list would be obtained by searching the tree in order, as shown in Figure 10.10.

The fundamental problem with all of these tree constructions is that the initial division is done by a single processor, which will seriously limit the speed. Suppose the pivot selection is ideal and each division creates two sublists of equal size.

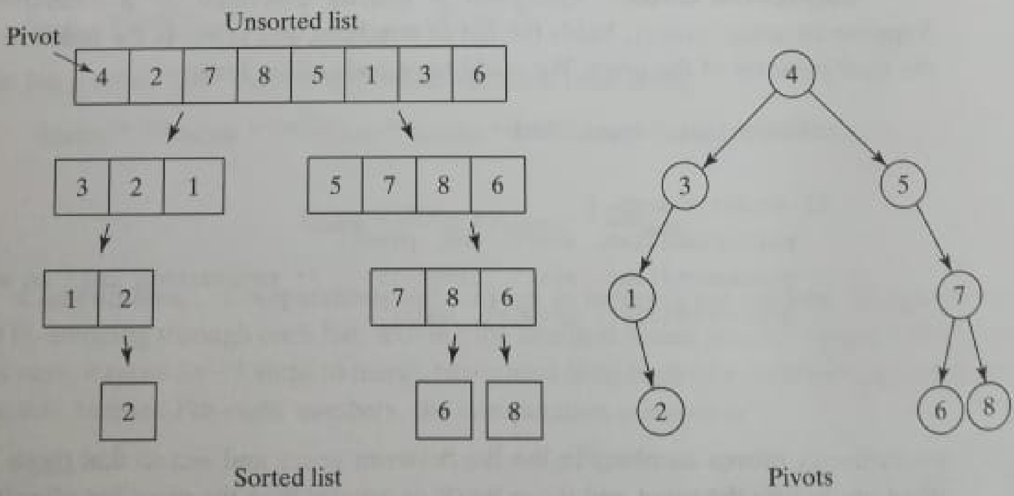


Figure 10.10 Quicksort showing pivot withheld in processes.

Computation. First one processor operates upon n numbers. Then two processors each operate upon $n/2$ numbers. Then four processors each operate upon $n/4$ numbers, and so on:

$$t_{\text{comp}} = n + n/2 + n/4 + n/8 + \ldots \approx 2n$$

Communication. Communication also occurs in a fashion similar to mergesort:

$$t_{\text{comm}} = (t_{\text{startup}} + (n/2)t_{\text{data}}) + (t_{\text{startup}} + (n/4)t_{\text{data}}) + (t_{\text{startup}} + (n/8)t_{\text{data}}) + \cdots$$
$$= (\log p)t_{\text{startup}} + nt_{\text{data}}$$

The analysis so far is the ideal situation. The major difference between quicksort and mergesort is that the tree in quicksort will not, in general, be perfectly balanced. The tree will be unbalanced if the pivots do not divide the lists into equal sublists. The depth of the tree is no longer fixed at $\log n$. The worst-case situation occurs when each pivot selected happens to be the largest of the sublist. The time complexity then degenerates to $O(n^2)$. If we always choose the first number in a sublist as the pivot, the original ordering of the numbers being sorted is the determining factor in the speed of quicksort. Another number could be used as the pivot. In that case, the selection of the pivot is very important to make quicksort operate fast.

Work-Pool Implementation. The load-balancing techniques described in Chapter 7, notably the work pool, can be applied to divide-and-conquer sorting algorithms such as quicksort. The work pool can hold as tasks the sublists to be divided. First, the work pool holds the initial unsorted list, which is given to the first processor. This processor divides the list into two parts. One part is returned to the work pool to be given to another processor, while the other part is operated upon again. This approach is illustrated in Figure 10.11. It does not eliminate the problem of idle processors, but it deals with the case where some sublists are longer and require more work than others.

Quicksort on the hypercube network as a way to achieve better performance will be be explored in Section 10.3.2.

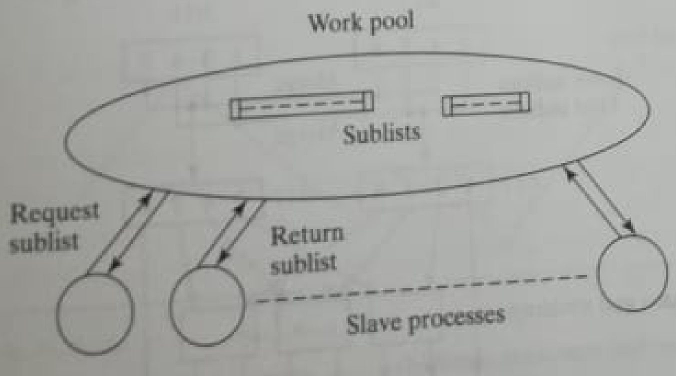


Figure 10.11 Work-pool implementation of quicksort.

10.2.5 Odd-Even Mergesort

The odd-even mergesort algorithm, introduced by Batcher in 1968, is a parallel sorting network based upon his odd-even merge algorithm (Batcher, 1968). The odd-even merge algorithm will merge two *sorted* lists into one sorted list, and this is used recursively to build up larger and larger sorted lists. Given two sorted lists, $a_1, a_2, a_3, \ldots, a_n$ and $b_1, b_2, b_3, \ldots, b_n$ (where n is a power of 2), the following actions are performed:

1. The elements with odd indices of each sequence — that is, $a_1, a_3, a_5, \ldots, a_{n-1}$, and $b_1, b_3, b_5, \ldots, b_{n-1}$ — are merged into one sorted list, $c_1, c_2, c_3, \ldots, c_n$.
2. The elements with even indices of each sequence — that is, $a_2, a_4, a_6, \ldots, a_n$, and $b_2, b_4, b_6, \ldots, b_n$ — are merged into one sorted list, $d_1, d_2, \ldots, d_n$.
3. The final sorted list, $e_1, e_2, \ldots, e_{2n}$, is obtained by the following:

$$e_{2i} = \min\{c_{i+1}, d_i\}$$
$$e_{2i+1} = \max\{c_{i+1}, d_i\}$$

for $1 \le i \le n-1$. Essentially the odd and even index lists are interleaved, and pairs of odd/even elements are interchanged to move the larger toward one end, if necessary. The first number is given by $e_1 = c_1$ (since this will be the smallest of the first elements of each list, a_1 or b_1) and the last number by $e_{2n} = d_n$ (since this will be the largest of the last elements of each list, a_n or b_n).

Batcher (1968) provides a proof of the algorithm. The odd-even merging algorithm is demonstrated in Figure 10.12 for merging two sorted sequences of four numbers into one sorted list of eight numbers. The merging algorithm can also be applied to lists of different lengths, but here we have assumed that the lists are powers of 2.

Each of the original sorted sublists can be created by the same algorithm, as shown in Figure 10.13, and the algorithm can be used recursively, leading to a time complexity of $O(\log^2 n)$ with n processors (Problem 10-15). In this case, pairs of processors perform the compare-and-exchange operations. The whole algorithm can be implemented with hardware units that perform the compare-and-exchange operations, as envisioned by Batcher. It is left as an exercise to draw the final arrangements.

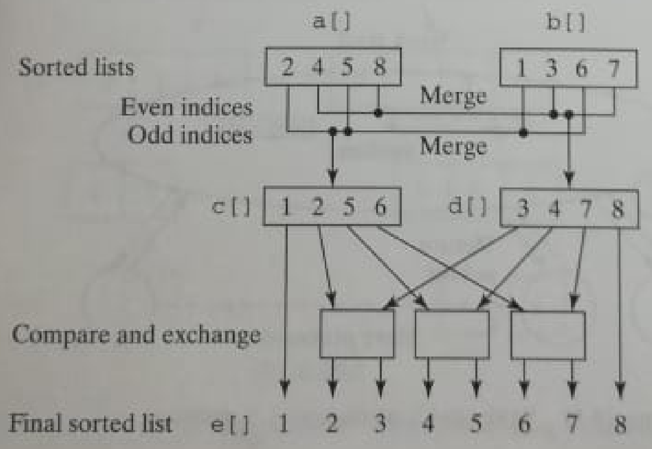


Figure 10.12 Odd-even merging of two sorted lists.

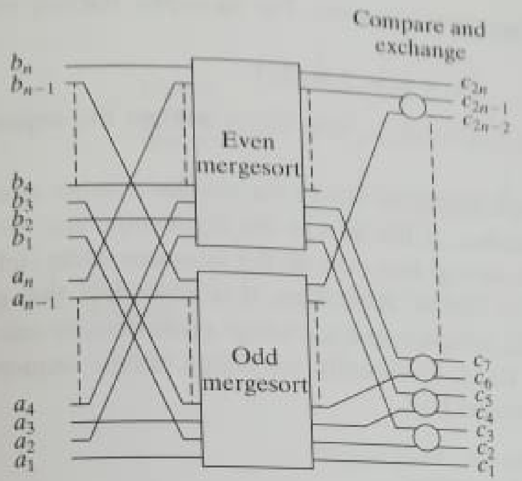


Figure 10.13 Odd-even mergesort.

10.2.6 Bitonic Mergesort

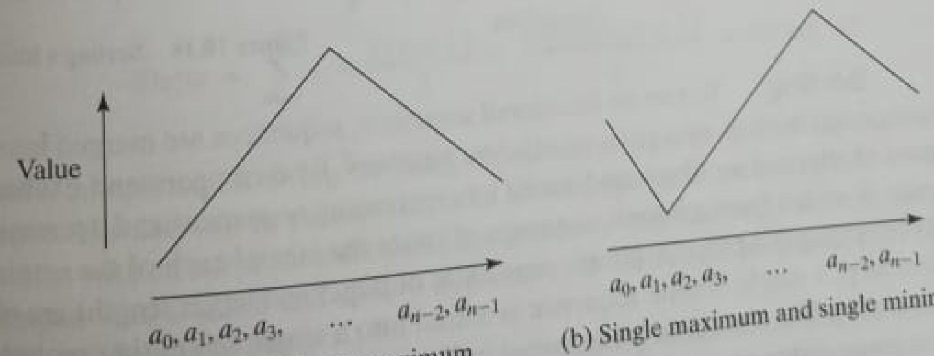
Bitonic mergesort was also introduced by Batcher in 1968 as a parallel sorting algorithm.

Bitonic Sequence. The basis of bitonic mergesort is the *bitonic sequence*, a list having specific properties that will be utilized in the sorting algorithm. A monotonic increasing sequence is a sequence of increasing numbers. A bitonic sequence has two sequences, one increasing and one decreasing. Formally, a bitonic sequence is a sequence of numbers, $a_0, a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1}$, which monotonically increases in value, reaches a single maximum, and then monotonically decreases in value; for example,

$$a_0 < a_1 < a_2 < a_3, \ldots, a_{i-1} < a_i > a_{i+1}, \ldots, a_{n-2} > a_{n-1}$$

for some value of i $(0 \le i < n)$. A sequence is also bitonic if the preceding can be achieved by shifting the numbers cyclically (left or right). Bitonic sequences are illustrated in Figure 10.14. Note that a bitonic sequence can be formed by sorting two lists, one in increasing order and one in decreasing order, and concatenating the sorted lists.

The "special" characteristic of bitonic sequences is that if we perform a compare-and-exchange operation on a_i with $a_{i+n/2}$ for all i $(0 \le i < n/2)$, where there are n numbers in the sequence, we get two bitonic sequences, where the numbers in one sequence are



(a) Single maximum

(b) Single maximum and single minimum

Figure 10.14 Bitonic sequences.

all less than the numbers in the other sequence. For example, starting with the bitonic sequence

3, 5, 8, 9, 7, 4, 2, 1

and performing a compare and exchange, a_i with $a_{i+n/2}$, we get the sequences shown in Figure 10.15.

The compare-and-exchange operation moves the smaller number of each pair to the left sequence and the larger number of the pair to the right sequence. Note that all the numbers in the left sequence are indeed less than all the numbers in the right sequence, in addition to both sequences being bitonic sequences. It is now clear that given a bitonic sequence, recursively performing compare-and-exchange operations to subsequences will sort the list, as shown in Figure 10.16. Eventually, we obtain bitonic sequences consisting of one number each and a fully sorted list.

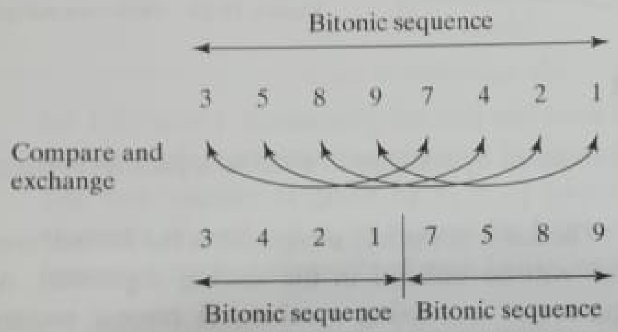


Figure 10.15 Creating two bitonic sequences from one bitonic sequence.

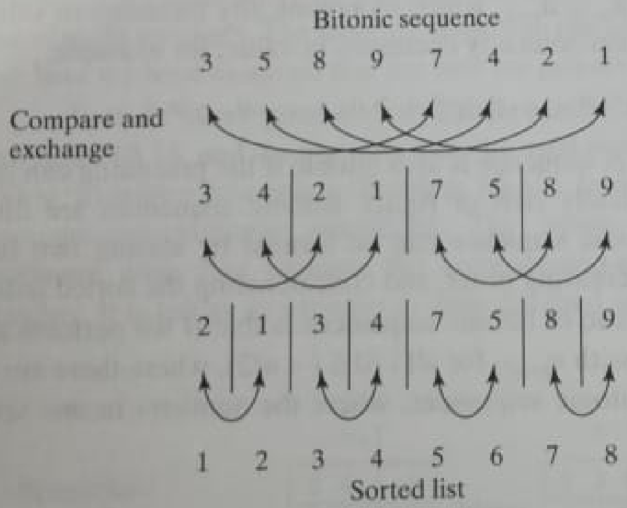


Figure 10.16 Sorting a bitonic sequence.

Sorting. To sort an unordered sequence, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers. By a compare-and-exchange operation, pairs of adjacent numbers are formed into increasing sequences and decreasing sequences, pairs of which form a bitonic sequence of twice the size of each of the original sequences. By repeating this process, bitonic sequences of larger and larger lengths are obtained. In the final step, a single bitonic sequence is sorted into a single increasing sequence (the sorted sequence). The algorithm is illustrated in Figure 10.17. Compare-and-exchange operations can create either an increasing sequence or a decreasing sequence, and alternate directions

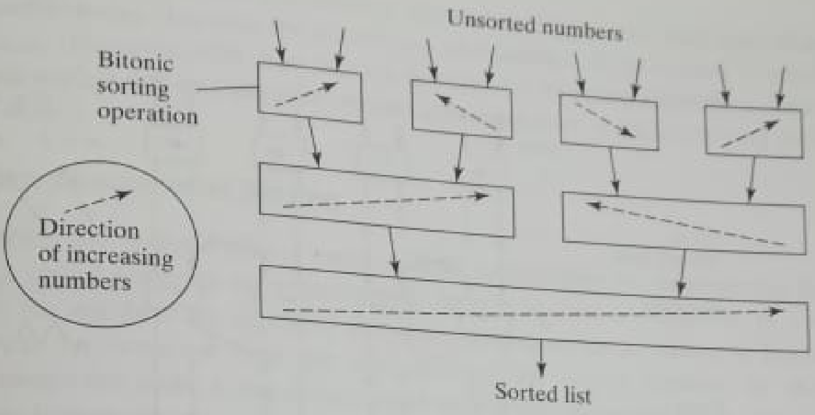


Figure 10.17 Bitonic mergesort.

can form a bitonic sequence with a maximum at the center of the sequence. In Figure 10.17 the first bitonic sequence at the right has a single minimum at the center; the left bitonic sequence has a single maximum at the center. (The algorithm could also work if both sides created bitonic sequences with a maximum at the center.)

First let us expand the algorithm into the basic operations using a numerical example. Figure 10.18 shows bitonic mergesort as applied to sorting eight numbers. The basic compare-and-exchange operation is given by a box, with an arrow indicating which output is the larger number of the operation. The six steps (for eight numbers) are divided into three phases:

Phase 1 (Step 1)	Convert pairs of numbers into increasing/decreasing sequences and hence into 4-bit bitonic sequences.
Phase 2 (Steps 2/3)	Split each 4-bit bitonic sequence into two 2-bit bitonic sequences, higher sequences at the center.
	Sort each 4-bit bitonic sequence into increasing/decreasing sequences and merge into an 8-bit bitonic sequence.
Phase 3 (Steps 4/5/6)	Sort 8-bit bitonic sequence (as in Figure 10.17).

In general, with $n = 2^k$, there are k phases, each of 1, 2, 3, …, k steps. Therefore the total number of steps is given by

$$\text{Steps} = \sum_{i=1}^{k} i = \frac{k(k+1)}{2} = \frac{\log n(\log n + 1)}{2} = \text{O}(\log^2 n)$$

The time complexity of $\text{O}(\log^2 n)$ using n processors (one processor for each number) is attractive. The data can be partitioned, as in all sorting algorithms, to reduce the number of processors with an attendant increase in the number of internal steps. Bitonic mergesort can be mapped onto a mesh and onto a hypercube, as described in Quinn (1994). See also Nassimi and Sahni (1979).

The algorithm is also attractive for implementing in hardware (i.e., as a logic circuit that accepts numbers and sorts them, each compare-and-exchange operation being one

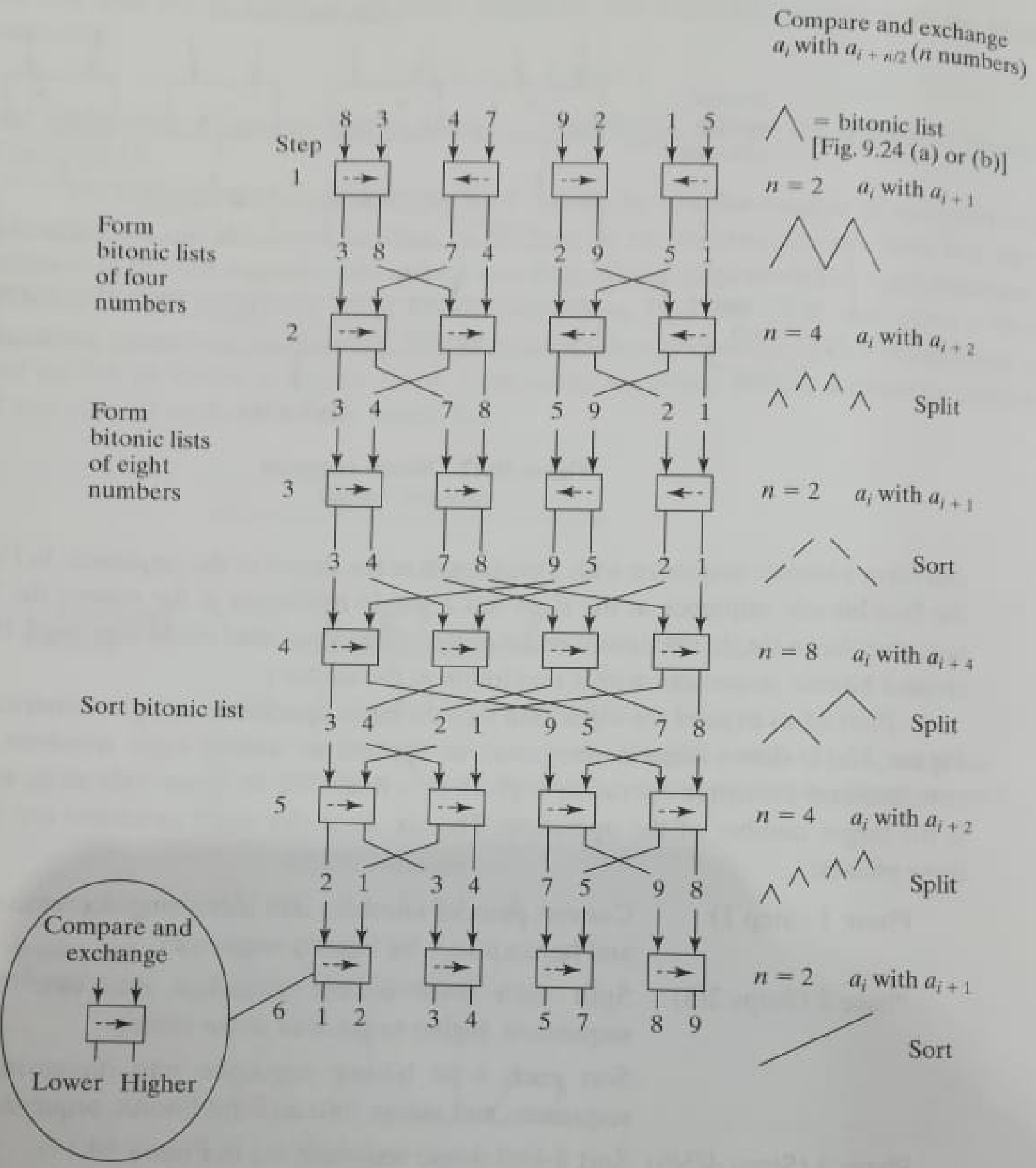

Compare and exchange a_i with $a_{i+n/2}$ (n numbers)
= bitonic list [Fig. 9.24 (a) or (b)]
Step
Form bitonic lists of four numbers
Form bitonic lists of eight numbers
Sort bitonic list
Compare and exchange
Lower Higher
$n = 2$ a_i with a_{i+1}
$n = 4$ a_i with a_{i+2}
Split
$n = 2$ a_i with a_{i+1}
Sort
$n = 8$ a_i with a_{i+4}
Split
$n = 4$ a_i with a_{i+2}
Split
$n = 2$ a_i with a_{i+1}
Sort

Figure 10.18 Bitonic mergesort on eight numbers.

comparator logic circuit) and is usually described in this context. Each step requires $n/2$ two-input/two-output comparators. The "wiring" in Figure 10.18 can be drawn more directly but is given as shown to clarify the position of numbers that are being compared and their subsequent positions.

10.3 SORTING ON SPECIFIC NETWORKS

Algorithms can take advantage of the underlying interconnection network of the parallel computer. Two network structures have received considerable attention over the years, the mesh and the hypercube, because parallel computers have been built with these networks.

We will describe a couple of representative algorithms. In general, such algorithms are of less interest nowadays because the underlying architecture of the system is often hidden from the user. (However, MPI does provide features for mapping algorithms onto meshes, and one can always use a mesh or hypercube algorithm even if the underlying architecture is not the same.)

10.3.1 Two-Dimensional Sorting

If the numbers are mapped onto a mesh, distinct possibilities exist for sorting them. The layout of a sorted sequence on a mesh could be row by row or *snakelike*. In a snakelike sorted list, the numbers are arranged in nondecreasing order, as shown in Figure 10.19. Numbers can be extracted from the node holding the largest numbers by shifting the numbers toward this node. A one-dimensional sorting algorithm that maps onto a line, such as odd-even transposition sort, could be applied to the numbers leading to an O(n) sorting algorithm on the mesh, but this is not cost-optimal and does not take full advantage of the mesh configuration. The actual lower bound for any sorting algorithm on a $\sqrt{n} \times \sqrt{n}$ mesh is $2(\sqrt{n} - 1)$ steps or O($\sqrt{n}$), since in the worst case this number of steps is required to reposition a number. Note that the diameter of the network is $2(\sqrt{n} - 1)$.

Scherson, Sen, and Shamir (1986) describe an ingenious sorting algorithm for a mesh architecture called *shearsort*, which requires $\sqrt{n}(\log n + 1)$ steps for n numbers on a $\sqrt{n} \times \sqrt{n}$ mesh. The algorithm is also described in detail in Leighton (1992). First the numbers are mapped onto the mesh. Then a series of phases are performed (1, 2, 3, ...). In odd phases (1, 3, 5, ...), the following actions are done:

Each row of numbers is sorted independently, in alternative directions:

Even rows — The smallest number of each column is placed at the rightmost end and the largest number at the leftmost end.
Odd rows — The smallest number of each column is placed at the leftmost end and the largest number at the rightmost end.

In even phases (2, 4, 6, ...), the following actions are done:

Each column of numbers is sorted independently, placing the smallest number of each column at the top and the largest number at the bottom.

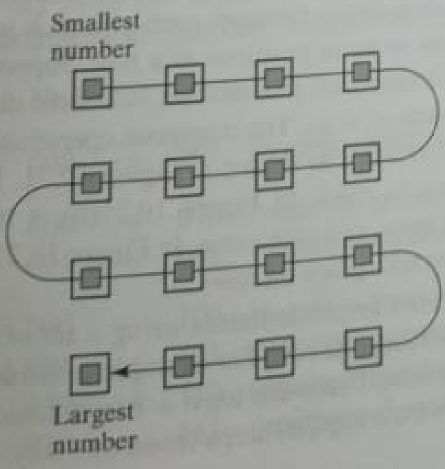
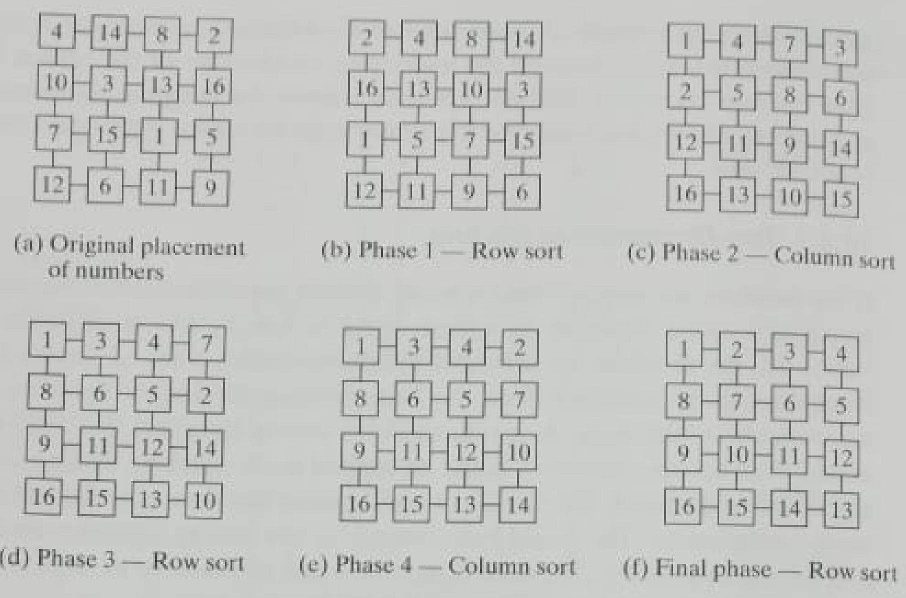


Figure 10.19 Snakelike sorted list.


(a) Original placement of numbers (b) Phase 1 — Row sort (c) Phase 2 — Column sort

(d) Phase 3 — Row sort (e) Phase 4 — Column sort (f) Final phase — Row sort

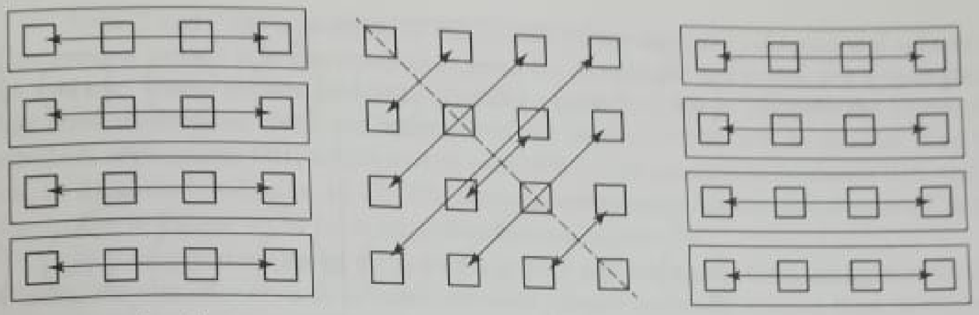
Figure 10.20 Shearsort.

After $\log n + 1$ phases, the numbers are sorted with a snakelike placement in the mesh. (Note the alternating directions of the row-sorting phase, which matches the final snakelike layout.) The algorithm is demonstrated in Figure 10.20 for 16 numbers placed on a 4×4 mesh. The proof is given in Scherson, Sen, and Shamir (1986) and Leighton (1992). Sorting columns and rows can use any sorting algorithm, including odd-even sort. If odd-even sort were used, sorting either columns or rows of $\sqrt{n}$ numbers would require $\sqrt{n}$ compare-and-exchange steps, and the total number of steps would be $\sqrt{n}(\log n + 1)$.

Apart from shearsort, other algorithms have been specifically devised for mapping onto a mesh, as described by Gu and Gu (1994). Existing sorting algorithms can be modified to be mapped onto a mesh. Thompson and Kung (1977) also present sorting algorithms that reach the lower bound of $O(\sqrt{n})$ on a $\sqrt{n} \times \sqrt{n}$ mesh.

Using Transposition. The operations in any mesh algorithm, such as shearsort, which alternate between acting within rows and acting within columns, can be limited to rows by transposing the array of data points between each phase. A transpose operation causes the elements in each column to be in positions in a row. Suppose the elements of array are a_{ij} $(0 \leq i < n, 0 \leq j < n)$. Elements are moved from below the diagonal in the array to above the diagonal, so that element $a_{ij} = a_{ji}$. The transpose operation is placed between the row operations and column operations, as shown in Figure 10.21. In Figure 10.21(a), operations occur between elements in rows. In Figure 10.21(b), a transpose operation occurs, moving the columns to the rows, and vice versa. In Figure 10.21(c), the operations that were specified on columns now take place on rows.

Figure 10.21 suggests a parallel implementation using a set of processors, where each processor is responsible for sorting one row. For example, consider a $\sqrt{n} \times \sqrt{n}$ array with $\sqrt{n}$ processors, one for each row. There are $\log n + 1$ iterations. In each iteration, each processor can sort its row in $O(\sqrt{n} \log \sqrt{n})$ steps (even one way, odd the other way,


(a) Operations between elements in rows

(b) Transpose operation

(c) Operations between elements in rows (originally columns)

Figure 10.21 Using the transpose operation to maintain operations in rows.

cording to the algorithm). The transposition can be achieved with $\sqrt{n}(\sqrt{n}-1)$ communications or O(n) communications (Problem 10-7). Note that data is exchanged between pairs of processes, each exchange requiring two communications. A single *all-to-all* routine could reduce this if available (see Chapter 4). Then each row is sorted again in O($\sqrt{n}\log\sqrt{n}$) steps. On a mesh, the overall communication time complexity will be O(n) as it is dominated by the transpose operation. The technique could be used on other structures. For example, each row could be mapped onto one processor of a line structure.

10.3.2 Quicksort on a Hypercube

The hypercube network has structural characteristics that offer scope for implementing efficient divide-and-conquer sorting algorithms, such as quicksort.

Complete List Placed in One Processor. Suppose a list of n numbers is initially placed on one node of a d-dimensional hypercube. The list can be divided into two parts according to the quicksort algorithm by using a pivot determined by the processor, with one part sent to the adjacent node in the highest dimension. Then the two nodes can repeat the process, dividing their lists into two parts using locally selected pivots. One part is sent to a node in the next-highest dimension. This process is continued for d steps in total, so that every node has a part of the list. For a three-dimensional hypercube with the numbers originally in node 000, we have the splits

	Node		Node	
1st step:	000	→	001	(numbers greater than a pivot, say p_1)
2nd step:	000	→	010	(numbers greater than a pivot, say p_2)
	001	→	011	(numbers greater than a pivot, say p_3)
3rd step:	000	→	100	(numbers greater than a pivot, say p_4)
	001	→	101	(numbers greater than a pivot, say p_5)
	010	→	110	(numbers greater than a pivot, say p_6)
	011	→	111	(numbers greater than a pivot, say p_7)

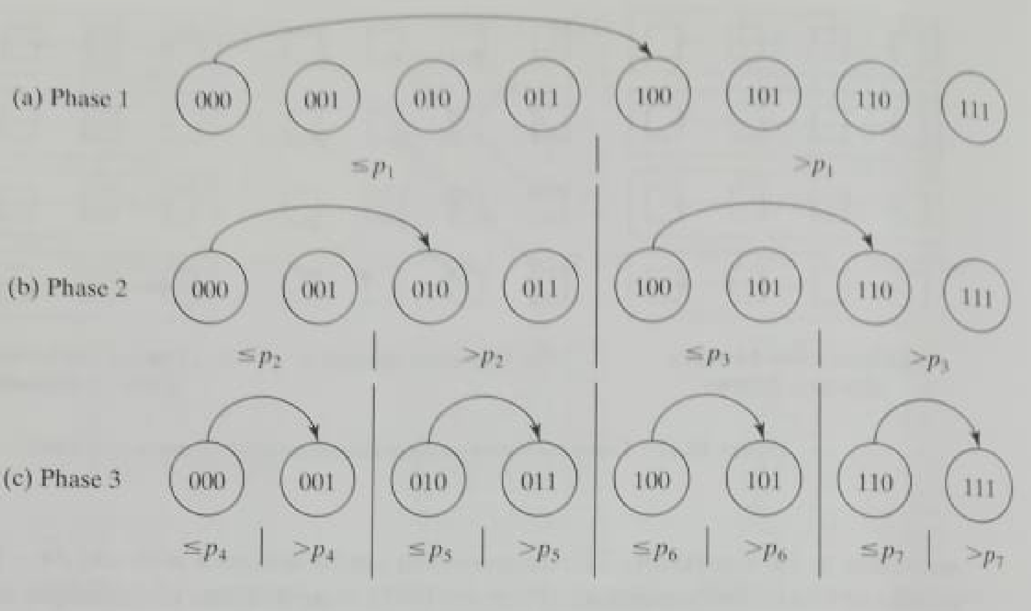


Figure 10.22 Hypercube quicksort algorithm when the numbers are originally in node 000.

The actions are illustrated in Figure 10.22. Finally, the parts can be sorted using a sequential algorithm, all in parallel. If required, the sorted parts can be returned to one processor in a sequence that allows the processor to concatenate the sorted lists to create the final sorted list.

Numbers Initially Distributed Across All Processors. Suppose the unsorted numbers are initially distributed across the nodes in an equitable fashion but not in any special order. A 2^d-node hypercube (d-dimensional hypercube) is composed of two smaller 2^{d-1}-node hypercubes, which are interconnected with links between pairs of nodes in each cube in the dth dimension. These smaller hypercubes can similarly be decomposed into smaller hypercubes (until a single node is reached in each). This feature can be used in a direct extension of the quicksort algorithm to a hypercube as follows. First, consider the hypercube as two subcubes. Processors in the upper subcube have as the most significant bit of their address a 1, and processors in the lower subcube have as the most significant bit of their address a 0. Pairs of processors having addresses 0xxx and 1xxx, where the x's are the same, are "partners." The first steps are

1. One processor (say P_0) selects (or computes) a suitable pivot and broadcasts this to all others in the cube.
2. The processors in the lower subcube send their numbers, which are greater than the pivot, to their partner processor in the upper subcube. The processors in the upper subcube send their numbers, which are equal to or less than the pivot, to their partner processor in the lower cube.
3. Each processor concatenates the list received with what remains of its own list.

Given a d-dimensional hypercube, after these steps the numbers in the lower $(d-1)$-dimensional subcube will all be equal to or less than the pivot, and all the numbers in the upper $(d-1)$-dimensional hypercube will be greater than the pivot.

Steps 2 and 3 are now repeated recursively on the two $(d-1)$-dimensional subcubes. One process in each subcube computes a pivot for its subcube and broadcasts it throughout its subcube. These actions terminate after $\log d$ recursive phases. Suppose the hypercube has three dimensions. Now the numbers in the processor 000 will be smaller than the numbers in processor 001, which will be smaller than the numbers in processor 010, and so on, as illustrated in Figure 10.23. The communication patterns are also illustrated in the hypercube in Figure 10.24 for a three-dimensional hypercube.

To complete the sorting, the numbers in each processor need to be sorted sequentially. Finally, the numbers have to be retrieved from the processors by visiting them in numeric order. Since there is not always a direct link between adjacent processors in numeric order, it may be more convenient to use the construction shown in Figure 10.25, which leaves the sorted numbers in a sequence of processors having increasing Gray code order. It is left as an exercise to investigate larger hypercubes (Problem 10-13).

Pivot Selection. As with all formulations of quicksort, the selection of the pivot is important, and more so for multiprocessor implementations. A poor pivot selection could result in most of the numbers being allocated to a small part of the hypercube, leaving the rest idle. This is most deleterious in the first split. In the sequential quicksort algorithm, often the pivot is simply chosen to be the first number in the list, which could be obtained in a single step or with O(1) time complexity. One approach to improve the pivot selection is to take a sample of a numbers from the list, compute the mean value, and select the median as the pivot. If we were to select the median as the pivot, the numbers sampled

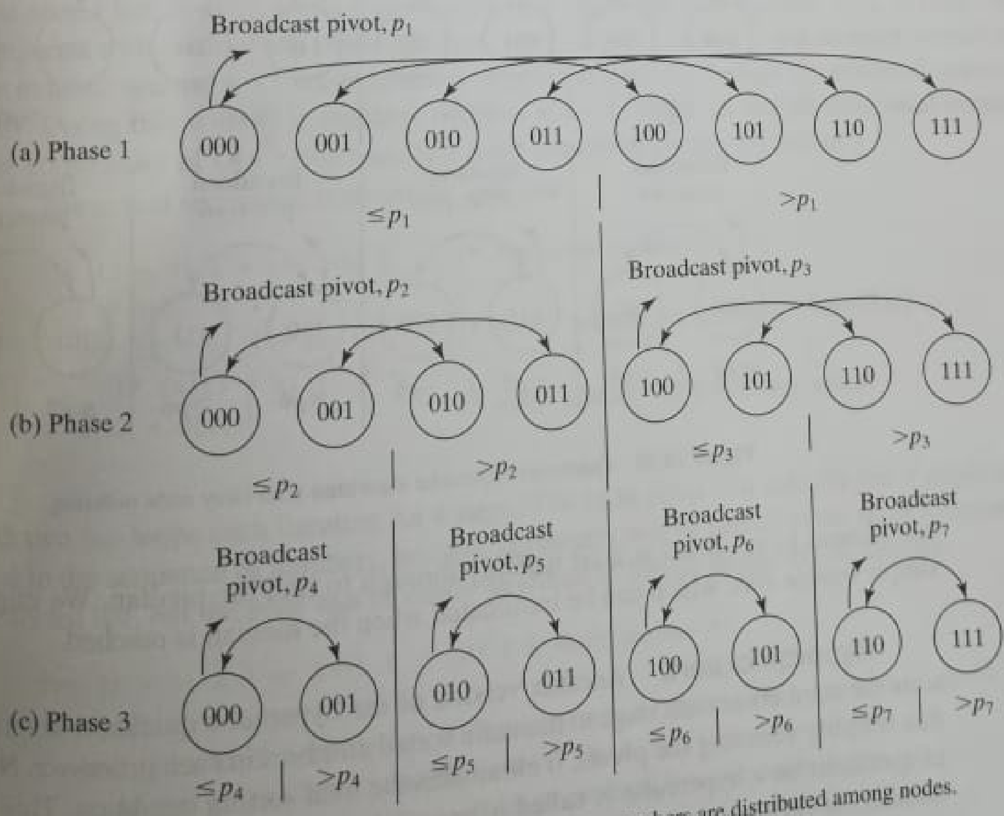
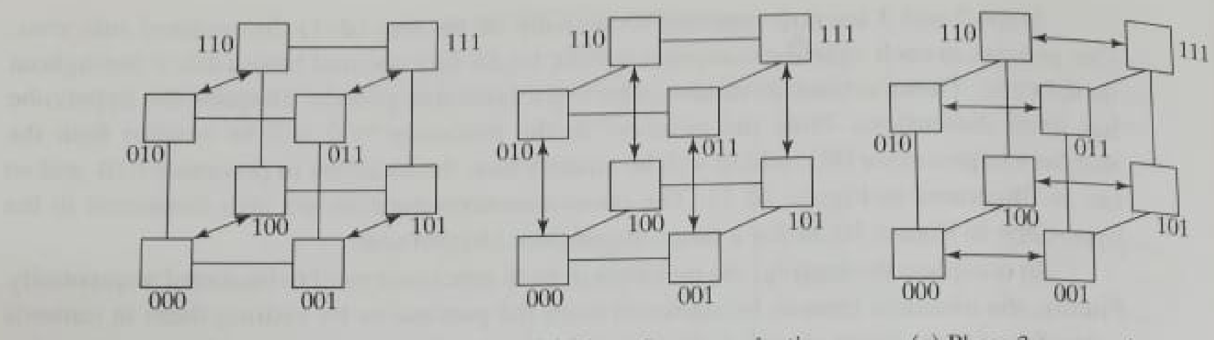


Figure 10.23 Hypercube quicksort algorithm when numbers are distributed among nodes.



110
111
010
011
100
101
000
001
(a) Phase 1 communication
(b) Phase 2 communication
(c) Phase 3 communication

Figure 10.24 Hypercube quicksort communication.

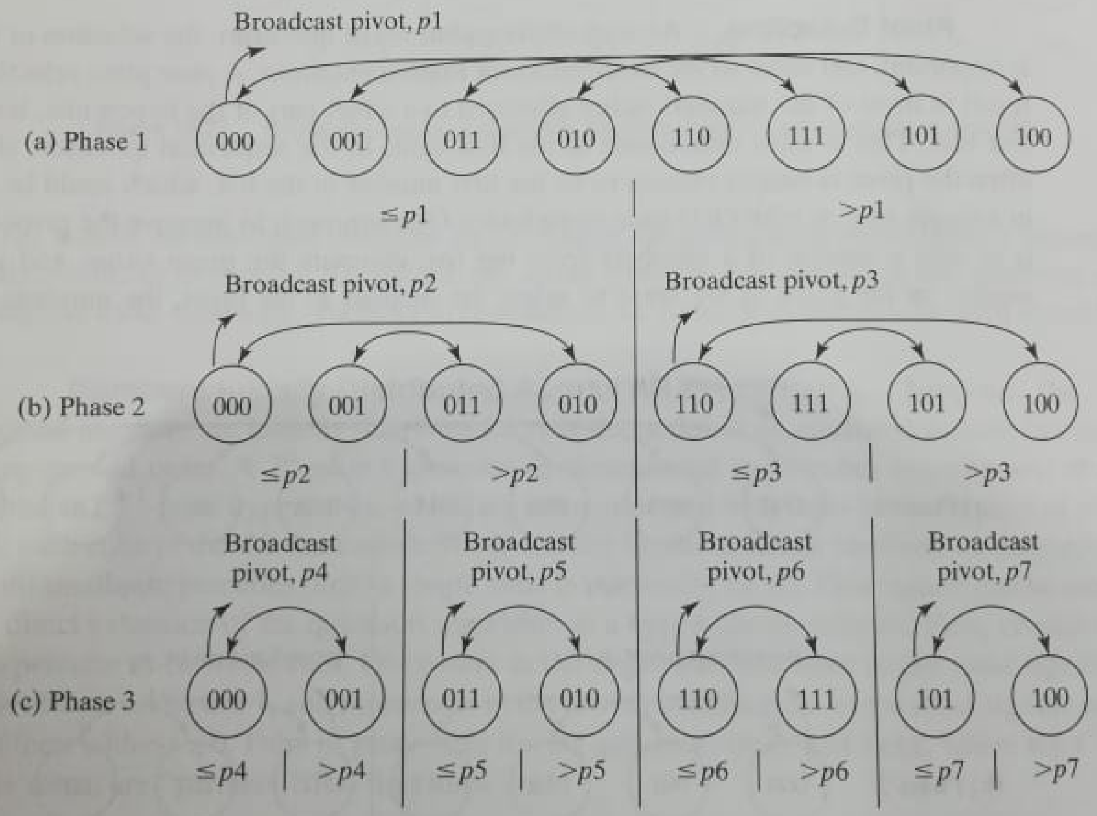


Figure 10.25 Quicksort hypercube algorithm with Gray code ordering.

would need to be sorted at least halfway through to find the median. We might choose a simple bubble sort, which can be terminated when the median is reached.

Hyperquicksort. Another version of the hypercube quicksort algorithm always sorts the numbers at each stage to maintain sorted numbers in each processor. Not only does this simplify selecting the pivots, it eliminates the final sorting operation. This formulation of quicksort on a hypercube is called *hyperquicksort* (Wagar, 1987) and maintains sorted numbers at each stage.

10.4 OTHER SORTING ALGORITHMS

We began this chapter by giving the lower bound for the time complexity of a sequential sorting algorithm based upon comparisons as $O(n \log n)$ (actually $\Omega(n \log n)$, but we have been using big-O notation throughout). Consequently, the time complexity of a parallel sorting algorithm based upon comparisons is $O((\log n)/p)$ with p processors or $O(\log n)$ with n processors. In fact, there are sorting algorithms that can achieve better than $O(n \log n)$ sequential time complexity and thus are very attractive candidates for parallelization, but they often assume special properties of the numbers being sorted. But first, let us look at one sorting algorithm, rank sort, that does not achieve a sequential time of $O(n \log n)$, but can be parallelized easily to achieve a parallel time of $O(n)$ with n processors and $O(\log n)$ with n^2 processors, and leads us onto linear sequential time algorithms which can be parallelized to achieve $O(\log n)$ parallel time and are attractive algorithms for clusters.

10.4.1 Rank Sort

In rank sort (also known as *enumeration sort*), the number of numbers that are smaller than each selected number is counted. This count provides the position of the selected number in the list; that is, its "rank" in the list. Suppose there are n numbers stored in an array, `a[0]` … `a[n-1]`. First `a[0]` is read and compared with each of the other numbers, `a[1]` … `a[n-1]`, recording the number of numbers less than `a[0]`. Suppose this number is x. This is the index of the location in the final sorted list. The number `a[0]` is copied into the final sorted list, `b[0]` … `b[n-1]`, at location `b[x]`. Then the next number, `a[1]`, is read and compared with each of the other numbers, `a[0]`, `a[2]`, …, `a[n-1]`, and so on. Comparing one number against $n - 1$ other numbers requires at least $n - 1$ steps if performed sequentially. Doing this with all n numbers requires $n(n - 1)$ steps, an overall sequential sorting time complexity of $O(n^2)$ (not exactly a good sequential sorting algorithm!).

The actual sequential code might look like

```
for (i = 0; i < n; i++) {        /* for each number */
   x = 0;
   for (j = 0; j < n; j++)       /* count number of numbers less than it */
      if (a[i] > a[j]) x++;
   b[x] = a[i];                  /* copy number into correct place */
}
```

with two `for` loops each iterating for n steps. The code given will actually fail if duplicates exist in the sequence of numbers, because duplicates will be placed in the same location in the sorted list, but the code can be easily modified to handle duplicates as follows:

```
for (i = 0; i < n; i++) {        /* for each number */
   x = 0;
   for (j = 0; j < n; j++)       /* count number of numbers less than it */
      if (a[i] > a[j] || (a[i] == a[j] && j < i)) x++;
   b[x] = a[i];                  /* copy number into correct place */
}
```

This code places duplicates in the same order as in the original sequence. Sorting algorithms that place duplicates in the same order as they were in the original sequence are called *stable sorting algorithms*. In the following, for simplicity, we will omit the modification of the code to handle duplicates. We should also point out that rank sort can be coded in a fashion that achieves O(n) sequential time complexity if the numbers are only integers. The code requires an additional array for each possible value of the numbers. We shall look at this implementation under the name of *counting sort* in Section 10.4.2.

Using n Processors. Suppose we have n processors. One processor would be allocated to one of the numbers and could find the final index of one number in O(n) steps. With all the processors operating in parallel, the parallel time complexity would also be O(n). In `forall` notation, the code would look like

```
forall (i = 0; i < n; i++) {          /* for each number in parallel*/
  x = 0;
  for (j = 0; j < n; j++)             /* count number of nos less than it */
    if (a[i] > a[j]) x++;
  b[x] = a[i];                        /* copy number into correct place */
}
```

The linear parallel time complexity, O(n), is good, but we can do even better if we have more processors.

Using n^2 Processors. Comparing one selected number with each of the other numbers in the list can be performed using multiple processors. For example, the structure shown in Figure 10.26 could be used. Here, $n - 1$ processors are used to find the rank of one number, and with n numbers, $(n - 1)n$ processors or (almost) n^2 processors are needed. A single counter is needed for each number. Incrementing the counter is done sequentially in Figure 10.26 and requires a maximum of n steps (including one step to initialize the counter). Therefore, the total number of steps would be given by $1 + n$ (the 1 for the processors in parallel). A tree structure could be used to reduce the number of steps involved in incrementing the counter, as shown in Figure 10.27. This leads to an O($\log n$) algorithm with n^2 processors for sorting numbers. The actual processor efficiency of this method is

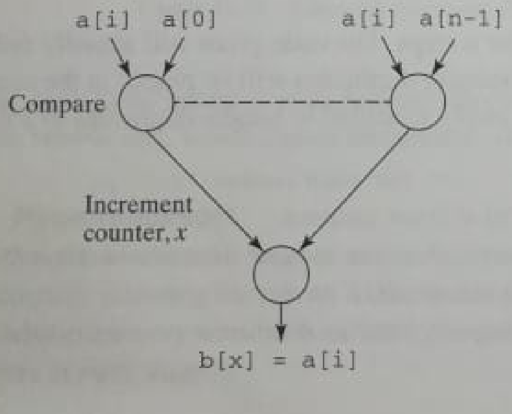


Figure 10.26 Finding the rank in parallel.

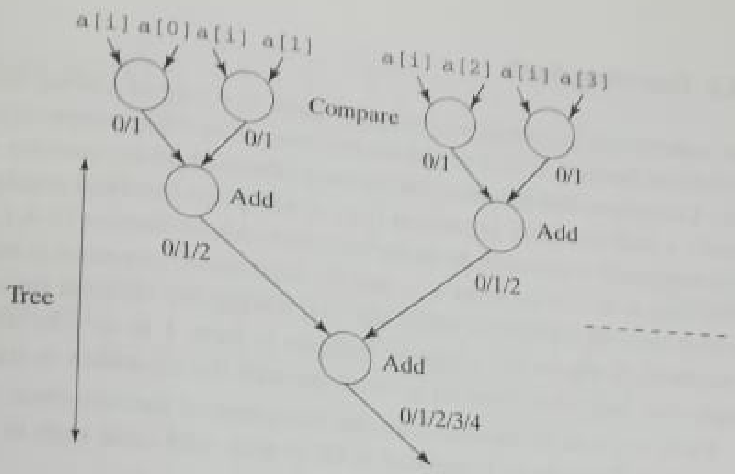


Figure 10.27 Parallelizing the rank computation.

relatively low (again left as an exercise to determine). In theoretical models of parallel computations, only one step is needed to perform the increment operations and combine their results. In these models, it is possible to reduce the time complexity of rank sort to O(1). O(1) is, of course, the lower bound for any problem. However, we do not explore theoretical models here.

Rank sort, then, can sort in $O(n)$ with n processors or in $O(\log n)$ using n^2 processors. In practical applications, using n^2 processors will be prohibitive. The algorithm requires shared access to the list of numbers, making the algorithm most suitable to a shared memory system. The algorithm could be implemented with message-passing, in which a master process responds to requests for numbers from slaves, as shown in Figure 10.28.

Of course, we could reduce the number of processors by partitioning the numbers into groups, say m numbers in each group. Then only n/m processors would be needed to rank each group of numbers (without parallelizing the comparison operation). The number of operations that each processor performs would increase by a factor of m. Such data partitioning applies to many sorting algorithms and is common because the number of numbers, n, is usually much larger than the number of processors available.

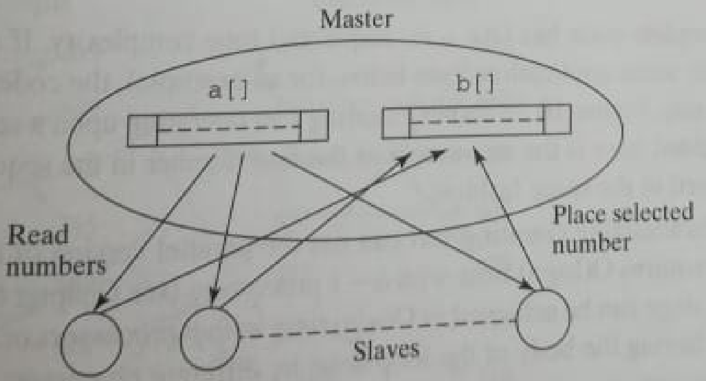


Figure 10.28 Rank sort using a master and slaves.

10.4.2 Counting Sort

If the numbers to be sorted are integers, there is a way of coding the rank sort algorithm described in Section 10.1.3 to reduce the sequential time complexity from $O(n^2)$ to $O(n)$. Coren, Leiserson, and Rivest (1990) refer to the method as *counting sort*. Counting sort is naturally a stable sorting algorithm (i.e., it will place identical numbers in the same order as in the original sequence). As in the rank sort code in Section 10.4.1, the original unsorted numbers are stored in an array `a[]`, and the final sorted sequence is stored in array `b[]`. The algorithm uses an additional array, say `c[]`, having one element for each possible value of the numbers. Suppose the range of integers is from 1 to m. The array has element `c[1]` through `c[m]` inclusive. Now let us work through the algorithm in stages.

First, `c[]` will be used to hold the histogram of the sequence, that is, the number of each number. This can be computed in $O(m)$ time with code such as:

```
for (i = 1; i <= m; i++)
   c[i] = 0;
for (i = 1; i <= m; i++)
   c[a[i]]++;
```

In the next stage of the algorithm, the number of numbers less than each number is found by preforming a prefix sum operation on array `c[]`. In the prefix sum calculation, given a list of numbers, $x_0, \ldots, x_{n-1}$, all the partial summations (i.e., x_0; $x_0 + x_1$; $x_0 + x_1 + x_2$; $x_0 + x_1 + x_2 + x_3$; …) are computed, as first mentioned in Chapter 6, Section 6.2.1. Now, though, the prefix sum is computed using the histogram originally held in `c[]` in $O(m)$ time, as described below:

```
for (i = 2; i <= m; i++)
   c[i] = c[i] + c[i-1];
```

In the final stage of the algorithm, the numbers are placed in the sorted order in $O(n)$ time, as described below:

```
for (i = 1; i <= n; i++) {
   b[c[a[i]]] = a[i];
   c[a[i]]--;                  // done to ensure stable sorting
}
```

The complete code has $O(n + m)$ sequential time complexity. If m is linearly related to n, as it is in some applications (see below for an example), the code has $O(n)$ sequential time complexity. Figure 10.29 shows counting sort operating upon a sequence of eight numbers. Highlighted here is the movement of the first number in the sequence. The other numbers are moved in the same fashion.

Parallelizing counting sort can use the parallel version of the prefix sum calculation which requires $O(\log n)$ time with $n - 1$ processors (see Chapter 6, Section 6.2.2). The final sorting stage can be achieved in $O(n/p)$ time with p processors or $O(1)$ with n processors by simply having the body of the loop done by different processors.

Counting sort is a very efficient algorithm when there are few different numbers and they are all integers. In the next section, we shall use counting sort in the radix sort algorithm.

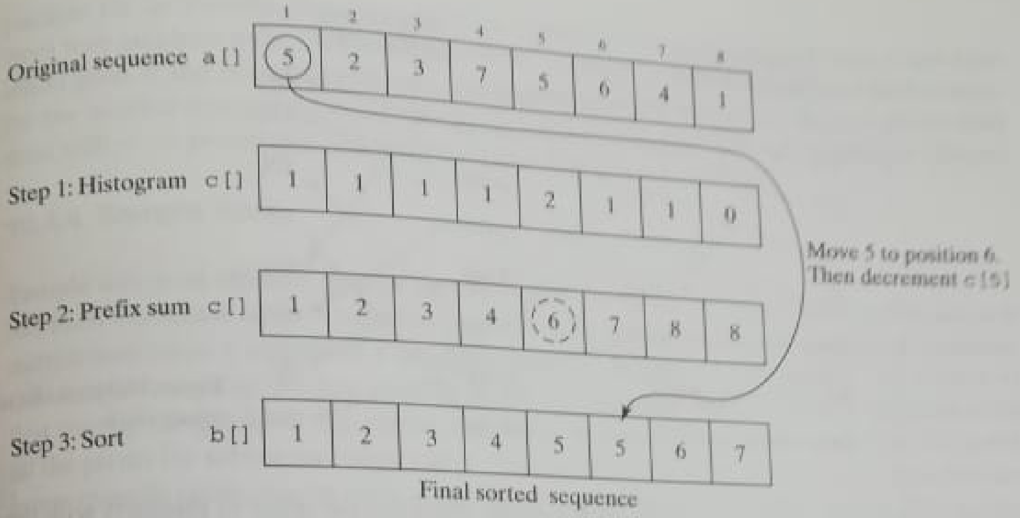


Figure 10.29 Counting sort.

10.4.3 Radix Sort

Radix sort assumes that the numbers to be sorted are represented in a positional digit representation, such as binary and decimal numbers. The digits represent values, and the position of each digit indicates its relative weighting (i.e., what to multiple the value by, the base 10 in decimal, the base 2 in binary). The positions are ordered from the least significant digit through to the most significant digit. Radix sort starts at the least significant digit (rather than the more intuitive method of the most significant digit) and sorts the numbers according to their least significant digits. The sequence is then sorted according to the next least significant digit, and so on until the most significant digit, after which the sequence is sorted. For this to work, it is necessary to maintain the order of numbers with the same digit, that is, one must use a stable sorting algorithm.

Radix sort can sort on individual digits of the number or on groups of digits. Figure 10.30 shows radix sort sorting on individual decimal digits. A example of sorting on individual bits of binary numbers is shown in Figure 10.31. In this case, it requires

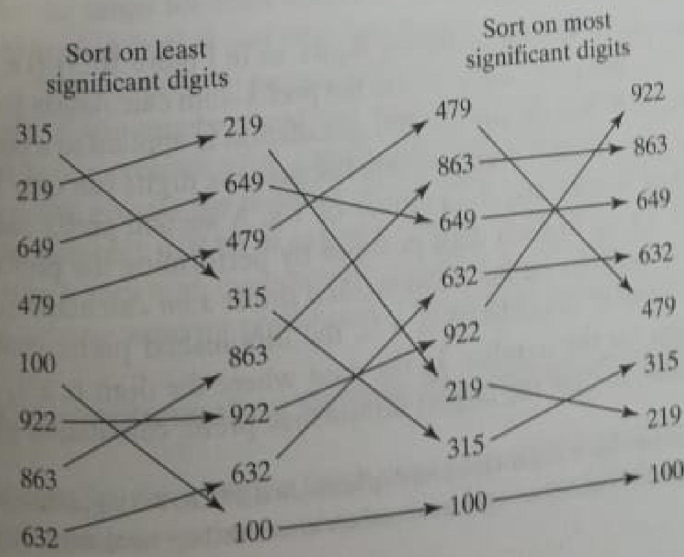


Figure 10.30 Radix sort using decimal digits.

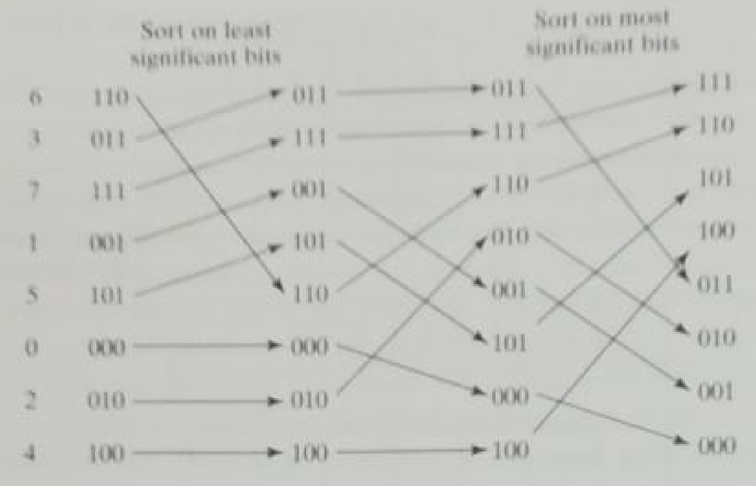


Figure 10.31 Radix sort using binary digits.

b phases if there are b bits in each number. Note that the order of numbers with the same bit value (0 or 1 in this example) is maintained. Radix sort would normally be applied to groups of binary digits at each stage rather than only considering one bit at each stage. In general, b-bit numbers can be sorted in radix sort by sorting on groups of r digits at a time and there are $\lceil b/r \rceil$ phases in that case. The time complexity of radix sort will depend upon the algorithm for sorting at each pass; the algorithm must be a stable sort. Given the small range of numbers, counting sort is suitable, but it is not the only choice because it does not sort in place (i.e., it requires additional storage for numbers as they are sorted).

Suppose that counting sort is used in radix sort. As mentioned in Section 10.4.2, counting sort has an $O(n + m)$ sequential time complexity operating on n integers each within the range of 1 to m. Given r-bit numbers, the range is potentially 1 to $2^r - 1$. There are $\lceil b/r \rceil$ phases where there are b bits on the complete numbers. Hence the sequential time complexity is $O(b/r(n + 2^r))$. If b and r are constants,[1] the sequential time complexity is $O(n)$; that is, again linear time complexity.

Radix sort can be parallelized by using a parallel sorting algorithm in each phase of sorting on bits or groups of bits. We have already mentioned parallelized counting sort using prefix sum calculation, which leads to $O(\log n)$ time with $n - 1$ processors and constant b and r. Radix sort with counting sort is used by Bader and JáJá (1999) in their study of programming clusters of SMPs.

Consider the example of sorting on binary digits, as in Figure 10.31 (i.e., with $r = 1$). A clever way to parallelize this radix sort is to use the prefix-sum calculation for positioning each number at each stage. When the prefix-sum calculation is applied to a column of bits, it gives the number of 1's up to each digit position because the digits can only be 0 or 1 and the prefix calculation will simply add the number of 1's. A second prefix calculation can also give the number of 0's up to each digit position by performing the prefix calculation on the digits inverted (sometimes called a *diminished prefix-sum* calculation). In the case where the digit of the number considered is a 0, the diminished prefix-sum calculation provides the new position for the number. In the case where the digit is a 1, the result of normal prefix-sum calculation plus the largest diminished prefix calculation gives the final

[1] There is a relationship between n and b. Given no duplicates, $n \le 2^b$, i.e., $b \ge \log_2 n$. However, computers normally operate with a fixed number of bits to represent numbers no matter how many numbers there are.

position for the number. For example, if the number under consideration has a 1 and there were four numbers with a 0 plus two previous numbers with a 1, diminished prefix calculation gives 4 and normal prefix-sum calculation gives 3, resulting in the seventh position for the number (counting from position 1). Clearly this algorithm again leads to O(log$n$) time with $n - 1$ processors, but it does require b phases as $r = 1$.

10.4.4 Sample Sort

Sample sort is an old idea (Frazer and McKellar, 1970), as are many basic sorting ideas. It has been discussed in the context of quicksort and bucket sort. In the context of quicksort, sample sort takes a sample of s numbers from the sequence of n numbers. The median of this sample is used as the first pivot to divide the sequence into two parts, as required as the first step by the quicksort algorithm, rather than the usual first number in the list. For speed, all the pivots for subsequent steps can be found at the same time. For example, the upper and lower quartile points can be used as pivots for the first two subsequent sublists. This method addresses the problem of badly selected pivots in quicksort unevenly dividing the lists.

Sample sort has also been applied to bucket sort, as described in Chapter 4, Section 4.2. The fundamental problem in bucket sort is the same as in quicksort: unevenly divided lists. In bucket sort, if the numbers are not equally distributed, more numbers will fall into some buckets than others. In the worst case, all the numbers fall into one bucket, reducing the potential linear sequential time complexity to that of the comparison sorting algorithm used to sort n numbers (i.e., O($n$log$n$)). The basic problem is that the range of numbers for each bucket is fixed and chosen by dividing the complete range of numbers into equal regions.

The objective of sample sort is to divide the ranges so that each bucket will have approximately the same number of numbers. It does this by using a sampling scheme which picks out numbers from the sequence of n numbers as splitters that define the range of numbers for each bucket. If there are m buckets, $m - 1$ splitters are needed. These can be found by the following method. The numbers to be sorted are first divided into n/m groups. Each group is sorted, and a sample of s equally spaced numbers is chosen from each group. This creates ms samples in total, which are then sorted, and $m - 1$ equally spaced numbers are selected as splitters. The method for selecting the splitters is shown in Figure 10.32. After the range for each bucket is set by the splitters, the algorithm continues in the same fashion as in bucket sort (the buckets are sorted and the results concatenated).

The method can be parallelized in much the same way as bucket sort, by assigning one group of n/m numbers and one bucket to each processor ($p = m$). The mp numbers can be sent to one processor for sorting, although there are other possibilities. The larger the value for s, the greater the number of numbers used in the final splitter. If $s = m - 1$, less than $2n/m$ numbers will be in any one bucket. There are variations in the sample sort described, in which very high oversampling is done to take advantage of regular collective operations and achieve superior performance on clusters (see Helman, Bader, and JáJá, 1998).

10.4.5 Implementing Sorting Algorithms on Clusters

Efficient implementation on clusters calls for the use of broadcast and other collective operations, such as gather, scatter, and reduce, provided in message-passing software such as

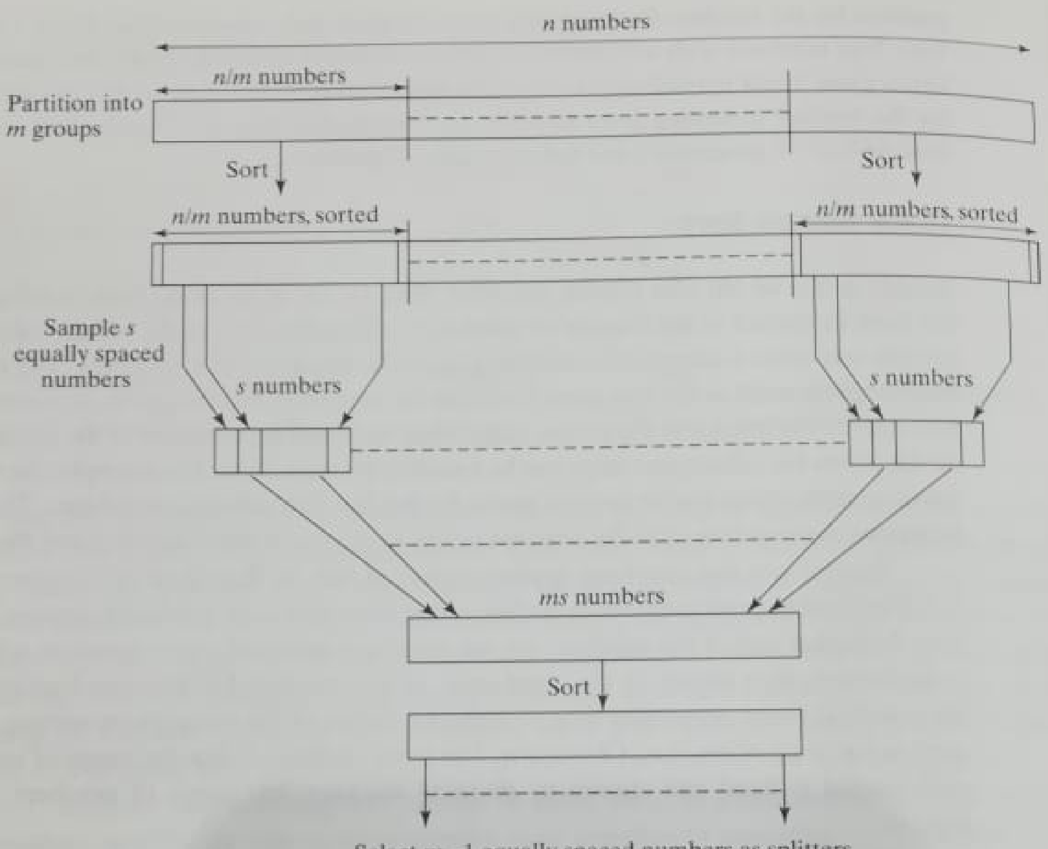


Figure 10.32 Selecting spliters in sample sort version of bucket sort.

MPI, rather than non-uniform communication patterns that require point-to-point communication, because collective operations are expected to be implemented efficiently. The distributed memory of a cluster does not favor algorithms requiring access to widely separated numbers. Algorithms that require only local operations are better, although in the worst case, all sorting algorithms finally have to move numbers from one end of the sequence to the other.

Processors always have cache memory, and it is better to have an algorithm that operates upon a block of numbers that can be placed in the cache. In consequence, one will need to know the size and organization of the cache, and this has to become part of the algorithm as parameters. Finally, with the advent of clusters of SMP processors (SMP clusters), algorithms need to take into account that the groups of processors in each SMP system may operate in a shared memory mode where the shared memory is only within each SMP system, whereas each system may communicate with other SMP systems in the cluster in a message-passing mode. Taking this into account again requires parameters such as number of processors within each SMP system and size of the memory in each SMP system. More information on algorithms on SMP systems can be found in Bader and JáJá (1999).

10.5 SUMMARY

This chapter described several ways that sorting can be done with multiprocessors. We began by stating that the lower bound of comparison-based sorting algorithms implemented sequentially is O($n \log n$) and for parallel implementation is O($\log n$) with n processors. We then considered compare-and exchange operations and sorting algorithms:

- Bubble sort
- Odd-even transposition sort
- Mergesort
- Quicksort, including on a hypercube
- Odd-even mergesort
- Bitonic mergesort

We then briefly looked at representative sorting algorithms that take advantage of specific interconnection networks:

- Shearsort (on a mesh)
- Quicksort on a hypercube

Finally, we explored sorting algorithms that do not use compare-and-exchange operations:

- Rank sort
- Counting sort, for integers only

and algorithms that obtain superior parallel performance when parallelized:

- Radix sort
- Sample sort

FURTHER READING

There are very many sorting algorithms and variations not covered in this chapter. We have selected those which are most common, representative of the technique, or have the potential for parallelization. Richards (1986) provides a list of 373 papers on parallel sorting up to the early 1980s. Akl (1985) has also written a book devoted entirely to parallel sorting algorithms. Subsequent papers proposing sorting algorithms include Blackston and Ranade (1993) and Bolorforoush et al. (1992). Survey papers have been published on parallel sorting, notably Bitton et al. (1984) and Lakshmivarahan, Dhall, and Miller (1984), the latter concentrating totally upon sorting networks.

In this second edition, we have reordered the material and added two well-known sorting algorithms, radix sort and sample sort, because of the attention recently given them as the basis for sorting on clusters and, in particular, SMP clusters, although the full details of cluster implementation are omitted because of their complexity. Further details can be found in Helman, Bader, and JáJá (1998). Sample sort is described by Shi and Schaeffer (1992).

BIBLIOGRAPHY

AJTAI, M., J. KOMLÓS, AND E. SZEMERÉDI (1983), "An $O(n \log n)$ Sorting Network," *Proc. 15th Annual ACM Symp. Theory of Computing*, Boston, MA, pp. 1–9.

AKL, S. (1985), *Parallel Sorting Algorithms*, Academic Press, New York.

BADER, D. A., AND J. JAJA (1999), "SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs), *J. Par. Dist Computing*, Vol. 58, pp. 92–108.

BATCHER, K. E. (1968), "Sorting Networks and Their Applications," *Proc. AFIPS Spring Joint Computer Conference*, Vol. 32, AFIPS Press, Reston, VA, pp. 307–314.

BITTON, D., D. J. DEWITT, D. K. HSIAO, AND J. MENON (1984), "A Taxonomy of Parallel Sorting," *Computing Surveys*, Vol. 16, No. 3 (September), pp. 287–318.

BLACKSTON, D. T., AND A. RANADE (1993), "SnakeSort: A Family of Simple Optimal Randomized Sorting Algorithms," *Proc. 1993 Int. Conf. Par. Proc.*, Vol. 3, pp. 201–204.

BOLORFOROUSH, M., N. S. COLEMAN, D. J. QUAMMEN, AND P. Y. WANG (1992), "A Parallel Randomized Sorting Algorithm," *Proc. 1992 Int. Conf. Par. Proc.*, Vol. 3, pp. 293–296.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

FRAZER, W. D., AND A. C. MCKELLAR (1970), "Samplesort: A sampling approach to minimal storage tree sorting," *J. ACM*, Vol. 17, No. 3 (July), pp. 496–567.

FOX, G., M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER (1988), *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, Englewood Cliffs, NJ.

GU, Q. P., AND J. GU (1994), "Algorithms and Average Time Bounds of Sorting on a Mesh-Connected Computer," *IEEE Trans. Par. Distrib. Syst.*, Vol. 5, No. 3, pp. 308–314.

HELMAN, D. R., D. A. BADER, AND J. JÁJÁ (1998), "A Randomized Parallel Sorting Algorithm with an Experimental Study, *Journal of Parallel and Distributed Computing*, Vol. 52, No. 1, pp. 1–23

HIRSCHBERG, D. S. (1978), "Fast Parallel Sorting Algorithms," *Comm. ACM*, Vol. 21, No. 8, pp. 538–544.

HOARE, C. A. R. (1962), "Quicksort," *Computer Journal*, Vol. 5, No. 1, pp. 10–15.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA.

LAKSHMIVARAHAN, S., S. K. DHALL, AND L. L. MILLER (1984), "Parallel Sorting Algorithms," *Advances in Computers*, Vol. 23, pp. 295–354.

LEIGHTON, F. T. (1984), "Tight Bounds on the Complexity of Parallel Sorting," *Proc. 16th Annual ACM Symposium on Theory of Computing*, ACM, New York, pp. 71–80.

LEIGHTON, F. T. (1992), *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA.

NASSIMI, D., AND S. SAHNI (1979), "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Trans. Comp.*, Vol. C-28, No. 2, pp. 2–7.

QUINN, M. J. (1994), *Parallel Computing Theory and Practice*, McGraw-Hill, NY.

RICHARDS, D. (1986), "Parallel Sorting — A Bibliography," *SIGACT News* (Summer), pp. 28–48.

SCHERSON, I. D., S. SEN, AND A. SHAMIR (1986), "Shear-Sort: A True Two-Dimensional Sorting Technique for VLSI networks," *Proc. 1986 Int. Conf. Par. Proc.*, pp. 903–908.

SHI, H, AND J. SCHAEFFER (1992), "Parallel Sorting by Regular Sampling," *Journal of Parallel and Distributed Computing*, Vol. 14, pp. 361–372.

THOMPSON, C. D., AND H. T. KUNG (1977), "Sorting on a Mesh-Connected Parallel Computer," *Comm. ACM*, Vol. 20 (April), pp. 263–271.

WAGAR, B. (1987), "Hyperquicksort: A Fast Sorting Algorithm for Hypercubes," *Proc. 2nd Conf. Hypercube Multiprocessors*, pp. 292–299.

WAINWRIGHT, R. L. (1985), "A Class of Sorting Algorithms Based on Quicksort," *Comm. ACM*, Vol. 28, No. 4, pp. 396–402. Also see "Technical Correspondence," *Comm. ACM*, Vol. 29, No. 4, pp. 331–335.

PROBLEMS

Scientific/Numerical

10-1. Rewrite the compare-and-exchange code given in Section 10.2.1 to eliminate the message-passing when exchanges are not needed.

10-2. Implement both methods of compare-and-exchange operations described in Section 10.2.1 (i.e., Version 1 and Version 2), operating upon blocks of four numbers. Measure the performance of the methods. Devise an experiment to show that it is possible to obtain erroneous results on a heterogeneous system with computers having different arithmetic precisions (if available).

10-3. Determine the processor efficiency of the O(log n) algorithm based upon the rank sort described in Section 10.1.3 for sorting numbers.

10-4. Fox et al. (1988) present a method of compare and exchange in which individual numbers are exchanged between the processors, say P_0 and P_1. Suppose each processor has four numbers in its group. Processor P_0 sends the largest number in its group to processor P_1, while processor P_1 sends the smallest number of its group to processor P_0. Each processor inserts the received number into its group, so that both still have n/p numbers. The actions are repeated with the new largest and smallest numbers. The algorithm is most conveniently implemented using a queue with a pointer to the top or bottom, as illustrated in Figure 10.33.

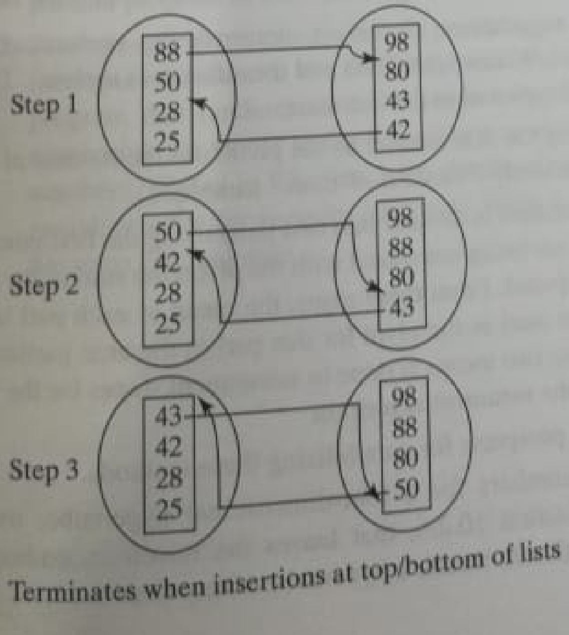


Figure 10.33 Compare-and-exchange algorithm for Problem 10-4.

The algorithm can terminate when the insertions would be at the top and bottom of the groups, as shown. The maximum number of communication steps would be when all the numbers in the left list were initially larger than all of the numbers in the other group (a maximum of x steps with x numbers in each group). Write a parallel program to implement Fox's method and evaluate it compared to the methods described in the text.

10-5. The following is an attempt to code the odd-even transposition sort of Section 10.2.2 as one SPMD program.

Process P_i

```
evenprocess = (i % 2 == 0);
evenphase = 1;
for (step = 0; step < n; step++, evenphase = !evenphase){
  if ((evenphase && evenprocess) || (!evenphase) && !evenprocess)) {
    send(&a, Pi+1);
    recv(&x, Pi+1);
    if (x < a) a = x;              /* keep smaller number */
  } else {
    send(&a, Pi-1);
    recv(&x, Pi-1);
    if (x > a) a = x;              /* keep larger number */
  }
}
```

Determine whether this code is correct, and if not, correct any mistakes.

10-6. The odd-even transposition sort of Section 10.2.2 was described on the assumption of an even number of numbers and processors. What changes to the code would be necessary to allow an odd number of numbers and processors?

10-7. Show that there are $\sqrt{n}(\sqrt{n}-1)$ communications for a transpose operation on a $\sqrt{n} \times \sqrt{n}$ array, as stated in Section 10.3.1.

10-8. Write a parallel program to implement shearsort.

10-9. Write a parallel program to find the kth smallest number of a set of numbers. Use a parallel version of quicksort, but apply it only to the set of numbers containing the kth smallest number. See Cormen, Leiserson, and Rivest (1990) for further discussion on the problem.

10-10. There have been many suggestions of ways to improve the performance of sequential quicksort (see, for example, Wainwright, 1985 and the references therein). The following are two suggestions that have appeared in the literature:

1. Rather than selecting the first number as the pivot, use the median of a group of three numbers picked randomly ("median-of-three" technique).
2. The initial set of numbers is divided into two parts using the first number as the pivot. While the numbers are being compared with the pivot, the sum of the numbers in each of two parts is computed. From these sums, the mean of each part is computed. The mean of each part is used as the pivot for this part in the next partitioning stage. The process of computing two means is done in subsequent stages for the next pivots. This algorithm is called the meansort algorithm.

Determine empirically the prospects for parallelizing these methods.

10-11. Draw the exchanges of numbers for a four-dimensional hypercube, using the parallel hypercube described in Section 10.2.6, that leaves the results in embedded ring order. Determine a general algorithm to handle a hypercube of any size.

10-12. Draw the compare-and-exchange circuit configuration for the odd-even mergesort algorithm described in Section 10.2.7 to sort 16 numbers. Sort the following sequence by hand using the odd-even mergesort algorithm:

12 2 11 4 9 1 10 15 5 7 14 3 8 13 6 16

10-13. Repeat Problem 10-12 for bitonic mergesort.

10-14. Compare Batcher's odd-even mergesort algorithm (Section 10.2.5) and his bitonic mergesort algorithm (Section 10.2.6), and assess their relative advantages for parallel implementation on a message-passing multicomputer.

10-15. Prove that the time complexity of odd-even mergesort is $O(\log^2 n)$ with *n* processors.

10-16. Identify those sorting algorithms in Chapter 10 that operate in place (i.e., do not use additional storage for the numbers as they are sorted. In place algorithms will reduce the storage requirements, and this factor may be important when sorting a large number of numbers.

10-17. Would radix sort work if the sorting operated from the most significant digit to the least significant digit? Discuss.

10-18. Implement two sorting algorithms in MPI and compare their performance.

10-19. Read the paper by Helman, Bader, and JáJá (1998) and implement their sorting algorithm in MPI.

Real Life

10-20. Fred has a deck of 52 playing cards that have been thoroughly shuffled. He has asked you to determine several things related to reordering them:

1. What modification to bitonic mergesort will be needed to sort the cards, given that there are four suits (spades, hearts, clubs, and hearts)?
2. How many friends will Fred have to invite over (and feed) to carry out a modified bitonic mergesort in parallel, and how many steps will each friend have to carry out?

10-21. One way to find matching entries in two files is first to sort the contents of the files and then to step through the files comparing entries. Analyze this method compared to simply comparing each entry of one file with each entry of the second file without sorting. Write a parallel program to find matching entries of two files by first sorting the files.

10-22. Sorting algorithms can create any permutation of an input sequence by simply numbering the input elements in the required order and sorting on these numbers. Write a parallel program that randomizes the contents of a document file for security purposes by first numbering the words in the file using a random-number generator and then sorting on the numbers. (Refer to Chapter 3 for details of random numbers.) Is this a good method of encoding a file for security purposes? Write a parallel program that restores the document file to its original state.

THOMPSON, C. D., AND H. T. KUNG (1977), "Sorting on a Mesh-Connected Parallel Computer," *Comm. ACM*, Vol. 20 (April), pp. 263–271.

WAGAR, B. (1987), "Hyperquicksort: A Fast Sorting Algorithm for Hypercubes," *Proc. 2nd Conf. Hypercube Multiprocessors*, pp. 292–299.

WAINWRIGHT, R. L. (1985), "A Class of Sorting Algorithms Based on Quicksort," *Comm. ACM*, Vol. 28, No. 4, pp. 396–402. Also see "Technical Correspondence," *Comm. ACM*, Vol. 29, No. 4, pp. 331–335.

PROBLEMS

Scientific/Numerical

10-1. Rewrite the compare-and-exchange code given in Section 10.2.1 to eliminate the message-passing when exchanges are not needed.

10-2. Implement both methods of compare-and-exchange operations described in Section 10.2.1 (i.e., Version 1 and Version 2), operating upon blocks of four numbers. Measure the performance of the methods. Devise an experiment to show that it is possible to obtain erroneous results on a heterogeneous system with computers having different arithmetic precisions (if available).

10-3. Determine the processor efficiency of the O(log n) algorithm based upon the rank sort described in Section 10.1.3 for sorting numbers.

10-4. Fox et al. (1988) present a method of compare and exchange in which individual numbers are exchanged between the processors, say P_0 and P_1. Suppose each processor has four numbers in its group. Processor P_0 sends the largest number in its group to processor P_1, while processor P_1 sends the smallest number of its group to processor P_0. Each processor inserts the received number into its group, so that both still have n/p numbers. The actions are repeated with the new largest and smallest numbers. The algorithm is most conveniently implemented using a queue with a pointer to the top or bottom, as illustrated in Figure 10.33.

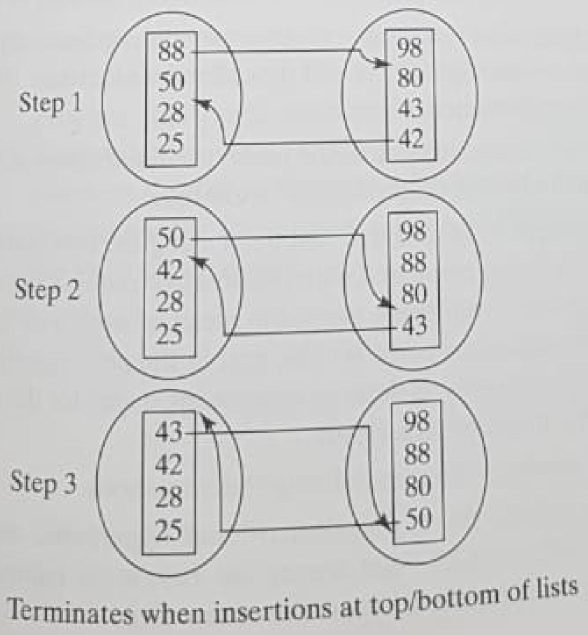


Figure 10.33 Compare-and-exchange algorithm for Problem 10-4.

The algorithm can terminate when the insertions would be at the top and bottom of the groups, as shown. The maximum number of communication steps would be when all the numbers in the left list were initially larger than all of the numbers in the other group (a maximum of x steps with x numbers in each group). Write a parallel program to implement Fox's method and evaluate it compared to the methods described in the text.

10-5. The following is an attempt to code the odd-even transposition sort of Section 10.2.2 as one SPMD program.

Process P_i

```
evenprocess = (i % 2 == 0);
evenphase = 1;
for (step = 0; step < n; step++, evenphase = !evenphase){
  if ((evenphase && evenprocess) || (!evenphase) && !evenprocess)) {
    send(&a, P_{i+1});
    recv(&x, P_{i+1});                      /* keep smaller number */
    if (x < a) a = x;
  } else {
    send(&a, P_{i-1});
    recv(&x, P_{i-1});                      /* keep larger number */
    if (x > a) a = x;
  }
}
```

Determine whether this code is correct, and if not, correct any mistakes.

10-6. The odd-even transposition sort of Section 10.2.2 was described on the assumption of an even number of numbers and processors. What changes to the code would be necessary to allow an odd number of numbers and processors?

10-7. Show that there are $\sqrt{n}(\sqrt{n}-1)$ communications for a transpose operation on a $\sqrt{n} \times \sqrt{n}$ array, as stated in Section 10.3.1.

10-8. Write a parallel program to implement shearsort.

10-9. Write a parallel program to find the kth smallest number of a set of numbers. Use a parallel version of quicksort, but apply it only to the set of numbers containing the kth smallest number. See Cormen, Leiserson, and Rivest (1990) for further discussion on the problem.

10-10. There have been many suggestions of ways to improve the performance of sequential quicksort (see, for example, Wainwright, 1985 and the references therein). The following are two suggestions that have appeared in the literature:

1. Rather than selecting the first number as the pivot, use the median of a group of three numbers picked randomly ("median-of-three" technique).
2. The initial set of numbers is divided into two parts using the first number as the pivot. While the numbers are being compared with the pivot, the sum of the numbers in each of two parts is computed. From these sums, the mean of each part is computed. The mean of each part is used as the pivot for this part in the next partitioning stage. The process of computing two means is done in subsequent stages for the next pivots. This algorithm is called the meansort algorithm.

Determine empirically the prospects for parallelizing these methods.

10-11. Draw the exchanges of numbers for a four-dimensional hypercube, using the parallel hypercube described in Section 10.2.6, that leaves the results in embedded ring order. Determine a general algorithm to handle a hypercube of any size.

10-12. Draw the compare-and-exchange circuit configuration for the odd-even mergesort algorithm described in Section 10.2.7 to sort 16 numbers. Sort the following sequence by hand using the odd-even mergesort algorithm:

12 2 11 4 9 1 10 15 5 7 14 3 8 13 6 16

10-13. Repeat Problem 10-12 for bitonic mergesort.

10-14. Compare Batcher's odd-even mergesort algorithm (Section 10.2.5) and his bitonic mergesort algorithm (Section 10.2.6), and assess their relative advantages for parallel implementation on a message-passing multicomputer.

10-15. Prove that the time complexity of odd-even mergesort is $O(\log^2 n)$ with n processors.

10-16. Identify those sorting algorithms in Chapter 10 that operate in place (i.e., do not use additional storage for the numbers as they are sorted. In place algorithms will reduce the storage requirements, and this factor may be important when sorting a large number of numbers.

10-17. Would radix sort work if the sorting operated from the most significant digit to the least significant digit? Discuss.

10-18. Implement two sorting algorithms in MPI and compare their performance.

10-19. Read the paper by Helman, Bader, and JáJá (1998) and implement their sorting algorithm in MPI.

Real Life

10-20. Fred has a deck of 52 playing cards that have been thoroughly shuffled. He has asked you to determine several things related to reordering them:

1. What modification to bitonic mergesort will be needed to sort the cards, given that there are four suits (spades, hearts, clubs, and hearts)?
2. How many friends will Fred have to invite over (and feed) to carry out a modified bitonic mergesort in parallel, and how many steps will each friend have to carry out?

10-21. One way to find matching entries in two files is first to sort the contents of the files and then to step through the files comparing entries. Analyze this method compared to simply comparing each entry of one file with each entry of the second file without sorting. Write a parallel program to find matching entries of two files by first sorting the files.

10-22. Sorting algorithms can create any permutation of an input sequence by simply numbering the input elements in the required order and sorting on these numbers. Write a parallel program that randomizes the contents of a document file for security purposes by first numbering the words in the file using a random-number generator and then sorting on the numbers. (Refer to Chapter 3 for details of random numbers.) Is this a good method of encoding a file for security purposes? Write a parallel program that restores the document file to its original state.