From Czech, Introduction to Parallel Computing; and Leighton, Introduction to Parallel Algorithms and Architectures.

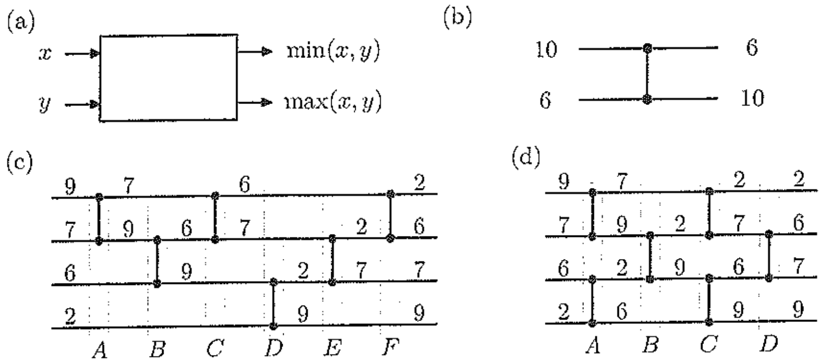


Figure 3.35. A comparator (a–b) and the comparator networks corresponding to sequential insertion sort (c) and to parallel odd-even transposition sort (d).

time of the algorithm is $3 \log n$. In what follows we prove that the execution of stage $t = 3k$ for $k > 0$ of the algorithm takes $O(1)$ time when $O(n)$ processors are used (similar estimates can be provided for stages $t = 3k + 1$ and $t = 3k + 2$). Note that at stage $t = 3k$ we have incomplete (active) vertices on level k, and also a number of incomplete vertices on certain levels higher than k, whereas the vertices below level k are incomplete. Examine the execution of a stage for vertices on level k. In $n/2^k$ vertices of this level the sequences of length 2^{k-1} are merged. According to Lemma 3.1 to merge two such sequences in $O(1)$ time 2^k processors are required, so for a total we need n processors for all vertices on level k. In vertices on consecutive levels from $k + 1$ upwards, the sequences of the following lengths: $2^{k-3}, 2^{k-5}, 2^{k-7}, \ldots,$ 1, respectively, are merged, which can be done in $O(1)$ time with:

$$\frac{n}{2^{k+1}} \cdot 2^{k-2} + \frac{n}{2^{k+2}} \cdot 2^{k-4} + \frac{n}{2^{k+3}} \cdot 2^{k-6} + \cdots + 2 = \frac{n}{8} + \frac{n}{64} + \frac{n}{512} + \cdots + 2 = O(n)$$

processors. Thus, to implement stage $3k$ in incomplete vertices on level k and on levels higher than k we need $O(n)$ processors. In the algorithm under discussion there are no conflicts among writes to the shared memory. However, there are conflicts among reads during the $O(1)$ time computation of the rank of an element a in a sorted sequence $S = (b_1, b_2, \ldots, b_k)$ (see Lemma 3.3 and the answer to Exercise 15b). Summing up, Cole's algorithm runs in $O(\log n)$ time in a CREW PRAM model. □

Cole's sorting algorithm (also in a more complicated EREW PRAM version compared with the CREW PRAM version discussed above) was presented in the article [78]. Likewise, it is presented in works [77, 79] and in books by Gibbons and Rytter [153] (sect. 5.2), and by Casanova et al. [67] (sect. 1.4).

3.3.2 Bitonic Sort—Batcher's Network

Cole's algorithm and the algorithm in Figure 3.13 are designed for a model with shared memory. In the literature the sorting problem is also solved adopting a model of **comparator network**. Such a network is made up of interconnected comparators operating in parallel. A **comparator** is a device with two inputs and two outputs (Figure 3.35a). The comparator with elements x and y on the inputs returns the minimum and maximum of x and y (in this order) on the outputs. It is assumed that the comparator operates in $O(1)$ time. In the sequel we will use the simplified designation of the comparator depicted in Figure 3.35b. Among comparator networks a special group of **sorting networks** are distinguished. Figures 3.35c–d show the examples of sorting networks with $n = 4$ inputs and outputs. The number of comparators in a network determines its size $R(n)$. The size of the example networks is $R(4) = 6$. The sequence of input elements depicted on the left side of Figures 3.35c–d are to be sorted. They are forwarded to the outputs depicted on the right side of the figures. In the course of sorting, the elements "flow" from left to right changing their positions as the result of comparator operations. More specifically, in time 0 the sequence (9,7,6,2) is entered on the inputs of the network in Figure 3.35c. In time 1 the comparator from group A is active, since its inputs become available, which are elements 9 and 7. In time 2 the comparator from group B performs its work on elements 9 and 6, and so on. The groups whose drawings are shaded and marked with letters from A to F include comparators to which data arrive in the same time. So the comparators within a group can operate in parallel. Note that in Figure 3.35c each group of the network contains only one comparator, while in Figure 3.35d the groups A and C contain two comparators. Thus the degree of parallelism in the latter network is higher. Each comparator is located at a certain depth of the network. In the example networks, the depth of the comparator in group A is 1, of the comparator in group B is 2, and so on. The speed of a network is evaluated by its depth $G(n)$, which is equal to the depth of the last group of comparators. Informally, the depth of a network determines the maximum number of comparators that an item of data has to pass from an input to an output of the network. Evidently, the smaller the depth of the network, the better. The network in Figure 3.35c demonstrates operation of a sequential algorithm of **insertion sort**, and that in Figure 3.35d presents operation of a parallel algorithm of **odd–even transposition sort**. Assuming that both networks have n inputs, the depths of networks in Figures 3.35c and 3.35d are $G(n) = n(n-1)/2 = O(n^2)$ and $G(n) = n$, respectively.

Batcher [38] proposed a **sorting network** with n inputs and $O((\log n)^2)$ depth. It is composed of **bitonic sorting networks** (BSN) and **merging networks** (MN). We will discuss them in turn, but first some definitions are needed. A bitonic sequence consists of two monotone subsequences, the first of which is nondecreasing and the second nonincreasing, or the other way around. For example, the sequences (2,4,7,9,6,1) and (9,6,1,2,4,7) are bitonic. One of the subsequences may be empty, which means that a nondecreasing or nonincreasing sequence are also bitonic. For the purpose of sorting, we concentrate on sequences of lengths that are powers of two,[28] $n = 2^r$ for some $r > 0$.

Suppose we are given a bitonic sequence such that $a_0 \le a_1 \le \cdots \le a_{j-1} \le a_j \ge a_{j+1} \ge \cdots \ge a_{n-1}$ or $a_0 \ge a_1 \ge \cdots \ge a_{j-1} \ge a_j \le a_{j+1} \le \cdots \le a_{n-1}$ for some index j, where $j = 0, 1, \ldots, n-1$. Then the sequences $\min(a_0, a_{n/2})$, $\min(a_1, a_{n/2+1}), \ldots,$ $\min(a_{n/2-1}, a_{n-1})$ and $\max(a_0, a_{n/2})$, $\max(a_1, a_{n/2+1}), \ldots, \max(a_{n/2-1}, a_{n-1})$ are bitonic. Moreover, each element of the first sequence is not larger than all the elements of the second sequence. Owing to this property, we can construct a bitonic sorting network BSN(n) that enables sorting n-element bitonic sequences for $n = 2^r$ (see Figure 3.36; it is assumed that the input sequence (a_i) for $i = 0, 1, \ldots, n-1$ is bitonic). A collection of $n/2$ comparators (shaded in the figure) create two bitonic

[28] Sequences that do not meet this condition may be supplemented by neutral elements.

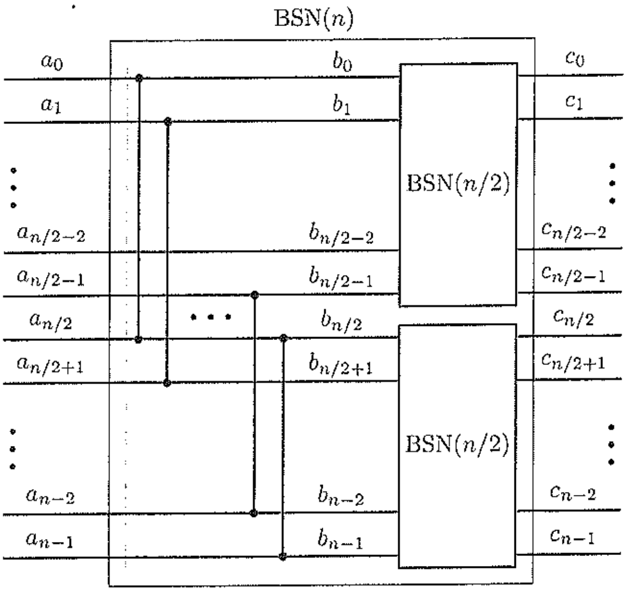


Figure 3.36. A recursive scheme of bitonic sorting network.

sequences: (b_i) for $i = 0, 1, \ldots, n/2-1$ and (b_j) for $j = n/2, n/2+1, \ldots, n-1$, where $b_i \leq b_j$ for all pairs of indices i and j. The sequences (b_i) and (b_j) are recursively sorted by networks BSN$(n/2)$, BSN$(n/4)$, ... etc. until the sorted nondecreasing sequence (c_i) for $i = 0, 1, \ldots, n-1$ is obtained. Figure 3.37a demonstrates the process of sorting a 8-element bitonic sequence.

While proving correctness of operation of a sorting network the zero-one principle is often used. It states that if the network is working correctly for sequences $(\{0,1\}^n)$, then it operates equally well for sequences (x_i) for $i = 1, 2, \ldots, n$ consisting of integers, reals or elements of any linearly ordered set.

Theorem 3.6. (zero-one principle) [222], [84], [335], [12] *If a comparator sorting network with n inputs correctly sorts all 2^n sequences of zeros and ones $(\{0,1\}^n)$, then it sorts correctly arbitrary sequences of elements.*

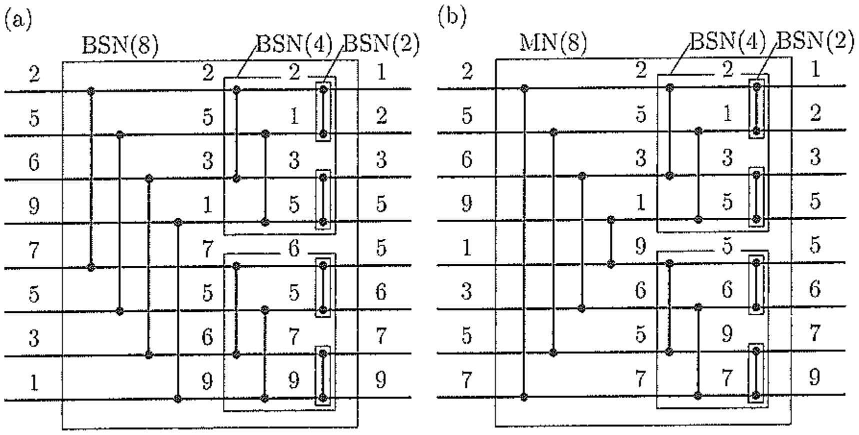


Figure 3.37. (a) Sorting 8-element bitonic sequence; (b) merging two nondecreasing 4-element sequences: (2,5,6,9) and (1,3,5,7).

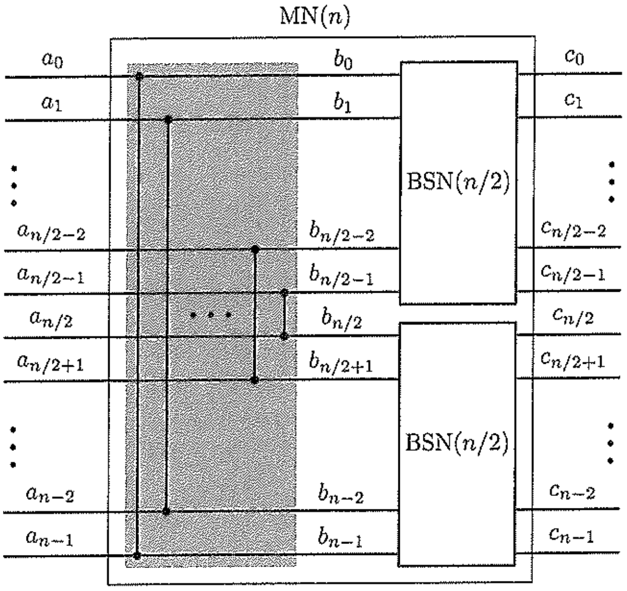


Figure 3.38. A recursive scheme of merging network.

Proof. We carry out the proof by contradiction. Suppose that the comparator network converts an input sequence $(x_1, x_2, \ldots, x_n)$ into an output sequence $(y_1, y_2, \ldots, y_n)$. Let f be an arbitrary monotone function that satisfies $f(x) \leq f(y)$ when $x \leq y$. Let us replace each element of the sequence $(x_1, x_2, \ldots, x_n)$ with a value $f(x_i)$ for $i = 1, 2, \ldots, n$. The sorting network consists of comparators, so any comparison of elements x_{i_1} and x_{i_2} will be replaced by a comparison of values $f(x_{i_1})$ and $f(x_{i_2})$. Because we have $f(x_{i_1}) \leq f(x_{i_2})$ when $x_{i_1} \leq x_{i_2}$, the set of comparisons made by the network for the sequence $(x_1, x_2, \ldots, x_n)$ will be the same as for the sequence $(f(x_1), f(x_2), \ldots, f(x_n))$. So the network converts the sequence $(f(x_1), f(x_2), \ldots, f(x_n))$ into $(f(y_1), f(y_2), \ldots, f(y_n))$. Let us now assume that the network correctly sorts the sequences of zeros and ones but it does not sort correctly some sequence of elements $(x_1, x_2, \ldots, x_n)$, that is in the output sequence there appear elements y_i and y_j such that $y_i > y_j$ for $i < j$. Consider the function f that for all elements less than or equal to y_i takes value 0, and for all elements greater than y_i takes value 1. In this way, we define the sequence $(f(x_1), f(x_2), \ldots, f(x_n))$ consisting of zeros and ones, which is not properly sorted by the network, which contradicts the assumption of the theorem. □

The zero-one principle entails that to prove correctness of any network action, it is sufficient to indicate that it works correctly for sequences consisting of zeros and ones.

Lemma 3.6. *The bitonic sorting network BSN(n) shown in Figure 3.36 correctly sorts any bitonic sequence (a_i) for $i = 0, 1, \ldots, n-1$ where $n = 2^r$ and $r > 0$.*

Proof. See Exercise 17a. □

The second component of the Batcher's sorting network is the merging network MN(n) (Figure 3.38). On the input of this network two sequences (a_i) for

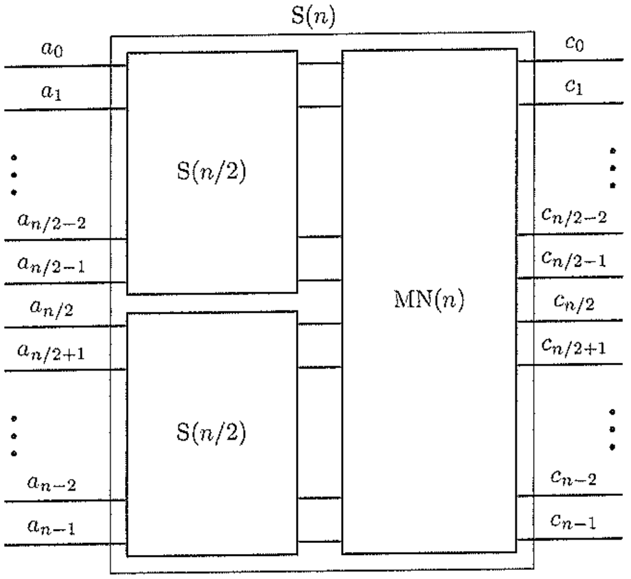


Figure 3.39. A recursive scheme of Batcher's sorting network.

$i = 0, 1, \ldots, n/2 - 1$ and (a_j) for $j = n/2, n/2 + 1, \ldots, n - 1$ sorted in nondecreasing order are entered. Based on these sequences, $n/2$ comparators (shaded in the figure) create two bitonic sequences: (b_i) for $i = 0, 1, \ldots, n/2 - 1$ and (b_j) for $j = n/2$, $n/2 + 1, \ldots, n - 1$ such that $b_i \leq b_j$ for all pairs of indices i and j. The sequences are then recursively sorted in a nondecreasing sequence (c_i) for $i = 0, 1, \ldots, n - 1$ by the bitonic sorting networks BSN(k) for $k = n/2, n/4, \ldots, 2$. Figure 3.37b depicts merging of two nondecreasing 4-element sequences.

Lemma 3.7. *The merging network MN(n) depicted in Figure 3.38 merges correctly the sequences sorted in nondecreasing order (a_i) for $i = 0, 1, \ldots, n/2 - 1$ and (a_j) for $j = n/2, n/2 + 1, \ldots, n - 1$ in one output sequence sorted in nondecreasing order (c_i) for $i = 0, 1, \ldots, n - 1$.*

Proof. See Exercise 17b. □

It is easy to note that operation times of networks BSN(n) and MN(n) defined by their depth are $G(n) = r = \log n$, and the sizes of networks are $R(n) = r(n/2) = (n \log n)/2$. Figure 3.39 shows the recursive scheme of a sorting network S(n) proposed by Batcher. After expanding the recursion the full network consists of, from left to right, the group of networks S(2), where each of the networks is a single comparator, and then of subsequent groups of merging networks MN(4), MN(8), ..., MN(n). Figure 3.40 illustrates the process of sorting 8-element sequence by Batcher's network.

Theorem 3.7. *Batcher's network given in Figure 3.39 correctly sorts in nondecreasing order any sequence of elements (a_i) for $i = 0, 1, \ldots, n - 1$.*

Proof. Correctness of Batcher's network can be easily proved by induction on the nesting level of the network applying the zero-one principle and Lemmas 3.6 and 3.7,

nondecreas-
ed in the fig-
for $j = n/2$,
e sequences
$1, \dots, n-1$
.37b depicts

ges correctly
and (a_j) for
ng order (c_i)

$\mathrm{N}(n)$ defined
$= r(n/2) =$
rk S(n) pro-
ists of, from
single com-
MN(8), ...,
by Batcher's

ndecreasing

ction on the
s 3.6 and 3.7,

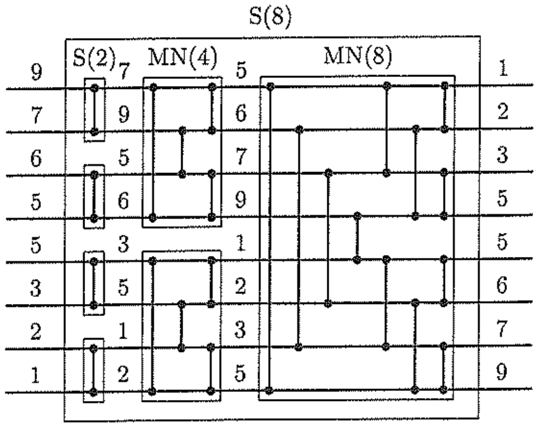


Figure 3.40. Sorting 8-element sequence by Batcher's network.

which demonstrate correct operation of the network components, that is bitonic sorting networks BSN(n) and merging networks MN(n). □

The depth of the merging networks in particular groups equals 1 for the networks MN(2), and then $2, 3, \dots, r = \log n$ for the merging networks from MN(4) up to MN(n). The depth of Batcher's sorting network is then: $G(n) = \sum_{i=1}^{r} i = (1+r)r/2 = O(r^2) = O((\log n)^2)$, and the size: $R(n) = \sum_{i=1}^{r}(n/2^i)(2^i \log 2^i)/2 = n(1+r)r/4 = O(n(\log n)^2)$.

Ajtai, Komlós, and Szemerédi [9] proposed a network with the optimum depth $G(n) = O(\log n)$ and optimum size $R(n) = O(n \log n)$. The network has only a theoretical importance, because due to complication in its design the constants of proportionality in the estimates of $G(n)$ and $R(n)$ are very large. Therefore in practice, Batcher's network despite the inferior asymptotic depth is faster. A lot of works were devoted to sorting networks. Information about them can be found in the works by Jájá [205] (sect. 4.4), Cormen et al. [84] (chap. 28), Knuth [222] (sect. 5.3.4), Casanova et al. [67] (chap. 2). Gibbons and Rytter [153] (sect. 5.3) presented a description of the network proposed by Ajtai, Komlós, and Szemerédi and improved and simplified by Paterson [305].

3.8.3 The Parallel Computation Thesis

The parallel computation thesis is related to the NC complexity class. Let $f(n)$ be an arbitrary function of size n of the problem.

Thesis. *The class of problems that can be solved by means of a sequential computer using $f(n)^{O(1)}$ memory locations is equal to the class of problems that can be solved ba means of a computer with unlimited parallelism in time $f(n)^{O(1)}$.*

Informally, the thesis says that the capacity of memory of sequential computation and the time of parallel computation are polynomially related. The thesis is not a theorem, but rather a hypothesis. The reason is the vague notion of a computer with "unlimited parallelism." It is assumed that such a computer—with unspecified structure—must have sufficiently high potential. For example, it must have a processor complexity significantly higher than polynomial to effectively simulate important issue—timing. How does each processor know when to stop one phase and start the next? There are at least three possible answers. First, each processor could maintain a local clock to keep track of the time, but this would be quite costly in practice and would require more than finite control. Second, we could simulate a broadcast by retiming, but this might cost a factor of two in speed. Third, we could have two kinds of clock "ticks" regulate the operation of each processor, one for regular operation and one to denote a change of phase. Although, the last operation doesn't rigorously fit into the fixed-connection network model as we have defined it, the approach is probably the simplest in practice.

1.6.3 A $(3\sqrt{N} + o(\sqrt{N}))$-Step Sorting Algorithm ⋆

For large values of N the $\sqrt{N}(\log N + 1)$-step algorithm described in Subsection 1.6.2 is unnecessarily slow, even when modified as suggested in the exercises. In what follows, we describe a more complicated algorithm that requires only $3\sqrt{N} + o(\sqrt{N})$ steps. As we show in Subsection 1.6.4, this is close to optimal for sorting on a mesh.

The algorithm sorts the N items into snakelike order as follows:

Phase 1: Divide the mesh into $N^{1/4}$ *blocks* of size $N^{3/8} \times N^{3/8}$ and simultaneously sort each block in snakelike order.

Phase 2: Perform an $N^{1/8}$-way unshuffle of the columns. In particular, permute the columns so that the $N^{3/8}$ columns in each block are distributed evenly among the $N^{1/8}$ vertical slices. (A *vertical slice* is simply a column of blocks. Similarly, a *horizontal slice* is a row of blocks. For example, see Figure 1-84.)

Phase 3: Sort each block into snakelike order.

Phase 4: Sort each column in linear order.

Phase 5: Collectively sort blocks 1 and 2, blocks 3 and 4, etc., of each vertical slice into snakelike order.

Phase 6: Collectively sort blocks 2 and 3, blocks 4 and 5, etc., of each vertical slice into snakelike order.

Phase 7: Sort each row in linear order according to the direction of the overall N-cell snake.

Phase 8: Perform $2N^{3/8}$ steps of odd-even transposition sort on the overall N-cell snake.

Phases 1, 3, 5, and 6 can all be accomplished using the algorithm described in Subsection 1.6.2. Collectively, these phases consume at most

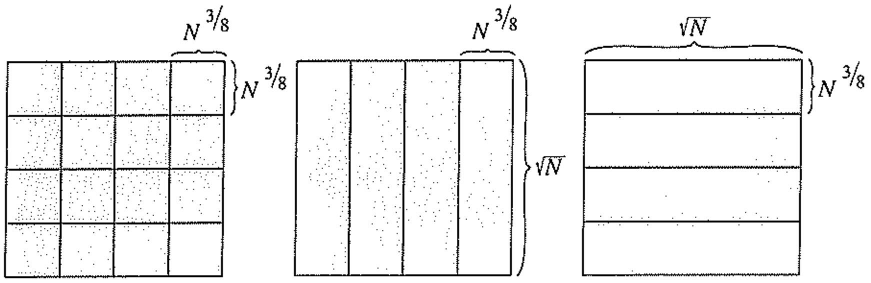
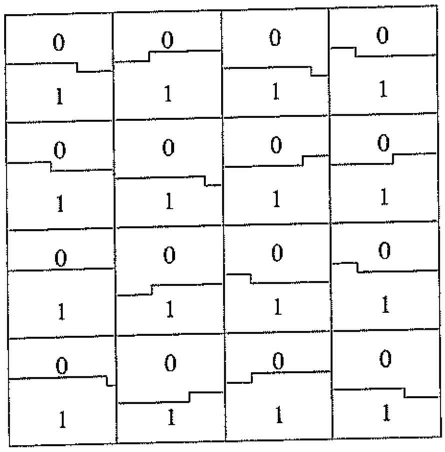


Figure 1-84 *Definition of blocks and slices. Each block contains* $N^{3/4}$ *cells and each slice contains* $N^{1/8}$ *blocks.*

$O(N^{3/8}\log N)$ steps. Phases 4 and 7 can be accomplished using odd-even transposition sort in $2\sqrt{N}$ steps. Phase 2 can be accomplished in a variety of ways using no more than $\sqrt{N}+O(N^{3/8})$ steps. (The details of implementing this operation with finite local control are left as Problem 1.167.) Phase 8 also uses odd-even transposition sort, but only for $2N^{3/8}$ steps. Hence, the total running time is $3\sqrt{N}+O(N^{3/8}\log N)=3\sqrt{N}+o(\sqrt{N})$. The size of the lower order term can be decreased by applying the algorithm recursively to sort the $N^{3/8}\times N^{3/8}$ blocks (Problem 1.168), but the improvement is hardly worth the effort for reasonable values of N.

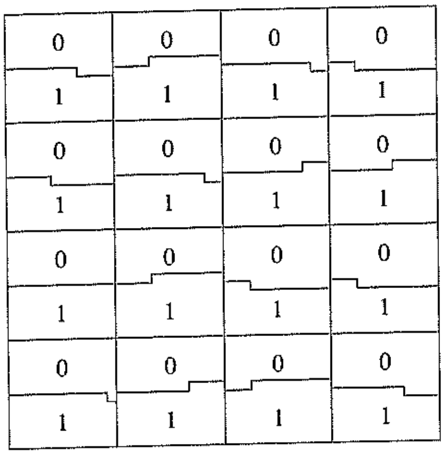
It remains to prove that the algorithm actually sorts any selection of N numbers into snakelike order. To prove this, we first observe that the algorithm only makes use of oblivious comparison-exchange operations, so we can apply Lemma 1.4 and restrict our attention to 0s and 1s. Thus, after Phase 1, the contents of the cells appear as in Figure 1-85. In particular, at most one row in each block is dirty. Hence, after Phase 2, the number of 1s in a block can differ by at most $N^{1/8}$ from the number of 1s in any other block in the same horizontal slice. After Phase 3, therefore, at most two rows in any horizontal slice are dirty. For example, see Figure 1-86.

After Phase 4, we know that there are at most $N^{1/8}$ dirty rows in each vertical slice. Hence, Phases 5 and 6 serve to sort the entire vertical slice in snakelike order. This leaves just one dirty row in each vertical slice. Since the number of 1s in two blocks of the same horizontal slice differ by at most $N^{1/8}$ after Phase 2, we know that the number of 1s in two vertical slices differ by at most $N^{1/4}$ after Phase 2. Since Phases 3–6 do not change the number of 1s in a vertical slice, the same condition holds after Phase 6. Hence, there are at most two dirty rows overall after Phase 6. For example,



0	0	0	0
1	1	1	1
0	0	0	0
1	1	1	1
0	0	0	0
1	1	1	1
0	0	0	0
1	1	1	1

Figure 1-85 *Configuration of* 0*s and* 1*s after Phase* 1. *At most one row in each block is dirty.*



0	0	0	0
1	1	1	1
0	0	0	0
1	1	1	1
0	0	0	0
1	1	1	1
0	0	0	0
1	1	1	1

Figure 1-86 *Configuration of* 0*s and* 1*s after Phase* 3. *At most two rows in each horizontal slice are dirty.*

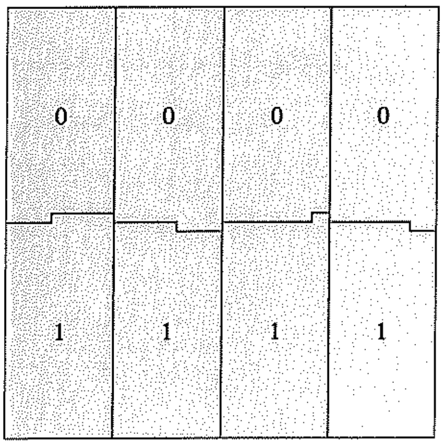


Figure 1-87 *Configuration of* 0*s and* 1*s after Phase* 6*. Each vertical slice contains at most one dirty row. Since the number of* 1*s in different vertical slices differs by at most* $N^{1/4}$*, there are at most two dirty rows overall.*

see Figure 1-87.

If there is only one dirty row after Phase 6, then Phase 7 completes the sorting. If there are two dirty rows after Phase 6, then the upper row must contain all 0s except for as many as $N^{3/8}$ 1s, and the lower row must contain all 1s except for as many as $N^{3/8}$ 0s. Hence, Phases 7 and 8 complete the sorting.

1.6.4 A Matching Lower Bound

For the most part, we have postponed the discussion of lower bounds on running time until Volume II, where a general theory is developed. In the particular case of sorting on a mesh, however, there is a simple proof that any algorithm requires $3\sqrt{N} - o(\sqrt{N})$ steps to sort N items in snakelike order. Since the bound is so close to that achieved by the algorithm just described in Subsection 1.6.3, and since the methods required for the proof are quite particular to snakelike-order sorting on a mesh, we included the result here.

It is immediately clear that any sorting algorithm for a $\sqrt{N} \times \sqrt{N}$ mesh must take at least $2\sqrt{N} - 2$ steps in the worst case, no matter what final order of items is desired. This is simply because the item in the $(1,1)$ position may have to move to the $(\sqrt{N}, \sqrt{N})$ position for the final order. Since every path from the $(1,1)$ cell to the $(\sqrt{N}, \sqrt{N})$ cell traverses