Parallel Computation: Models And Methods

(2)

Selim G. Akl

Queen's University
Kingston, Ontario, Canada

Prentice Hall, Upper Saddle River, New Jersey 07458

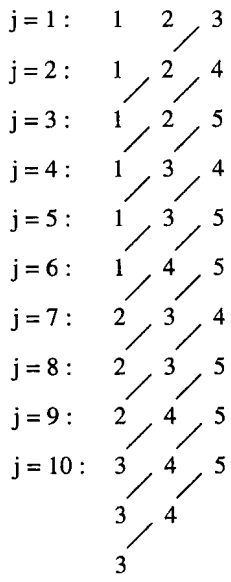

Figure 7.24: Processor P_i is one step ahead of P_{i+1} in the generation of combinations.

2. During the jth iteration, P_{i+1} updates $D(i+1)$; the new value is $a_k(i+1)$, where $k = (j+1)+n-(i+1) = j+n-i$. Since this calculation may require reference to $a_{j+n-i}(i)$, the latter needs to be stored by P_i.

3. During the jth iteration P_{i-1} updates $D(i-1)$; the new value is $a_k(i-1)$, where $k = (j+1)+n-(i-1)$. Since this calculation may require references to $a_{j+1}(i)$, $a_{j+2}(i)$, ..., $a_{j+n-i}(i)$, these values need to be stored by P_i.

The preceding analysis leads us to conclude that P_i needs to maintain $a_j(i)$, $a_{j+1}(i)$, ..., $a_{j+n-i}(i)$. We observe, however, that within any $n-i+1$ successive combinations, where the elements in positions 1, 2, ..., $i-1$ are kept fixed, the element in position i has at most three different values: $D(i)$ and two other values, denoted by $d_1(i)$ and $d_2(i)$. This can be easily seen by noting that, at position i, any element v, where $v \leq m-n+i$, occurs $C(n-i, m-v)$ times in successive combinations:

1. If $v = m-n+i$, then it appears once.

2. If $v = m-n+i-1$, then it appears $n-i+1$ times.

3. If $v < m-n+i-1$, then it appears more than $n-i+1$ times.

Therefore, P_i needs to store only five local variables, namely, the three values of v and the number of repetitions $r(i)$ and $r_1(i)$ when v appears more than once. All processors terminate when $D(i) = m - n + i$ has been repeated n times; this occurs simultaneously for all i, $1 \leq i \leq n$. The algorithm is given next as algorithm LINEAR ARRAY COMBINATIONS. Note that $d_1(i)$, $d_2(i)$, and $r_1(i)$ can be initialized arbitrarily, since each is needed only *after* it has been assigned a value.

Algorithm LINEAR ARRAY COMBINATIONS

Step 1: (1.1) $D(0) \leftarrow m - n$
(1.2) $D(n+1) \leftarrow m$
(1.3) $r(n+1) \leftarrow 0$
Step 2: **for** $i = 1$ **to** n **do in parallel**
(2.1) $D(i) \leftarrow i$
(2.2) $r(i) \leftarrow n - i + 1$
(2.3) **while not** ($D(i) = m - n + i$ **and** $r(i) > n$) **do**
(i) **if** $r(i) > n - i$
then output $D(i)$
else if $r(i) + r_1(i) > n - i$
then output $d_1(i)$
else output $d_2(i)$
end if
end if
(ii) $x \leftarrow D(i-1)$
(iii) $y \leftarrow D(i+1)$
(iv) $z \leftarrow r(i+1)$
(v) **if** $D(i) = m - n + i$
then (a) $d_2(i) \leftarrow d_1(i)$
(b) $d_1(i) \leftarrow D(i)$
(c) $r_1(i) \leftarrow r(i)$
(d) $D(i) \leftarrow x + 1$
(e) $r(i) \leftarrow 1$
else if $y = m - n + i$ **and** $z = n - i$
then (a) $d_2(i) \leftarrow d_1(i)$
(b) $d_1(i) \leftarrow D(i)$
(c) $r_1(i) \leftarrow r(i)$
(d) $D(i) \leftarrow D(i) + 1$
(e) $r(i) \leftarrow 1$
else if $r(i) \leq n$
then $r(i) \leftarrow r(i) + 1$
end if
end if
end if
end while
end for. ■

Analysis. The algorithm uses n processors, each with a constant amount of storage space, and generates all $C(n,m)$ combinations, in constant time per combination. It therefore has a cost of $O(nC(n,m))$, which is optimal, in view of the $\Omega(nC(n,m))$ lower bound on the number of operations required to generate all n-combinations of m elements.

7.4.4 Permutations

A *permutation* of $S = \{s_1, s_2, \ldots, s_n\}$ is an arrangement of the elements of S in a certain order. Thus, for $n = 5$, $s_4s_2s_5s_1s_3$ and $s_2s_3s_5s_4s_1$ are two different permutations. There are $n!$ distinct permutations of n elements, requiring $\Omega(nn!)$ operations to be generated. In this section, we describe an algorithm for generating all permutations of n *arbitrary* elements on which a total order is defined such that $s_1 \prec s_2 \prec \cdots \prec s_n$. The algorithm runs on a linear array of n processors P_1, $P_2, \ldots, P_n$, indexed from left to right. Each processor is responsible for producing one element of every permutation generated. Once a permutation has been generated, each processor updates the element it just produced, and the next permutation is generated.

The algorithm generates permutations such that each permutation differs from the previous one in the least possible way. This is accomplished by creating each new permutation through a transposition of two neighboring elements in the previous one. The resulting permutations are said to be generated in *minimal-change order*. This approach is particularly suitable for the linear array, since each processor P_i has direct access only to its two adjacent processors P_{i-1} and P_{i+1}.

Let $E(i)$ be the output of processor P_i. After each adjacent transposition, P_i generates an updated $E(i)$, $1 \leq i \leq n$, resulting in an entire new permutation being produced as output. Initially, $E(i) \leftarrow s_i$, $1 \leq i \leq n$. The following steps are then repeated until the algorithm terminates:

1. Move element s_n to the left, from P_n to P_1, by repeatedly exchanging it with its left neighbor.

2. Generate the next permutation of $\{s_1, s_2, \ldots, s_{n-1}\}$ in $P_2, P_3, \ldots, P_n$.

3. Move element s_n to the right, from P_1 to P_n, by repeatedly exchanging it with its right neighbor.

4. Generate the next permutation of $\{s_1, s_2, \ldots, s_{n-1}\}$ in $P_1, P_2, \ldots, P_{n-1}$. ■

The algorithm is based on the idea of generating the permutations of $\{s_1, s_2, \ldots, s_n\}$ from the permutations of $\{s_1, s_2, \ldots, s_{n-1}\}$ by taking each such permutation and inserting s_n in all n possible positions of it. For example, taking the permutation $s_1s_2 \ldots s_{n-1}$ of $\{s_1, s_2, \ldots, s_{n-1}\}$, we get n permutations of $\{s_1, s_2, \ldots, s_n\}$ as follows:

$$
\begin{array}{cccccc}
s_1 & s_2 & \cdots & s_{n-2} & s_{n-1} & s_n \\
s_1 & s_2 & \cdots & s_{n-2} & s_n & s_{n-1} \\
s_1 & s_2 & \cdots & s_n & s_{n-2} & s_{n-1} \\
 & & \vdots & & & \\
s_n & s_1 & \cdots & s_{n-3} & s_{n-2} & s_{n-1}
\end{array}
$$

In Step 1 of the algorithm, we are given a permutation of $\{s_1, s_2, \ldots, s_{n-1}\}$ in $P_1, P_2, \ldots, P_{n-1}$ and the element s_n in P_n. The element s_n is moved to the left $n-1$ times, thus generating $n-1$ distinct permutations of $\{s_1, s_2, \ldots, s_n\}$. Therefore, Step 1 can be viewed as consisting of $n-1$ pulses, each producing a distinct permutation.

With s_n in P_1, Step 2 generates the next permutation of $\{s_1, s_2, \ldots, s_{n-1}\}$ in $P_2, P_3, \ldots, P_n$. Step 3 moves s_n to the right $n-1$ times, and $n-1$ additional permutations of $\{s_1, s_2, \ldots, s_n\}$ are obtained. Like Step 1, Step 3 can be viewed as consisting of $n-1$ pulses, each producing a new permutation. Finally, Step 4 generates the next permutation of $\{s_1, s_2, \ldots, s_{n-1}\}$ in $P_1, P_2, \ldots, P_{n-1}$ while s_n is in P_n, and the loop is restarted at Step 1.

Steps 1 and 3 are trivial to implement on a linear array of processors: During each pulse, the processor holding s_n exchanges it with its left neighbor in Step 1 and its right neighbor in Step 3. It remains to show how Steps 2 and 4 are implemented.

To generate the $(n-1)! - 1$ permutations of $\{s_1, s_2, \ldots, s_{n-1}\}$ that follow $s_1 s_2 \ldots s_{n-1}$, also by adjacent transpositions, we assign a *direction* to every element. This is denoted by an arrow above the element, for illustration. Initially, all arrows point to the left. Thus, if the permutations of $\{s_1, s_2, s_3, s_4\}$ are to be generated, we would have

$$\overleftarrow{s_1}\ \overleftarrow{s_2}\ \overleftarrow{s_3}\ \overleftarrow{s_4}\ .$$

Now, an element is said to be *mobile* if its direction points to a "smaller" adjacent neighbor—that is, a neighbor which precedes it according to the order relation $\prec$. (Recall that $s_1 \prec s_2 \prec \cdots \prec s_n$.) In the foregoing example, s_2, s_3, and s_4 are mobile, while in

$$\overrightarrow{s_3}\ \overleftarrow{s_1}\ \overleftarrow{s_2}\ \overrightarrow{s_4},$$

only s_2 and s_3 are mobile. The algorithm is as follows:

while there are mobile elements **do**
(1) Find the largest mobile element; call it s_m.
(2) Switch s_m with the adjacent neighbor to which its direction points.
(3) Reverse the direction of all elements larger than s_m.
end while. ■

To implement this idea on the linear array of processors, we set the direction of $E(1), E(2), \ldots, E(n-1)$ to the left initially. The direction of s_n is immaterial to the proper execution of the algorithm and can be defined arbitrarily. How do all processors in Steps 2 and 4 know the largest mobile element? This is done by

propagating that information during the $n-1$ pulses that precede each of these steps. A variable can be used that travels along with s_n and holds at any given time the largest mobile element in $\{s_1, s_2, \ldots, s_{n-1}\}$ it has encountered so far. When s_n reaches its destination at the end of Step 1 (Step 3), the largest mobile element is "known" to P_1 (P_n). It would appear that $n-1$ additional pulses are needed to make this information "known" to the other processors, thus violating the constant-time condition. In order to avoid this, a second variable is used that travels in the direction opposite to s_n; this variable also holds at any given time the largest mobile element it has encountered so far. Thus, each processor P_i stores two variables:

leftmax(i) = index of the largest mobile element from the sequence $\{s_1, s_2, \ldots, s_{n-1}\}$ in $P_1, P_2, \ldots, P_i$.

rightmax(i) = index of the largest mobile element from the sequence $\{s_1, s_2, \ldots, s_{n-1}\}$ in $P_i, P_{i+1}, \ldots, P_n$.

These two variables are initialized at the beginning of Steps 1 and 3 as follows: Let $E(i) = s_k$; then:

$$\begin{array}{lcll} \textit{leftmax}(i) = \textit{rightmax}(i) & = & k, & \text{provided that } E(i) \prec s_n \text{ and } E(i) \text{ is mobile} \\ & = & 0 & \text{otherwise.} \end{array}$$

The variables are then updated during the $n-1$ pulses of Steps 1 and 3.

Example 7.16 Let $n = 5$. The values of $E(i)$, *leftmax*(i), and *rightmax*(i) during the first iteration of Step 1 are shown in Figs. 7.25 and 7.26. □

The complete algorithm is given next as algorithm LINEAR ARRAY PERMUTATIONS. It begins with an initialization phase A in which the first permutation is produced as output. This is followed by a second phase B in which Steps 1–4 are iterated to generate the remaining permutations. Note that:

1. Whenever the index k is used, it is assumed that $E(i) = s_k$.
2. The direction of $E(i)$ is stored in a variable *arrow*(i), taking one of the two values **left** and **right**.
3. Two additional variables $E(0)$ and $E(n+1)$ are used for convenience such that $E(0) = E(n+1) = \Delta$, where $s_n \prec \Delta$.
4. The algorithm terminates when no mobile element is found, a condition that is detected simultaneously by all processors, since in this case *leftmax*(i) = *rightmax*(i) for all $1 \leq i \leq n$. Each processor for which this equality holds checks whether the same is true for one of its two neighbors, and if so, the processor terminates execution.

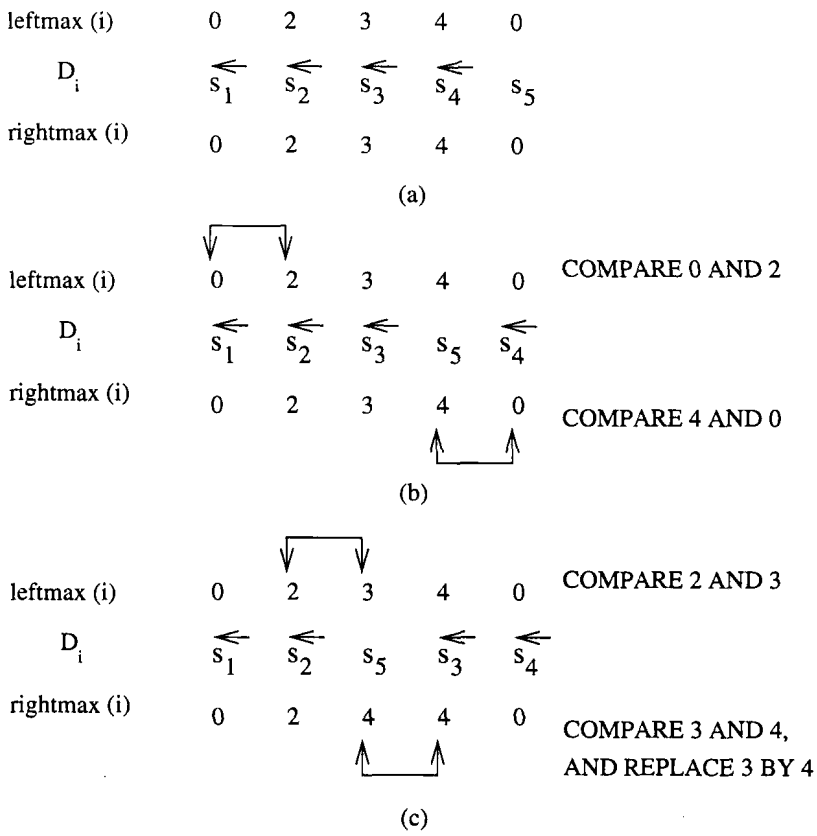

Figure 7.25: Finding the largest mobile element on a linear array (first two pulses): (a) Initially; (b) After one pulse; (c) After two pulses.

Algorithm LINEAR ARRAY PERMUTATIONS

A. Initialization

Step 1: **for** $i = 1$ **to** n **do in parallel**

(1.1) $E(i) \leftarrow s_i$

(1.2) $arrow(i) \leftarrow$ **left**

(1.3) **output** $E(i)$

end for

Step 2: $E(0) \leftarrow \Delta, \quad E(n+1) \leftarrow \Delta$

B. Repeat

Step 1: (1.1) **for** $i = 1$ **to** n **do in parallel**

if ($arrow(i) =$ **left and** $E(i-1) \prec E(i) \prec s_n$)

or ($arrow(i) =$ **right and** $E(i+1) \prec E(i) \prec s_n$)

then $leftmax(i) \leftarrow k$, $rightmax(i) \leftarrow k$

else $leftmax(i) \leftarrow 0$, $rightmax(i) \leftarrow 0$

end if

end for

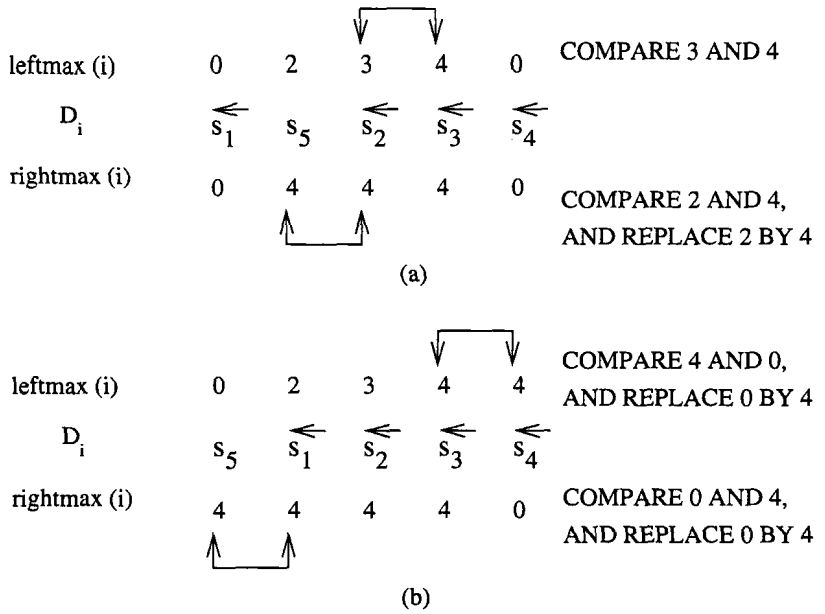

Figure 7.26: Finding the largest mobile element on a linear array (second two pulses): (a) After three pulses; (b) After four pulses.

(1.2) **for** $i = 1$ **to** $n-1$ **do**
(i) $E(n-i) \leftrightarrow E(n-i+1)$, $arrow(n-i) \leftrightarrow arrow(n-i+1)$
(ii) **for** $j = 1$ **to** n **do in parallel**
output $E(j)$
end for
(iii) $leftmax(i+1) \leftarrow \max(leftmax(i), leftmax(i+1))$
(iv) $rightmax(n-i) \leftarrow \max(rightmax(n-i), rightmax(n-i+1))$
end for

Step 2: (2.1) **for** $i = 2$ **to** n **do in parallel**
if $\max(leftmax(i), rightmax(i)) < k$
then reverse the direction of $arrow(i)$
else if $\max(leftmax(i), rightmax(i)) = k$
then if $arrow(i) =$ **left**
then $E(i-1) \leftrightarrow E(i)$, $arrow(i-1) \leftrightarrow arrow(i)$
else $E(i) \leftrightarrow E(i+1)$, $arrow(i) \leftrightarrow arrow(i+1)$
end if
end if
end if
end for
(2.2) **for** $i = 1$ **to** n **do in parallel**
output $E(i)$
end for

Step 3: (3.1) Same as Step (1.1) of phase B
(3.2) **for** $i = 1$ **to** $n - 1$ **do**
(i) $E(i) \leftrightarrow E(i+1)$, $arrow(i) \leftrightarrow arrow(i+1)$
(ii) **for** $j = 1$ **to** n **do in parallel**
output $E(j)$
end for
(iii) $leftmax(i+1) \leftarrow \max(leftmax(i), leftmax(i+1))$
(iv) $rightmax(n-i) \leftarrow \max(rightmax(n-i), rightmax(n-i+1))$
end for
Step 4: (4.1) **for** $i = 1$ **to** $n - 1$ **do in parallel**
Same as body of loop in Step (2.1)
end for
(4.2) **for** $i = 1$ **to** n **do in parallel**
output $E(i)$
end for
until there are no mobile elements. ■

Analysis. The algorithm generates the $n!$ permutations in constant time per permutation, using a linear array of n processors, each with a constant amount of storage space. The algorithm is cost optimal, in view of the $\Omega(nn!)$ operations required to generate the $n!$ permutations.

7.5 PROBLEMS

7.1 Prove the correctness of the sorting algorithm in Section 7.1.1.

7.2 One interesting feature of the algorithm in Section 7.1.1 is that, once input is complete and output starts, the array can begin processing a new input sequence of n elements. This is useful when several sequences are queued for sorting. Discuss the changes required in order for the algorithm to be able to handle m consecutive input sequences, and analyze the performance of the modified algorithm.

7.3 Another variant of the algorithm of Section 7.1.1 allows both P_1 and P_n to handle input and output. While P_1 is producing output, P_n can receive input, and conversely. Sorted sequences are produced alternately in nondecreasing order (through P_1) and in nonincreasing order (through P_n). Analyze the performance of this algorithm in the case where m sequences are queued for sorting.

7.4 Consider the following algorithm for sorting a sequence of numbers $S = \{s_1, s_2, \ldots, s_n\}$ on a linear array of n processors $P_1, P_2, \ldots, P_n$. Initially, P_i holds s_i, $1 \leq i \leq n$. As in Section 7.1.1, let "compare-exchange (P_i, P_{i+1})" denote the operation of comparing the numbers held by P_i and P_{i+1} and

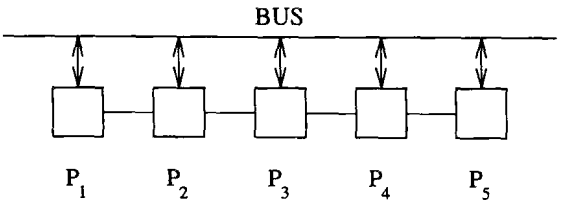

Figure 7.27: Linear array augmented with a bus.

then placing the smaller number in P_i and the larger in P_{i+1}. The algorithm, known as *odd-even transposition sort*, consists of two steps that are performed repeatedly:

Step 1: for $i = 1, 3, \ldots, 2\lfloor n/2 \rfloor - 1$ **do in parallel**
compare-exchange (P_i, P_{i+1})
end for
Step 2: for $i = 2, 4, \ldots, 2\lfloor (n-1)/2 \rfloor$ **do in parallel**
compare-exchange (P_i, P_{i+1})
end for. ■

Show that after $\lceil n/2 \rceil$ repetitions of the preceding two steps, in that order, P_i holds the ith smallest number. (*Hint*: Refer to Section 8.1.2.)

7.5 The algorithm in Problem 7.4 uses n processors and runs in $O(n)$ time, for a cost of $O(n^2)$, which is not optimal. Suppose instead that $\log n$ processors are used, each capable of storing $n/\log n$ elements of the input sequence. The 'compare-exchange (P_i, P_{i+1})' operation is now replaced with a 'merge-split (P_i, P_{i+1})' operation. Assuming that each of P_i and P_{i+1} holds a sorted subsequence, this operation merges the two subsequences into one and then splits the latter into two halves, the first going to P_i and the second to P_{i+1}. Provide a complete description of the new algorithm, and analyze its running time and cost.

7.6 One limitation of the linear array is its large diameter: It takes $n - 1$ steps for a datum to travel from P_1 to P_n on an n-processor array. In an attempt to circumvent this weakness, a variant of the model is used in which a further communication path, known as a *bus*, is available in addition to the usual links connecting the processors. The setup is shown in Fig. 7.27 for $n = 5$. At any given time during the execution of an algorithm, exactly one processor is allowed to *broadcast* a datum to the other processors using the bus. All processors receive the datum simultaneously. The time required by the broadcast operation is assumed to be constant; thus to go from P_1 to P_n, a datum now takes one time unit. The bus can also be used to provide input to the array

from the outside world. An input sequence $S = \{s_1, s_2, \ldots, s_n\}$ is to be sorted on this model so that when the algorithm terminates, P_i holds the ith smallest element. Show how this can be done using the following approach to sorting: An element occupies position i of the sorted sequence if exactly $i-1$ elements are found to be smaller than it. Analyze the resulting algorithm's running time.

7.7 As defined in Section 2.3.1, a *ring* is a linear array of processors $P_1, P_2, \ldots, P_m$ in which P_1 and P_m are directly connected. Show how a ring can be used to compute an $m \times 1$ vector $\boldsymbol{v}$ resulting from multiplying an $m \times n$ matrix $\boldsymbol{A}$ and an $n \times 1$ vector $\boldsymbol{u}$. Compare the running time of your solution to that of the algorithm in Section 7.2.1.

7.8 Design an algorithm for multiplying two $n \times n$ matrices on a linear array of processors, and analyze the running time, number of processors, and cost of the algorithm.

7.9 An *upper triangular* matrix is a square matrix, all of whose elements below the main diagonal are 0. Design a parallel algorithm for solving the system of equations $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is a given $n \times n$ upper triangular matrix, $\boldsymbol{b}$ is a given $n \times 1$ vector, and $\boldsymbol{x}$ is an $n \times 1$ vector of unknowns.

7.10 Two methods were mentioned at the end of Section 7.2.2 to improve the utilization of processors:

(a) Using $n/2$ processors instead of n.

(b) Solving two systems simultaneously, using n processors.

Provide the details of these two methods.

7.11 Describe a dual to the algorithm of Section 7.3.2 in which the w's move twice as fast as the x's, and analyze the performance of the new algorithm.

7.12 Describe a dual to the algorithm of Section 7.3.4 in which the x's move twice as fast as the y's, and analyze the performance of the new algorithm.

7.13 Describe an algorithm for performing convolution on a linear array such that inputs, weights, and outputs all move during each time unit. Analyze your algorithm.

7.14 Consider an array of processors $P_1, P_2, \ldots, P_k$, where P_k is the leftmost processor. A convolution can be performed on this array as follows: Let P_i store w_i. The inputs $x_1, x_2, \ldots, x_n$ are fed into P_k and travel to the right. When x_1 reaches P_1, each P_i holds x_i and w_i; it computes $w_i x_i$ and sends the result into an adder attached to the linear array, where $y_1 = w_1 x_1 + w_2 x_2 + \cdots + w_k x_k$ is computed. The sequence of x's is now shifted to the right, discarding

x_1 and bringing in x_{k+1}. Again, all processors compute a product, and the adder produces y_2. This continues until y_{n+1-k} has been obtained. Discuss this algorithm and analyze its performance, paying particular attention to the design of the adder.

7.15 Consider the linear array augmented with a bus, as described in Problem 7.6. It is desired to solve the convolution problem on this model. Let the array consist of k processors $P_1, P_2, \ldots, P_k$ (the leftmost processor being P_1), with P_i holding w_i. The outputs $y_1, y_2, \ldots, y_{n+1-k}$ are fed into P_1 (with y_1 entering first) and travel down the array from left to right. The bus is used to provide the sequence $\{x_1, x_2, \ldots, x_n\}$ as input to the array. Give the details of this algorithm and analyze its performance.

7.16 A variant of the approach in Problem 7.15 uses a ring (instead of an array) augmented with a bus. Here, P_i holds y_i (which, as in Section 7.3.1, stands for y_{i+sk}, where $s = 0, 1, \ldots, \lfloor \frac{n+1-k-i}{k} \rfloor$), while the bus brings $x_1, x_2, \ldots, x_n$ into the array, in that order. The weights $w_1, w_2, \ldots, w_k$, are shifted cyclically around the ring so that, at each step, each weight enters a processor. Specify what computations are performed by the processors, and analyze the resulting algorithm.

7.17 Describe an algorithm for performing convolution on a mesh-of-trees interconnection network in $O(\log k)$ time using $n \times k$ processors.

7.18 The following definition of convolution (slightly different from the one in Section 7.3) is sometimes used: Given two sequences $\{a_1, a_2, \ldots, a_n\}$ and $\{b_1, b_2, \ldots, b_n\}$, a sequence $\{y_1, y_2, \ldots, y_{2n-1}\}$ is computed from

$$y_i = \sum_{j=1}^{n} a_{i-j+1} b_j, \qquad 1 \leq i \leq 2n-1,$$

where $a_k = 0$ if $k \leq 0$ or $k > n$. For example, when $n = 3$,

$$\begin{aligned} y_1 &= a_1 b_1, \\ y_2 &= a_1 b_2 + a_2 b_1, \\ y_3 &= a_1 b_3 + a_2 b_2 + a_3 b_1, \\ y_4 &= a_2 b_3 + a_3 b_2, \\ y_5 &= a_3 b_3. \end{aligned}$$

Express this computation as a matrix-by-vector product, and show how it can be performed on a linear array.

7.19 Show how algorithm LINEAR ARRAY BINARY COUNTER of Section 7.4.1 can be used to generate all *subsets* of a set S of m elements.

7.20 Design an algorithm for the linear array that generates subsets of a set in the *minimal-change* order (as defined in Section 7.4.4). In other words, each subset should differ from the previous one in the least possible way.

7.21 An *n-subset* of a set of m elements is a subset with exactly n members. Combinations of n out of m elements are n-subsets. Show how an algorithm for generating binary strings can be used to generate n-subsets.

7.22 Given a sequence $S = \{s_1, s_2, \ldots, s_m\}$ of arbitrary elements, an *n-variation* of S is a string $v_1 v_2 \ldots v_n$ such that $v_i \in S$ for all $1 \leq i \leq n$. Note that repeated elements are allowed. For example, $DCABA$ and $ACCDD$ are both 5-variations of $\{A, B, C, D\}$. The number of variations of n elements out of m is m^n. Special instances of m-variations are binary and decimal counters, where $S = \{0, 1\}$, and $S = \{0, 1, \ldots, 9\}$, respectively.

(a) Design an algorithm for generating all n-variations of $S = \{0, 1, \ldots, m-1\}$.

(b) Extend your algorithm in (a) to the case where the elements of S are arbitrary symbols, all stored in the local memory of one processor.

7.23 Algorithm LINEAR ARRAY COMBINATIONS of Section 7.4.3 works for the case where $S = \{1, 2, \ldots, m\}$. Design a linear array algorithm for generating all n-combinations of a set of m arbitrary elements. One of the processors is allowed to store all the elements of S.

7.24 A *composition* of a positive integer m into n parts (or *n-composition*) is any sequence $\{x_1, x_2, \ldots, x_n\}$ of positive integers such that $x_1 + x_2 + \cdots + x_n = m$. Show how an algorithm for generating n-combinations of m elements can be used to generate all n-compositions of m.

7.25 Show how an algorithm for generating all subsets of a set can be used to generate compositions of an integer into *any* number of parts.

7.26 Design a parallel algorithm for generating compositions of an integer m, given that the largest part in each composition is k, for some integer $k \leq m$.

7.27 A combination of n out of m elements *with repetitions* is an unordered set of n elements taken from a set of m elements such that elements are allowed to repeat. In other words, $x_1 x_2 \ldots x_n$ is a combination with repetitions from $S = \{1, 2, \ldots, m\}$ if and only if $1 \leq x_1 \leq x_2 \leq \cdots \leq x_n \leq m$. Design an algorithm for generating combinations with repetitions on a linear array of processors.

7.28 Use the *split-and-plan* technique to design an algorithm for generating all permutations of n arbitrary elements on a linear array of n processors, in *lexicographic order.*

7.29 Let S be a set of m arbitrary elements. An *n-permutation* of S is obtained by selecting n distinct elements of S and arranging them in some order. (See Example 7.9.) Two n-permutations are distinct if they differ with respect to the elements they contain or with respect to the order of the elements. Design a parallel algorithm for generating all n-permutations of S.

7.30 Design a linear array algorithm for generating *random* permutations of n elements.

7.31 A permutation of the sequence $S = \{s_1, s_2, \ldots, s_n\}$ is a *derangement* if s_i does not appear in (its identity) position i, for all i, $1 \leq i \leq n$. Thus, for $n = 5$, $s_2s_3s_5s_1s_4$ and $s_5s_4s_2s_3s_1$ are derangements. There are $D_n = (n-1)(D_{n-1} + D_{n-2})$ derangements of n elements, with $D_0 = 1$ and $D_1 = 0$, requiring $\Omega(nD_n)$ operations to be generated. Describe an algorithm that uses the split-and-plan technique to generate all derangements of n arbitrary elements on a linear array of n processors.

7.32 A *partition* of an integer n is given by a sequence $\{x_1, x_2, \ldots, x_m\}$ of positive integers, where $x_1 \geq x_2 \geq \cdots \geq x_m$ and $x_1 + x_2 + \cdots + x_m = n$. Design a parallel algorithm for generating all partitions of an integer:

(a) Into m parts.

(b) Into any number of parts.

7.33 Let S be a set of n elements. An *equivalence relation* (or *partition*) of S consists of subsets $S_1, S_2, \ldots, S_k$ whose union is equal to S, such that the intersection of any two subsets is empty. Design a parallel algorithm for generating all equivalence relations of a set.

7.34 An *m-ary* tree is a data structure with n nodes that either is empty (i.e., $n = 0$) or consists of a root and m disjoint children, each of which is the root of an m-ary subtree. Design a parallel algorithm for generating all m-ary trees with n nodes.

7.35 A sequence of n left parentheses and n right parentheses is *balanced* if the number of right parentheses encountered when scanning from left to right never exceeds the number of left parentheses. (See Problem 4.31.)

(a) Show how an algorithm for generating binary trees on n nodes can be used to generate all balanced sequences of (n left and n right) parentheses.

(b) Design a parallel algorithm different from the one in **(a)** for generating balanced sequences of parentheses.

7.36 Suppose that each instance of a combinatorial object consists of n elements. When n processors are available on a linear array, each processor can be made responsible for producing one element. This approach was used in this chapter.

Now, let the number of processors be N, where $N < n$ or $N > n$. In this case, an *adaptive* algorithm is needed that adjusts its behavior according to the number of available processors. Discuss various approaches to obtaining adaptive algorithms for generating combinatorial objects in parallel.

7.37 In this chapter, when deriving a lower bound on the number of operations required to generate all instances of a combinatorial object, we took into account the number of operations needed to actually produce as output each instance in full. For example, all permutations of n elements require $\Omega(nn!)$ operations to be generated. An alternative definition simply takes into account the number of operations required to "create" each instance, without actually producing it in full as output. According to this second definition, the lower bound on generating all permutations of n elements is $\Omega(n!)$, since it may be possible to "create" each permutation from the previous one using a constant number of operations. Discuss approaches to designing parallel algorithms (for generating combinatorial objects) that are cost optimal under the new definition.

7.38 Given two n-bit numbers, show how they can be multiplied on a $2n$-processor linear array, and analyze the running time of your algorithm.

7.39 Extend the algorithm of Problem 7.38 so that the linear array multiplies *two* pairs of n-bit numbers in about the same time as it multiplies one pair.

7.40 Design an algorithm for dividing an integer x by an integer y on a linear array of processors. Make any appropriate assumptions about the input and the form of the output.

7.41 Pattern matching in strings is the problem of determining whether a string S_1 of m symbols occurs within a string S_2 of n symbols. For example, $S_1 = ABA$ occurs within $S_2 = AACCAABAC$. Design an algorithm for solving this problem on a linear array of processors.

7.42 Let W be a string of symbols from a given finite alphabet. The *reverse* of W, denoted W^R, consists of the symbols of W listed backwards. Given a string W, design an algorithm for determining, on a linear array of processors, whether $W = W^R$.

7.43 Given two sequences X and Y of symbols, a third sequence Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y. For example, if $X = \{a, c, b, c, d, a\}$ and $Y = \{b, a, c, c, b, a\}$, then $Z = \{c, c, a\}$ is a common subsequence of X and Y. Design a linear array algorithm that finds the longest common subsequence of two given sequences X and Y.

7.44 Given a sequence $X = \{x_1, x_2, \ldots, x_n\}$ of distinct integers, an *increasing subsequence* of X is a subsequence $\{x_i, x_j, \ldots, x_k\}$, where $i < j < \cdots < k$

and $x_i < x_j < \cdots < x_k$. Design a linear array algorithm that finds the longest increasing subsequence of a given sequence X.

7.45 Let $X = x_1x_2 \ldots x_n$ and $Y = y_1y_2 \ldots y_m$ be two strings of symbols from a finite alphabet. It is required to change X symbol by symbol, until it turns into Y. This is done by insertion, deletion, or replacement of symbols. It is required to minimize the number of single-symbol changes. Design a linear array algorithm to solve this problem.

7.46 Design an algorithm for computing the discrete Fourier transform on a linear array, and analyze the running time of the algorithm.

7.47 Given two sequences of weights $\{w_0, w_1, \ldots, w_h\}$ and $\{v_1, v_2, \ldots, v_k\}$, a sequence of initial values $\{y_{-k}, y_{-k+1}, \ldots, y_{-1}\}$, and an input sequence $\{x_{-h}, x_{-h+1}, \ldots, x_0, x_1, \ldots, x_n\}$, a process known as *filtering* calls for computing the output sequence $\{y_0, y_1, \ldots, y_n\}$ whose elements are defined by

$$y_i = \sum_{j=0}^{h} w_j x_{i-j} + \sum_{j=1}^{k} v_j y_{i-j}.$$

Design a linear array algorithm for filtering.

7.48 Let $U = \{u_0, u_1, \ldots, u_{n-1}\}$ and $V = \{v_0, v_1, \ldots, v_{n-1}\}$ be two given sequences of values. The *correlation* between U and V is defined as

$$R = \frac{nA - BC}{((nD - B^2)(nE - C^2))^{1/2}},$$

where

$$A = \sum_{j=0}^{n-1} u_j v_j, \quad B = \sum_{j=0}^{n-1} u_j, \quad C = \sum_{j=0}^{n-1} v_j, \quad D = \sum_{j=0}^{n-1} u_j^2, \quad \text{and } E = \sum_{j=0}^{n-1} v_j^2.$$

Design an algorithm for computing R on a linear array.

7.49 Describe a linear array algorithm for computing the convex hull of a set of n points in the plane (as defined in Example 4.7)

7.6 BIBLIOGRAPHICAL REMARKS

Interest in linear arrays as a viable model of parallel computation was fostered by the pioneering work of Kung [346, 347, 348, 349]. Linear array algorithms for sorting are described in Akl [18, 21], Akl and Schmeck [52], Baudet and Stevenson [92], Chen et al. [147], Goodman and Hedetniemi [263], Knuth [325], Kumar and Hirschberg [342], Kung [348], Lee et al. [362], Miranker et al. [430], Orton et

al. [455], Thompson and Kung [612], Todd [613], and Yasuura et al. [647]. Algorithm LINEAR ARRAY COMPARISON-EXCHANGE SORT was first proposed in Lee et al. [362] and Miranker et al. [430]. The algorithm of Section 7.1.2 is derived in Todd [613]. Algorithms for multiplying a matrix by a vector, solving tridiagonal systems of equations, performing convolution, filtering, computing Fourier transforms, and performing related computations on a linear array are described in Kung and Leiserson [351].

The design of algorithms to generate combinatorial objects has long fascinated mathematicians and computer scientists. Some of the earliest papers on the interplay between mathematics and computer science are devoted to the subject; see, for example, Lehmer [364] and Thompkins [609]. Because of their many applications in science and engineering, combinatorial algorithms continue to receive much attention, and interest has naturally been paid to the development of parallel algorithms for generating combinatorial objects; see, for example, Akl [20], Akl et al. [41], Chan and Akl [134], Cosnard and Ferreira [186], and Gupta and Bhattacharjee [275]. Several linear array algorithms have been proposed, including algorithms for counters (see Akl et al. [32]), combinations (see Akl et al. [37], Chen and Chern [144], Elhage and Stojmenović [217], Lin and Tsay [385], and Stojmenović [587]), permutations (see Akl et al. [46], Akl and Stojmenović [53], Chen and Chern [144], and Lin [384]), derangements (see Akl et al. [27]), binary trees (see Akl and Stojmenović [54]), t-ary trees (see Akl and Stojmenović [59]), integer partitions and compositions (see Akl and Stojmenović [55]), subsets and set partitions (see Djokić et al. [206]), and equivalence relations (see Stojmenović [586]). Algorithms for generating combinatorial objects at random are given in Rajan et al. [522] and Stojmenović [588]. The algorithms of Sections 7.4.1, 7.4.3, and 7.4.4 were first proposed in Akl et al. [32], Akl et al. [37], and Akl and Stojmenović [53], respectively. A survey of linear array algorithms for generating combinatorial objects is provided in Akl and Stojmenović [58].

Other algorithms for linear arrays are described in Asano and Umeo [68], Atallah and Tsay [75], Chaudhuri [140], Chazelle [141], Chen et al. [145], Holey and Ibarra [299], Kung and Lam [350], Leighton [367], Moldovan [432], Quinn [515], Quinton and Robert [517], Robert [537], and Ullman [615].