```
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
```

Algorithm 9.2 Sequential bubble sort algorithm.

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.

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 $\langle a_1, a_2, \ldots, a_n \rangle$ 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 $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-1}, a_n)$ 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 $(a_2, a_3), (a_4, a_5), \ldots, (a_{n-2}, a_{n-1})$ are compare-exchanged. After n phases of odd-even exchanges, the sequence is sorted. Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons, and there are a total of n phases; thus, the sequential complexity is $\Theta(n^2)$. The odd-even transposition sort is shown in Algorithm 9.3 and is illustrated in Figure 9.13.

```
1.  procedure ODD-EVEN(n)
2.  begin
3.      for i := 1 to n do
4.      begin
5.          if i is odd then
6.              for j := 0 to n/2 − 1 do
7.                  compare-exchange(a_{2j+1}, a_{2j+2});
8.          if i is even then
9.              for j := 1 to n/2 − 1 do
10.                 compare-exchange(a_{2j}, a_{2j+1});
11.     end for
12. end ODD-EVEN
```

Algorithm 9.3 Sequential odd-even transposition sort algorithm.

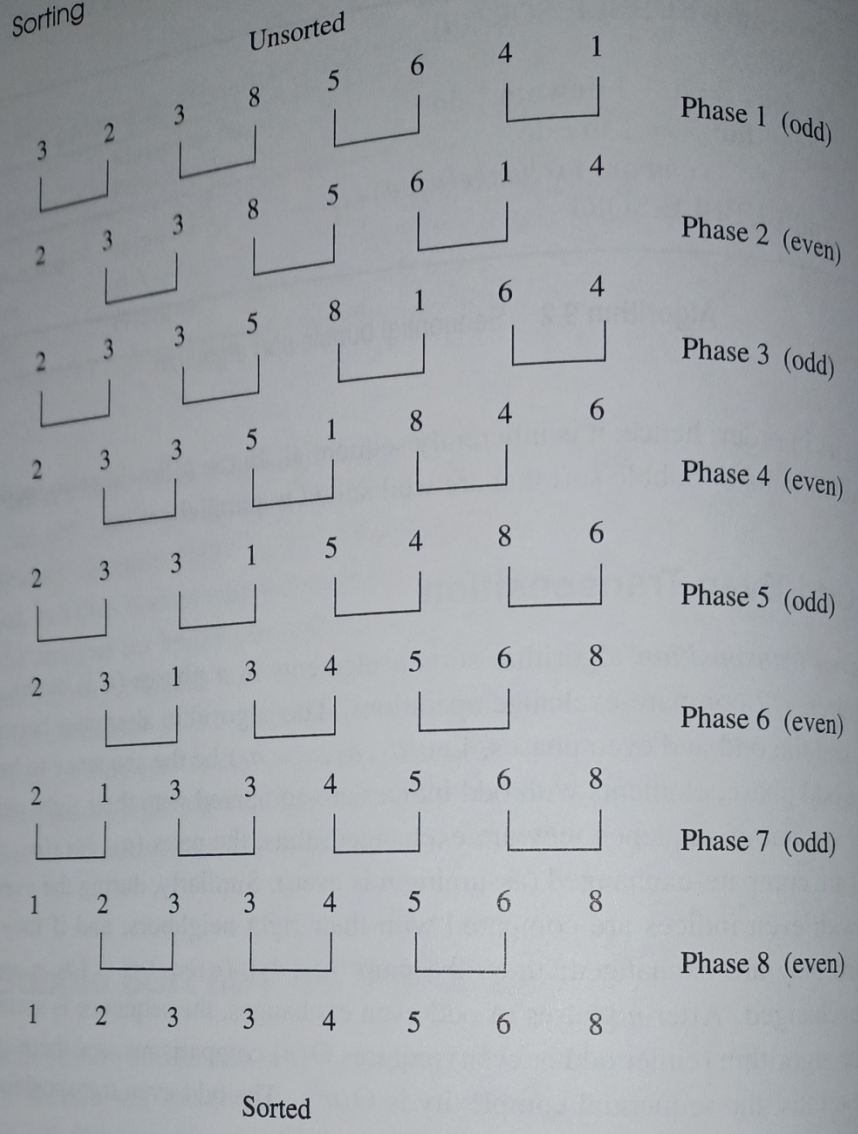

Figure 9.13 Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, $n = 8$ elements are compared.

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 $i = 1, 2, \ldots, n$. 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.

During each phase of the algorithm, the odd or even processes perform a compare-

```
1.  procedure ODD-EVEN_PAR(n)
2.  begin
3.      id := process's label
4.      for i := 1 to n do
5.      begin
6.          if i is odd then
7.              if id is odd then
8.                  compare-exchange_min(id + 1);
9.              else
10.                 compare-exchange_max(id − 1);
11.         if i is even then
12.             if id is even then
13.                 compare-exchange_min(id + 1);
14.             else
15.                 compare-exchange_max(id − 1);
16.     end for
17. end ODD-EVEN_PAR
```

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

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

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 $\Theta((n/p)\log(n/p))$ 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, $\Theta(n/p)$ comparisons are performed to merge two blocks, and time $\Theta(n/p)$ 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 $\Theta(n \log n)$, the speedup and efficiency of this formulation are as follows:

$$S = \frac{\Theta(n\log n)}{\Theta((n/p)\log(n/p)) + \Theta(n)} \qquad (9.6)$$

$$E = \frac{1}{1 - \Theta((\log p)/(\log n)) + \Theta(p/\log n)}$$