9.3 Bubble Sort and its Variants

The previous section presented a sorting network that could sort n elements in a time of Q($\log^2 n$). We now turn our attention to more traditional sorting algorithms. Since serial algorithms with Q($n \log n$) time complexity exist, we should be able to use Q(n) processes to sort n elements in time Q($\log n$). As we will see, this is difficult to achieve. We can, however, easily parallelize many sequential sorting algorithms that have Q(n^2) complexity. The algorithms we present are based on ***bubble sort***.

The sequential bubble sort algorithm compares and exchanges adjacent elements in the sequence to be sorted. Given a sequence $<a_1, a_2, ..., a_n>$, the algorithm first performs n - 1 compare-exchange operations in the following order: (a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n). This step moves the largest element to the end of the sequence. The last element in the transformed sequence is then ignored, and the sequence of compare-exchanges is applied to the resulting sequence $\langle a'_1, a'_2, \ldots, a'_{n-1} \rangle$. The sequence is sorted after n - 1 iterations. We can improve the performance of bubble sort by terminating when no exchanges take place during an iteration. The bubble sort algorithm is shown in Algorithm 9.2.

An iteration of the inner loop of bubble sort takes time Q(n), and we perform a total of Q(n) iterations; thus, the complexity of bubble sort is Q(n^2). Bubble sort is difficult to parallelize. To see this, consider how compare-exchange operations are performed during each phase of the algorithm (lines 4 and 5 of Algorithm 9.2). Bubble sort compares all adjacent pairs in order; hence, it is inherently sequential. In the following two sections, we present two variants of bubble sort that are well suited to parallelization.

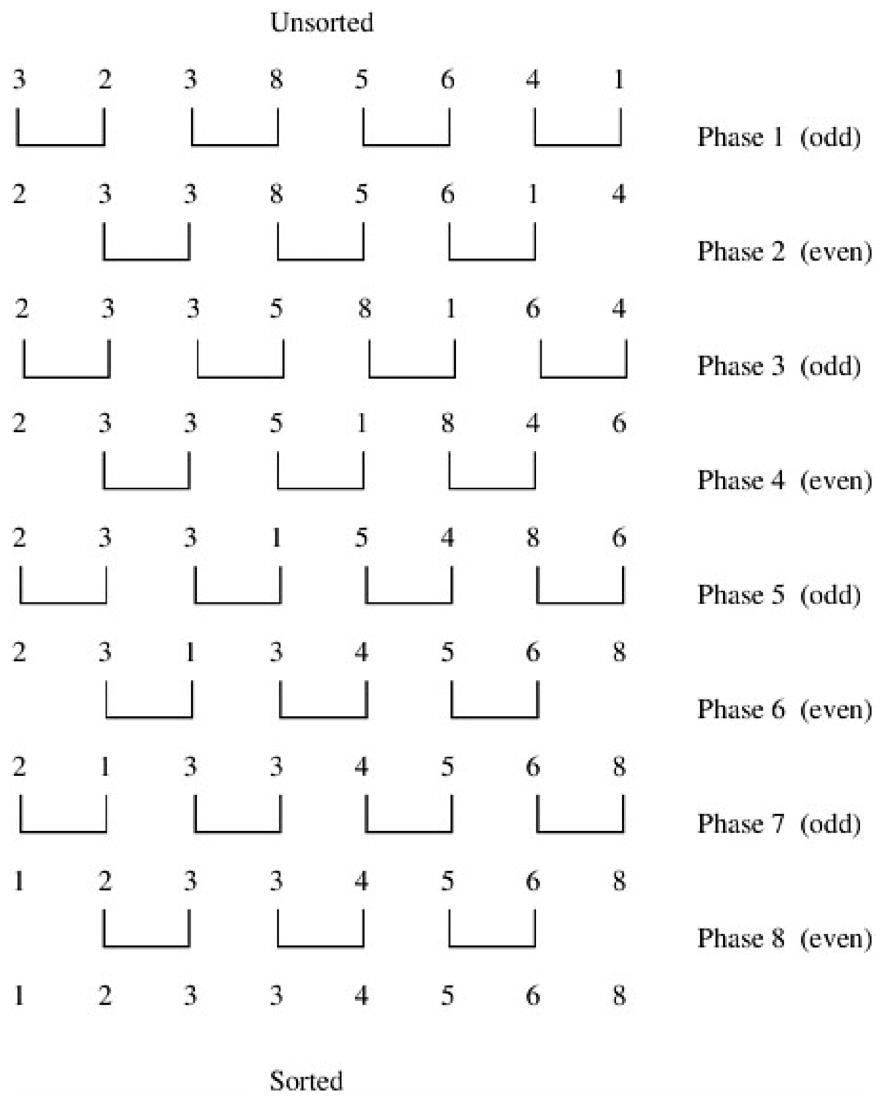
Algorithm 9.2 Sequential bubble sort algorithm.

```
1.   procedure BUBBLE_SORT(n)
2.   begin
3.      for i := n - 1 downto 1 do
4.          for j := 1 to i do
5.              compare-exchange(a_j, a_j + 1);
6.   end BUBBLE_SORT
```

9.3.1 Odd-Even Transposition

The ***odd-even transposition*** algorithm sorts n elements in n phases (n is even), each of which requires n/2 compare-exchange operations. This algorithm alternates between two phases, called the odd and even phases. Let $<a_1, a_2, ..., a_n>$ be the sequence to be sorted. During the odd phase, elements with odd indices are compared with their right neighbors, and if they are out of sequence they are exchanged; thus, the pairs (a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n) are compare-exchanged (assuming n is even). Similarly, during the even phase, elements with even indices are compared with their right neighbors, and if they are out of sequence they are exchanged; thus, the pairs (a_2, a_3), (a_4, a_5), ..., (a_{n-2}, a_{n-1}) are compare-exchanged. After n phases of odd-even exchanges, the sequence is sorted. Each phase of the algorithm (either odd or even) requires Q(n) comparisons, and there are a total of n phases; thus, the sequential complexity is Q(n^2). The odd-even transposition sort is shown in Algorithm 9.3 and is illustrated in Figure 9.13.

Figure 9.13. Sorting *n* = 8 elements, using the odd-even transposition sort algorithm. During each phase, *n* = 8 elements are compared.



Algorithm 9.3 Sequential odd-even transposition sort algorithm.

```
procedure ODD-EVEN($n$)
begin
   for $i$ := 1 to $n$ do
   begin
      if $i$ is odd then
           for $j$ := 0 to $n$/2 - 1 do
               compare-exchange($a_{2j + 1}$, $a_{2j + 2}$);
      if $i$ is even then
           for $j$ := 1 to $n$/2 - 1 do
               compare-exchange($a_{2j}$, $a_{2j + 1}$);
   end for
end ODD-EVEN
```

Parallel Formulation

It is easy to parallelize odd-even transposition sort. During each phase of the algorithm, compare-exchange operations on pairs of elements are performed simultaneously. Consider the one-element-per-process case. Let *n* be the number of processes (also the number of elements to be sorted). Assume that the processes are arranged in a one-dimensional array. Element a_i initially resides on process P_i for $i = 1, 2, ..., n$. During the odd phase, each process that has an odd label compare-exchanges its element with the element residing on its right neighbor. Similarly, during the even phase, each process with an even label compare-exchanges its element with the element of its right neighbor. This parallel formulation is presented in Algorithm 9.4.

Algorithm 9.4 The parallel formulation of odd-even transposition sort on an *n*-process ring.

```
procedure ODD-EVEN_PAR ($n$)
begin
   $id$ := process's label
   for $i$ := 1 to $n$ do
   begin
      if $i$ is odd then
          if $id$ is odd then
              compare-exchange_min($id$ + 1);
          else
              compare-exchange_max($id$ - 1);
      if $i$ is even then
          if $id$ is even then
              compare-exchange_min($id$ + 1);
          else
              compare-exchange_max($id$ - 1);
   end for
end ODD-EVEN_PAR
```

During each phase of the algorithm, the odd or even processes perform a compare-exchange step with their right neighbors. As we know from Section 9.1, this requires time Q(1). A total of *n* such phases are performed; thus, the parallel run time of this formulation is Q(*n*). Since the sequential complexity of the best sorting algorithm for *n* elements is Q(*n* log *n*), this formulation of odd-even transposition sort is not cost-optimal, because its process-time product is Q(n^2).

To obtain a cost-optimal parallel formulation, we use fewer processes. Let *p* be the number of processes, where $p < n$. Initially, each process is assigned a block of *n*/*p* elements, which it sorts internally (using merge sort or quicksort) in Q((*n*/*p*) log(*n*/*p*)) time. After this, the processes execute *p* phases (*p*/2 odd and *p*/2 even), performing compare-split operations. At the end of these phases, the list is sorted (Problem 9.10). During each phase, Q(*n*/*p*) comparisons are performed to merge two blocks, and time Q(*n*/*p*) is spent communicating. Thus, the parallel run time of the formulation is

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n).}^{\text{communication}}$$

Since the sequential complexity of sorting is Q(*n* log *n*), the speedup and efficiency of this formulation are as follows: **Equation 9.6**

$$\begin{aligned} S &= \frac{\Theta(n\log n)}{\Theta((n/p)\log(n/p)) + \Theta(n)} \\ E &= \frac{1}{1 - \Theta((\log p)/(\log n)) + \Theta(p/\log n)} \end{aligned}$$

From Equation 9.6, odd-even transposition sort is cost-optimal when *p* = *O*(log *n*). The isoefficiency function of this parallel formulation is Q($p\,2^p$), which is exponential. Thus, it is poorly scalable and is suited to only a small number of processes.

9.3.2 Shellsort

The main limitation of odd-even transposition sort is that it moves elements only one position at a time. If a sequence has just a few elements out of order, and if they are Q(*n*) distance from their proper positions, then the sequential algorithm still requires time Q(n^2) to sort the sequence. To make a substantial improvement over odd-even transposition sort, we need an algorithm that moves elements long distances. Shellsort is one such serial sorting algorithm.

Let *n* be the number of elements to be sorted and *p* be the number of processes. To simplify the presentation we will assume that the number of processes is a power of two, that is, $p = 2^d$, but the algorithm can be easily extended to work for an arbitrary number of processes as well. Each process is assigned a block of *n*/*p* elements. The processes are considered to be arranged in a logical one-dimensional array, and the ordering of the processes in that array defines the global ordering of the sorted sequence. The algorithm consists of two phases. During the first phase, processes that are far away from each other in the array compare-split their