2

Basic Techniques

The task of designing parallel algorithms presents challenges that are considerably more difficult than those encountered in the sequential domain. The lack of a well-defined methodology is compensated by a collection of techniques and paradigms that have been found effective in handling a wide range of problems. This chapter introduces these techniques as they apply to a selected set of combinatorial problems, which are interesting on their own and often appear as subproblems in numerous computations.

It is important to view the introduced techniques as general guidelines for designing parallel algorithms, rather than as a manual of directly applicable methods. The combinatorial problems discussed include the prefix sums (Section 2.1), parallel prefix (Section 2.2), the convex hull (Section 2.3), merging (Section 2.4), insertions into 2-3 trees (Section 2.5), computing the maximum (Section 2.6), and coloring the vertices of a directed cycle (Section 2.7). Most of these parallel algorithms are used frequently in the remainder of this book.

2.1 Balanced Trees

The PRAM algorithm to compute the sum of n elements presented in Section 1.3.2 is based on a balanced binary tree whose leaves are the given n elements

and whose internal nodes represent additions. This algorithm is an example of the general strategy of *building a balanced binary tree on the input elements and traversing the tree forward and backward to and from the root*. An internal node u usually holds information concerning the data stored at the leaves of the subtree rooted at u. The success of such a strategy depends partly on the existence of a fast method to determine the information stored in an internal node from the information stored in its children. We use the computation of the prefix sums of n elements to provide another illustration of this strategy.

2.1.1 AN OPTIMAL PREFIX-SUMS ALGORITHM

Consider a sequence of n elements $\{x_1, x_2, \ldots, x_n\}$ drawn from a set S with a binary **associative** operation, denoted by $*$. The **prefix sums** of this sequence are the n partial sums (or products) defined by

$$s_i = x_1 * x_2 * \ldots * x_i, \quad 1 \leq i \leq n.$$

A trivial sequential algorithm computes s_i from s_{i-1} with a single operation by using the identity $s_i = s_{i-1} * x_i$, for $2 \leq i \leq n$, and hence takes $O(n)$ time. Clearly, this algorithm is inherently sequential.

We can use a balanced binary tree to derive a fast parallel algorithm to compute the prefix sums. Each internal node represents the application of the operation $*$ to its children during a forward traversal of the tree. Hence, each node v holds the sum of the elements stored in the leaves of the subtree rooted at v. During a backward traversal of the tree, the prefix sums of the data stored in the nodes at a given height are computed.

We start by giving a recursive version described by the steps needed to obtain the data stored in the nodes at height 1 and the steps needed to obtain the prefix sums after the recursive call on the nodes at height 1 terminates.

ALGORITHM 2.1

(Prefix Sums)

Input: *An array of* $n = 2^k$ *elements* $(x_1, x_2, \ldots, x_n)$, *where* k *is a nonnegative integer.*

Output: *The prefix sums* s_i, *for* $1 \leq i \leq n$.

begin

1. **if** $n = 1$ **then** $\{set\ s_1 := x_1$; **exit**$\}$
2. **for** $1 \leq i \leq n/2$ **pardo**
   Set $y_i := x_{2i-1} * x_{2i}$
3. *Recursively, compute the prefix sums of* $\{y_1, y_2, \ldots, y_{n/2}\}$, *and store them in* $z_1, z_2, \ldots, z_{n/2}$.

4. **for** $1 \leq i \leq n$ **pardo**
   $\{i$ *even* : *set* $s_i := z_{i/2}$
   $i = 1$: *set* $s_1 := x_1$
   i *odd* > 1 : *set* $s_i := z_{(i-1)/2} * x_i\}$

end

EXAMPLE 2.1:

The prefix-sums algorithm on eight elements is illustrated in Fig. 2.1. The time units referred to in what follows are those indicated in the figure. During the first time unit, the four elements $y_1 = x_1 * x_2, y_2 = x_3 * x_4, y_3 = x_5 * x_6$, and $y_4 = x_7 * x_8$ are computed. The second time unit corresponds to computing $y'_1 = y_1 * y_2$ and $y'_2 = y_3 * y_4$ with a recursive call to handle these two inputs. The element $y''_1 = y'_1 * y'_2$ is computed during time unit 3. Hence, at the fourth time unit, the prefix sum of this single input is generated. The reverse process begins by generating, in time unit 5, the prefix sums z'_1 and z'_2 of the two inputs y'_1 and y'_2 generated during the second time unit. Similarly, during time unit 6, the prefix sums z_1, z_2, z_3, and z_4 of the four elements y_1, y_2, y_3, and y_4 are generated. Finally, the prefix sums $\{s_i\}$ of the x_i's are generated in time unit 7. □

On inputs of size $n = 2^k$, the prefix-sums algorithm requires $2k + 1$ time units such that, during the first k time units, the computation advances from the leaves of a complete binary tree to the root, where the leaves hold $x_1, x_2, \ldots, x_n$. During the last k time units, we transverse the tree in reverse order and compute prefix sums by using the data generated during the first k time units.

We note that the whole algorithm can be executed in place, and that we introduced additional variables for clarity.

We are now ready to examine the following theorem, which is stated within the WT presentation level.

Theorem 2.1: *The Prefix-Sums Algorithm (Algorithm 2.1) computes the prefix sums of* n *elements in time* $T(n) = O(\log n)$, *using a total of* $W(n) = O(n)$ *operations.*

Proof: The correctness proof will be by induction on k, where the size of the input is $n = 2^k$.

The base case $k = 0$ is handled correctly by step 1 of the algorithm.

Assume that the algorithm works correctly for all sequences of length $n = 2^k$, where $k > 0$. We will prove that the algorithm computes the prefix sums of any sequence of length $n = 2^{k+1}$.

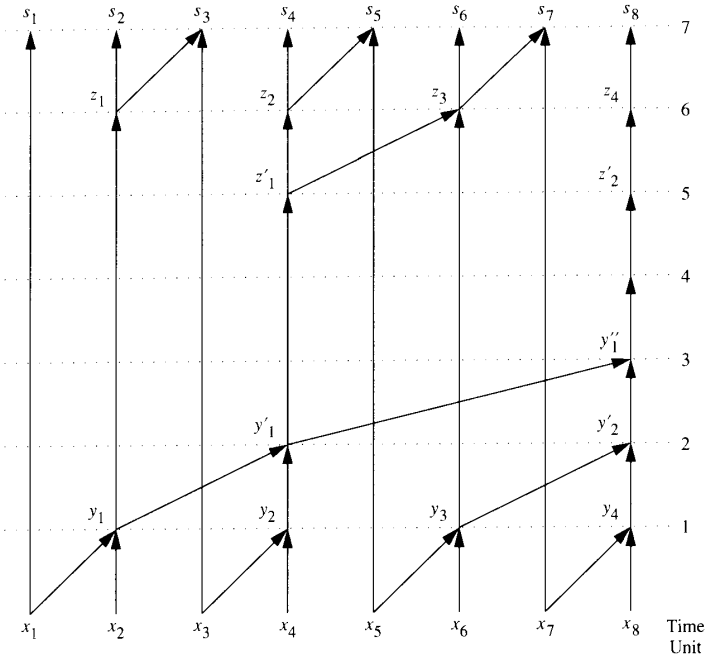


FIGURE 2.1
Prefix sums of eight elements. Each node represents a $*$ operation on the data stored in the tails of the incident arcs. During each time unit, the operations that take place are indicated by the presence of nodes with incoming arrows.

By the induction hypothesis, the variables $z_1, z_2, \ldots, z_{n/2}$, computed at step 3, hold the prefix sums of the sequence $\{y_1, y_2, \ldots, y_{n/2}\}$, where $y_i = x_{2i-1} * x_{2i}$, for $1 \le i \le n/2$. In particular, $z_j = y_1 * y_2 * \cdots * y_j$ and hence $z_j = x_1 * x_2 * \cdots * x_{2j-1} * x_{2j}$. That is, each z_j is nothing but the prefix sum s_{2j}, for $1 \le j \le n/2$. Thus, if i is even—say, $i = 2j$—then we have that $s_i = z_{i/2}$; otherwise, either $i = 1$ or $i = 2j + 1$, for some $1 \le j \le (n/2) - 1$. The case when $i = 1$ is trivial. For the remaining case of $i = 2j + 1$, we have $s_i = s_{2j+1} = s_{2j} * x_{2j+1} = z_{(i-1)/2} * x_i$. Therefore, all these cases are handled appropriately in step 4 of the algorithm. It follows that the algorithm works correctly on all inputs.

As for the resources required, they can be estimated as follows. Step 1 takes $O(1)$ sequential time. Steps 2 and 4 can be executed in $O(1)$ parallel steps using $O(n)$ operations. Therefore, the running time $T(n)$ and the work $W(n)$ required by the algorithm satisfy the following recurrences:

$$T(n) = T\left(\frac{n}{2}\right) + a$$

$$W(n) = W\left(\frac{n}{2}\right) + bn,$$

where a and b are constants. The solutions of these recurrences are $T(n) = O(\log n)$ and $W(n) = O(n)$. □

PRAM Model: Since steps 1, 2 and 4 of Algorithm 2.1 do not require concurrent read or concurrent write capability, this algorithm runs on the EREW PRAM model. A lower bound of $\Omega(\log n)$ on the time it takes a CREW PRAM to compute the Boolean OR of n variables (regardless of the number of operations used) implies that the previous algorithm is *WT optimal*, or *optimal in the strong sense, on the EREW and CREW PRAM*. This lower-bound proof is presented in Chapter 10. □

2.1.2 A NONRECURSIVE PREFIX-SUMS ALGORITHM

We present a nonrecursive version of the prefix algorithm, which will be used in Section 2.1.3 to illustrate the details involved in adapting the WT scheduling principle.

Let $A(i) = x_i$, where $1 \le i \le n$. Let $B(h,j)$ and $C(h,j)$ be sets of auxiliary variables, where $0 \le h \le \log n$ and $1 \le j \le n/2^h$. The array B will be used to record the information in the binary tree nodes during a forward traversal, whereas the array C will be used during the backward traversal of the tree. Figure 2.2 illustrates the forward traversal, and Fig. 2.3 illustrates the backward traversal, for $n = 8$.

ALGORITHM 2.2
(Nonrecursive Prefix Sums)
Input: *An array A of size $n = 2^k$, where k is a nonnegative integer.*
Output: *An array C such that $C(0, j)$ is the jth prefix sum, for $1 \le j \le n$.*
begin
1. **for** $1 \le j \le n$ **pardo**
   Set $B(0, j) := A(j)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \le j \le n/2^h$ **pardo**
   Set $B(h, j) := B(h - 1, 2j - 1) * B(h - 1, 2j)$

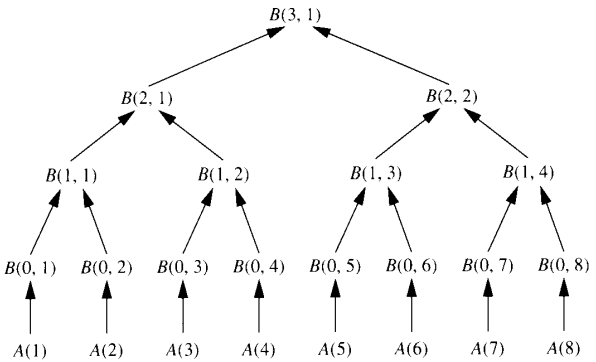


FIGURE 2.2
The bottom-up (forward) traversal of the binary tree used in the nonrecursive prefix sums algorithm. $B(0, j)$ is initially set to $A(j)$, and each internal node represents the operation $*$.

3. **for** $h = \log n$ **to** 0 **do**
   for $1 \leq j \leq n/2^h$ **pardo**
   $\{j$ *even* : *Set* $C(h, j) := C(h + 1, \frac{j}{2})$
   $j = 1$: *Set* $C(h, 1) := B(h, 1)$
   j *odd* > 1 : *Set* $C(h, j) := C(h + 1, \frac{j-1}{2}) * B(h, j)\}$

end

EXAMPLE 2.2:

Let $n = 8$ (see Figs. 2.2 and 2.3). We initially set $B(0, j) = A(j)$, for all $1 \leq j \leq 8$. The $B(0, j)$s correspond to the leaves of the binary tree. The variables $B(1, j)$, where $1 \leq j \leq 4$, correspond to the internal nodes at height 1; the variables $B(2, j)$, where $1 \leq j \leq 2$, correspond to the internal vertices at height 2. The root of the binary tree is stored in $B(3, 1)$. We then traverse this binary tree backward. We start by setting $C(3, 1) = B(3, 1)$. At the next step, we generate $C(2, 1) = B(2, 1)$ and $C(2, 2) = B(3, 1)$. Hence, $C(2, 1)$ and $C(2, 2)$ hold the prefix sums corresponding to the inputs $B(2, 1)$ and $B(2, 2)$. Similarly, $C(1, 1)$, $C(1, 2)$, $C(1, 3)$, and $C(1, 4)$ are the prefix sums of $B(1, 1)$, $B(1, 2)$, $B(1, 3)$, and $B(1, 4)$. Therefore the $C(0, j)$s hold the prefix sums of the original inputs. □

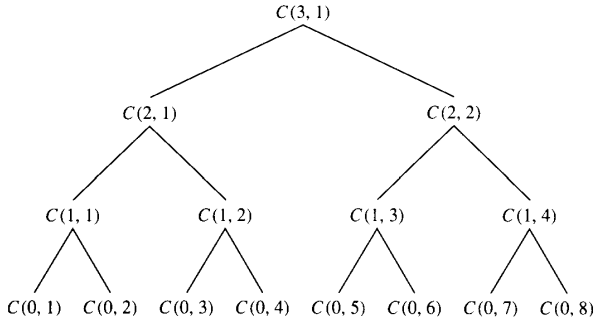


FIGURE 2.3
The elements of the array C as generated by the top-down (backward) traversal of the binary tree corresponding to the nonrecursive prefix-sums algorithm.

2.1.3 *ADAPTATION OF THE WT SCHEDULING PRINCIPLE

As indicated in Section 1.5, the adaptation of the WT scheduling principle requires the determination of the number of operations at each parallel step and a solution to the corresponding processor allocation problem.

Let $n = 2^k$. There are $2 \log n + 2 = 2k + 2$ parallel steps in Algorithm 2.2. Let $W_{1,1}$ be the number of operations performed at step 1, and let $W_{i,m}$ be the number of operations performed at step i in Algorithm 2.2 during the mth iteration, $i = 2, 3$. Then, $W_{1,1} = n$, $W_{2,m} = n/2^m = 2^{k-m}$ for $1 \leq m \leq k$, and $W_{3,m} = 2^m$ for $0 \leq m \leq k$. The total number of operations is given by $W(n) = W_{1,1} + \sum_{m=1}^{k} W_{2,m} + \sum_{m=0}^{k} W_{3,m} = n + 2^k \sum_{m=1}^{k} 2^{-m} + \sum_{m=0}^{k} 2^m = n + n(1 - (1/n)) + 2n - 1 = O(n)$. The processor allocation problem is addressed next.

Let our PRAM have $p = 2^q \leq n$ processors $P_1, P_2, \ldots, P_p$, and let $l = n/p = 2^{k-q}$. The input array is divided into p subarrays such that processor P_s is responsible for processing the sth subarray $A(l(s - 1) + 1), A(l(s - 1) + 2), \cdots, A(ls)$. At each height h of the binary tree, during either a forward or a backward traversal, the generation of the $B(h, \cdot)$ and $C(h, \cdot)$ values is divided in a similar way among the p processors. This division is performed in a way similar to the details we worked out for the parallel algorithm to compute the sum of n numbers in Example 1.13.

In Algorithm 2.3, which follows, conditions 2.1 and 3.1 hold whenever the number of operations at that particular iteration is greater than or equal

to p. These operations are then distributed equally among the p processors. In the case where the number of operations is less than p (steps 2.2 and 3.2), we assign one operation per processor starting from the lowest-indexed processor. Note that, for any given value of h in the loops defined in steps 2 and 3, the possible number of concurrent operations is $n/2^h = 2^{k-h}$.

The algorithm to be executed by the sth processor is given next. Figure 2.4 illustrates the corresponding processor allocation in the case of $n = 8$ and $p = 2$.

ALGORITHM 2.3

(Algorithm for Processor P_s)

Input: *An array A of size $n = 2^k$, and an index s that satisfies $1 \leq s \leq p = 2^q$, where $p \leq n$ is the number of processors.*

Output: *The prefix sums $C(0, j)$ for $\frac{n}{p}(s-1) + 1 \leq j \leq \frac{n}{p}s$.*

begin

1. **for** $j = 1$ **to** $l = n/p$ **do**
   Set $B(0, l(s-1) + j) := A(l(s-1) + j)$
2. **for** $h = 1$ **to** k **do**
   2.1. **if** $(k - h - q \geq 0)$ **then**
   for $j = 2^{k-h-q}(s-1) + 1$ **to** $2^{k-h-q}s$ **do**
   Set $B(h, j) := B(h-1, 2j-1) * B(h-1, 2j)$
   2.2. **else** {**if** $(s \leq 2^{k-h})$ **then**
   Set $B(h, s) := B(h-1, 2s-1) * B(h-1, 2s)$}
3. **for** $h = k$ **to** 0 **do**
   3.1. **if** $(k - h - q \geq 0)$ **then**
   for $j = 2^{k-q-h}(s-1) + 1$ **to** $2^{k-q-h}s$ **do**

$$\begin{cases} j \text{ even} & : \text{Set } C(h,j) := C\left(h+1, \frac{j}{2}\right) \\ j = 1 & : \text{Set } C(h,1) := B(h,1) \\ j \text{ odd} > 1 & : \text{Set } C(h,j) := C\left(h+1, \frac{j-1}{2}\right) * B(h,j) \end{cases}$$

   3.2. **else** {**if** $(s \leq 2^{k-h})$ **then**

$$\begin{cases} s \text{ even} & : \text{Set } C(h,s) := C\left(h+1, \frac{s}{2}\right) \\ s = 1 & : \text{Set } C(h,1) := B(h,1) \\ s \text{ odd} > 1 & : \text{Set } C(h,s) := C\left(h+1, \frac{s-1}{2}\right) * B(h,s) \end{cases}\}$$

end

EXAMPLE 2.3:

Let $n = 8$ and let $p = 2$. Consider the algorithm corresponding to processor P_2. At step 1, P_2 sets $B(0, 5) = A(5)$, $B(0, 6) = A(6)$, $B(0, 7) = A(7)$, and $B(0, 8) = A(8)$ (see Figs. 2.2 and 2.4). During the execution of step 2 of

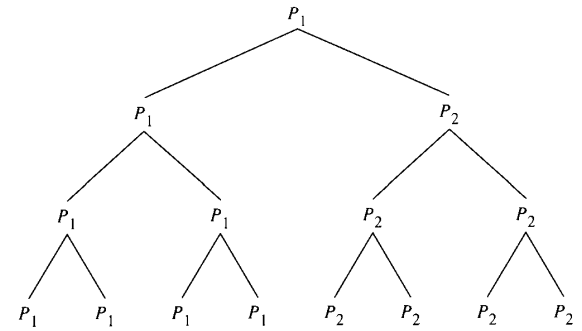


FIGURE 2.4
Processor allocation for computing the prefix sums of eight elements with two processors. The processor indicated at each node is responsible for computing the data generated at that node.

Algorithm 2.3, P_2 will be active for $h = 1, 2$ and idle for $h = 3$. Processor P_2 generates $B(1, 3)$, $B(1, 4)$ and $B(2, 2)$ during the loop defined by step 2. Similarly, during the backward traversal of the binary tree, P_2 will be idle for $h = 3$, and active for $h = 2, 1, 0$. Hence, P_2 generates the sums $C(2, 2)$, $C(1, 3)$, $C(1, 4)$, $C(0, 5)$, $C(0, 6)$, $C(0, 7)$, and $C(0, 8)$ (see Fig. 2.3). □

2.1.4 REVIEW

The basic scheme to build a balanced binary tree on the inputs and to traverse the binary tree to or from the root leads to efficient algorithms for many simple problems. This scheme is one of *the most elementary and the most useful parallel techniques*. We shall use this technique frequently in the rest of this book. Broadcasting a value to all the processors, and compacting the labeled elements of an array, are two simple examples that can be handled efficiently by this scheme (see Exercises 1.8 and 2.3).

The scheme using the balanced binary tree can be generalized to arbitrary balanced trees, where the number of children of an internal node could be nonconstant. As in the binary tree case, a fast algorithm is needed to determine the data stored in an internal node from the data stored in that node's children. It turns out that such a strategy can be carried out successfully for a number of specialized combinatorial problems. Computing the maximum of n elements is one such example discussed in Section 2.6.

2.2 Pointer Jumping

A **rooted-directed tree** T is a directed graph with a special node r such that (1) every $v \in V - \{r\}$ has outdegree 1, and the outdegree of r is 0, and (2) for every $v \in V - \{r\}$, there exists a directed path from v to r. The special node r is called the **root** of T. It follows that a rooted directed tree is a directed graph whose undirected version is a rooted tree such that each arc of T is directed from a node to that node's parent.

The *pointer jumping technique* allows the fast processing of data stored in the form of a set of rooted-directed trees. This technique is best introduced with an example.

2.2.1 FINDING THE ROOTS OF A FOREST

Let F be a forest consisting of a set of rooted directed trees. The forest F is specified by an array P of length n such that $P(i) = j$ if (i, j) is an arc in F; that is, j is the parent of i in a tree of F. For simplicity, if i is a root, we set $P(i) = i$. *The problem is to determine the root $S(j)$ of the tree containing the node j, for each j between 1 and n.*

A simple sequential algorithm—say, one based on first identifying the roots and reversing the links of the trees, followed by performing a depth-first or breadth-first traversal of each tree from its root—solves this problem in linear time. Our fast parallel algorithm follows a completely different strategy.

Initially, the *successor* $S(i)$ of each node i is defined to be the node $P(i)$. The technique of **pointer jumping** (or **path doubling**) consists of *updating the successor of each node by that successor's successor.* As the technique is applied repeatedly, the successor of a node is an ancestor that becomes closer and closer to the root of the tree containing that node. As a matter of fact, the distance between a node and its successor *doubles* unless the successor of the successor node is a root. Hence, after k iterations, the distance between i and $S(i)$ as they appear in a directed tree of F is 2^k unless $S(i)$ is a root. The detailed algorithm is given next.

ALGORITHM 2.4
(Pointer Jumping)
Input: *A forest of rooted directed trees, each with a self-loop at its root, such that each arc is specified by $(i, P(i))$, where $1 \le i \le n$.*
Output: *For each vertex i, the root $S(i)$ of the tree containing i.*
begin
1. **for** $1 \le i \le n$ **pardo**
   Set $S(i) := P(i)$
   while $(S(i)) \neq S(S(i))$ **do**
      Set $S(i) := S(S(i))$
end

EXAMPLE 2.4:

An illustration of the pointer jumping algorithm is shown in Fig. 2.5. In this case, the forest consists of two trees: one is rooted at vertex 8, and the other is rooted at vertex 13. The arcs in the figure correspond to $(i, P(i))$, $1 \le i \le 13$. The first execution of the **while** loop causes vertices 1, 2, 3, 4, 5, 9, 10, and 11 to change their successors, as indicated in Fig. 2.5(b). The second iteration causes the two trees to become of depth 1. We now have $S(i) = S(S(i))$, for all i; hence, the algorithm terminates. □

Let h be the maximum height of any tree in the forest. We can show the correctness of the previous procedure by using an inductive proof on h.

As for the time analysis, the distance (number of arcs in the initial trees) between i and $S(i)$ doubles after each iteration until $S(S(i))$ is the root of the tree containing i. Hence, the number of iterations is $O(\log h)$. Each iteration can be executed in $O(1)$ parallel time with $O(n)$ operations. Therefore, the algorithm runs in $O(\log h)$ time and uses a total of $W(n) = O(n \log h)$ operations, which is clearly nonoptimal (unless h is a constant), since a linear-time sequential algorithm exists.

Theorem 2.2: *Given a forest of rooted directed trees, Algorithm 2.4 generates for each vertex i the root of i's tree. This algorithm runs in $O(\log h)$ time using a total number of $O(n \log h)$ operations, where h is the maximum height of any tree in the forest, and n is the total number of vertices in the forest.* □

PRAM Model: Algorithm 2.4 requires concurrent-read capability because different nodes could have the same S value. The first iteration of Example 2.4 requires, for instance, that we set $S(3) = S(6)$, $S(4) = S(6)$, and $S(5) = S(6)$; hence, the datum $S(6)$ is used for the execution of three operations. However, no concurrent-write capability is needed. Hence, Algorithm 2.4 is a CREW PRAM algorithm. □

2.2.2 PARALLEL PREFIX

Let us assume next that each node i in the forest F contains a weight $W(i)$. The pointer jumping technique can be also used to compute, for each node i, the sum of the weights stored in the nodes on the path from the node i to the root

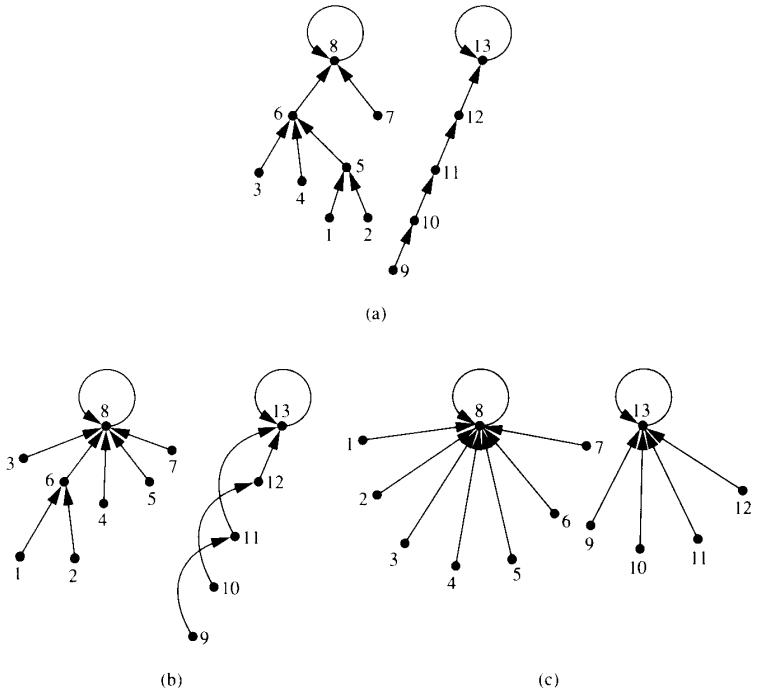


FIGURE 2.5
Illustration of pointer jumping. (a) The initial forest; (b) the forest after the first iteration; (c) the forest after the second iteration.

of i's tree. This computation amounts to generating the prefix sums of several sequences of elements such that each sequence is given by the order of the nodes as they appear on each path.

The next algorithm provides the details.

ALGORITHM 2.5

(Parallel Prefix on Rooted Directed Trees)

Input: *A forest of rooted directed trees, each with a self-loop at its root such that (1) each arc is specified by* $(i, P(i))$, *(2) each vertex* i *has a weight* $W(i)$, *and (3) for each root* r, $W(r) = 0$.

Output: *For each vertex* i, $W(i)$ *is set equal to the sum of the weights of vertices on the path from* i *to the root of its tree.*

begin

1. **for** $1 \leq i \leq n$ **pardo**

Set $S(i)$: $= P(i)$

while $(S(i)) \neq S(S(i))$ **do**

Set $W(i)$: $= W(i) + W(S(i))$

Set $S(i)$: $= S(S(i))$

end

EXAMPLE 2.5:

Let us go back to the example of Fig. 2.5. Suppose that, initially, $W(i) = 1$ for all $i \neq 8, 13$, and $W(8) = W(13) = 0$. During the first iteration of the loop, we obtain $W(1) = W(2) = W(3) = W(4) = W(5) = W(9) = W(10) = W(11) = 2$. The other vertices retain their initial W values. Now consider what happens to vertices 1 and 9 during the second iteration. Their associated values change as follows: $W(1) = W(1) + W(6) = 3$, and $W(9) = W(9) + W(11) = 4$. Similarly, we obtain that $W(2) = W(10) = 3$, and $W(3) = W(4) = W(5) = W(11) = 2$, and $W(6) = W(7) = W(12) = 1$. Hence, each $W(i)$ value corresponds to the length of the path from i to the root of its tree; that is, $W(i)$ is equal to the level of i, assuming the root is at level 0. □

PRAM Model: As in Algorithm 2.4, Algorithm 2.5 runs on the CREW PRAM. The bounds on the resources are also asymptotically the same as those of Algorithm 2.4. □

Remark 2.1: An important special case is when each tree is just a path represented by a **linked list.** The problem of computing prefix sums on linked lists is called **parallel prefix,** and is considered in detail in Chapter 3. We note here that Algorithm 2.5 provides a solution to the parallel prefix problem with n nodes such that the running time is $O(\log n)$ and the total number of operations is $O(n \log n)$, since the maximum height h is equal to n in this case. □

2.2.3 REVIEW

The pointer jumping technique provides a simple and powerful method for processing data stored in linked lists or directed rooted trees. In spite of the fact that Algorithms 2.4 and 2.5 are nonoptimal, the pointer jumping technique is useful in general because it is simple and can effectively handle subproblems arising in many computational tasks. These subproblems are usually of a size small enough that the pointer jumping technique will allow optimal overall processing. It is also possible to use the pointer jumping

technique in combination with other techniques to achieve optimality. In the next chapter, we present such an algorithm that solves the parallel prefix problem on linked lists optimally. The pointer jumping technique will be used heavily in Chapter 5.

2.3 Divide and Conquer

The basic **divide-and-conquer** strategy consists of three main steps. *The first step is to partition the input into several partitions of almost equal sizes. The second step is to solve recursively the subproblem defined by each partition of the input.* Note that these subproblems can be solved concurrently in the parallel framework. *The third step is to combine or merge the solutions of the different subproblems into a solution for the overall problem.*

The success of such a strategy depends on whether or not we can perform the first and third steps efficiently. This strategy has been shown to be effective in the development of fast sequential algorithms. It also leads to the most natural way of exploiting parallelism. We illustrate this technique on the convex-hull problem.

2.3.1 THE CONVEX-HULL PROBLEM

Given a set $S = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane, each represented by its (x, y) coordinates, the **planar convex hull** of S is the smallest convex polygon containing all the n points of S. A polygon Q is **convex** if, given any two points p and q in Q, the line segment whose endpoints are p and q lies entirely in Q. The convex-hull problem is to determine the ordered (say, clockwise) list $CH(S)$ of the points of S defining the boundary of the convex hull of S.

EXAMPLE 2.6:

Consider the set S of points shown in Fig. 2.6. In this case, the convex hull of S is given by $CH(S) = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$. □

The convex-hull problem is an important basic problem in computational geometry that arises in a variety of contexts. Chapter 6 is devoted to the study of several basic problems in planar computational geometry.

A simple divide-and-conquer strategy can be used to solve the convex-hull problem. This approach leads to an $O(n \log n)$ time sequential algorithm. On the other hand, it is not difficult to show that sorting can be reduced to the

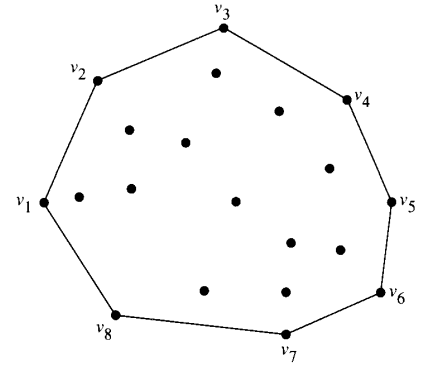


FIGURE 2.6
The convex hull of the set of points shown is the ordered list $(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$.

problem of solving the convex-hull problem (see Exercise 2.16); therefore the sequential complexity of the convex-hull problem is $T^*(n) = \Theta(n \log n)$.

Our parallel algorithm, based on the same divide-and-conquer strategy, will use an optimal number of operations.

2.3.2 A PARALLEL ALGORITHM FOR THE CONVEX-HULL PROBLEM

Let p and q be the points of S with the smallest and the largest x coordinates, respectively. Clearly, p and q belong to $CH(S)$ and partition $CH(S)$ into an **upper hull** $UH(S)$ consisting of all points from p to q of $CH(S)$ (clockwise), and a **lower hull** $LH(S)$ defined similarly from q to p. The upper and the lower hulls of the set of points introduced in Example 2.6 are given by $UH(S) = (v_1, v_2, v_3, v_4, v_5)$ and $LH(S) = (v_5, v_6, v_7, v_8, v_1)$.

For the remainder of this section, we concentrate on determining $UH(S)$. The problem of computing $LH(S)$ can be solved in a similar fashion. Before proceeding, we need the fact stated in the next remark.

Remark 2.2: *Sorting n numbers can be done in $O(\log n)$ time on the EREW PRAM using a total of $O(n \log n)$ operations.* This important fact is established in Chapter 4 for the CREW PRAM model. In a few instances, we use this fact before Chapter 4. □

Assume for simplicity that no two points have the same x or y coordinates and that n is a power of 2.

We start by sorting the points p_i by their x coordinates. By Remark 2.2, this preprocessing step can be performed in $O(\log n)$ time using $O(n \log n)$ operations. Let $x(p_1) < x(p_2) < \cdots < x(p_n)$, where $x(p_i)$ is the x coordinate of p_i.

Let $S_1 = (p_1, p_2, \ldots, p_{\frac{n}{2}})$ and $S_2 = (p_{\frac{n}{2}+1}, \ldots, p_n)$. Suppose that $UH(S_1)$ and $UH(S_2)$ have been determined. The **upper common tangent** between $UH(S_1)$ and $UH(S_2)$ is the common tangent such that both $UH(S_1)$ and $UH(S_2)$ are below it.

EXAMPLE 2.7:

Consider the two upper hulls $UH(S_1)$ and $UH(S_2)$ in Fig. 2.7. In this case, the segment (a, b) is the upper common tangent. □

Determining the upper common tangent between $UH(S_1)$ and $UH(S_2)$ can be done in $O(\log n)$ sequential time by using a binary search method. We shall not elaborate on this further since, in Chapter 6, we present in detail a faster parallel algorithm. However, *we assume for the remainder of this section that we have an $O(\log n)$-time sequential procedure to compute the upper common tangent.*

Let $UH(S_1) = (q_1, \cdots, q_s)$ and $UH(S_2) = (q'_1, \cdots, q'_t)$ be the upper hulls of S_1 and S_2, respectively, given in a left-to-right order. Note that, in particular, $q_1 = p_1$ and $q'_t = p_n$. Suppose that the upper common tangent has

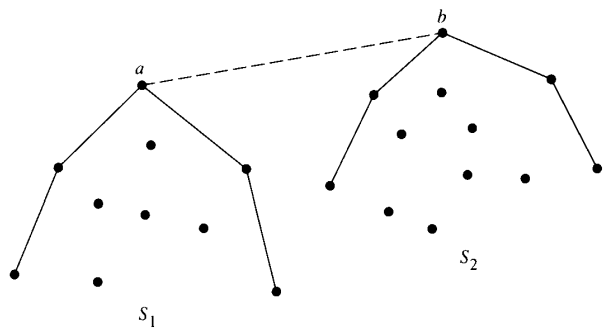


FIGURE 2.7
The line segment (a, b) is the upper common tangent of the upper hulls of S_1 and S_2.

been determined and is given by (q_i, q'_j). Then, $UH(S)$ is the array consisting of the first i entries of $UH(S_1)$ and the last $t - j + 1$ entries of $UH(S_2)$; that is, $UH(S) = (q_1, \ldots, q_i, q'_j, \ldots, q'_t)$. If s and t are given, once the indices i and j are known, $UH(S)$ and its size can be determined in $O(1)$ parallel time using a total of $O(n)$ operations.

PRAM Model: The method just given to combine $UH(S_1)$ and $UH(S_2)$ into $UH(S)$ requires a concurrent-read capability. The requirement is due to the fact that indices i and j defining the upper common tangent are needed to determine whether a point of $UH(S_1)$ or $UH(S_2)$ belongs to $UH(S)$, and, if it does, where it fits.

For example, in the time-processors framework, we can assign a processor to determine an entry of $UH(S)$ as follows. Processor P_k sets $UH(S)(k) := q_k$ whenever $k \leq i$, and sets $UH(S)(k) := q'_{k+j-i-1}$ whenever $i < k \leq i + t - j + 1$. In this case, a processor P_k may have to access i, j, and t. Hence, a CREW PRAM can be used to execute this step within the stated time bound. □

The overall algorithm is given next.

ALGORITHM 2.6
(Simple Upper Hull)
Input: *A set S of n points in the plane, no two of which have the same x or y coordinates such that $x(p_1) < x(p_2) < \cdots < x(p_n)$, where n is a power of 2.*
Output: *The upper hull of S.*
begin
1. If $n \leq 4$, then use a brute-force method to determine $UH(S)$, and exit.
2. Let $S_1 = (p_1, p_2, \cdots, p_{\frac{n}{2}})$ and $S_2 = (p_{\frac{n}{2}+1}, \ldots, p_n)$. Recursively, compute $UH(S_1)$ and $UH(S_2)$ in parallel.
3. Find the upper common tangent between $UH(S_1)$ and $UH(S_2)$, and deduce the upper hull of S.
end

Theorem 2.3: *Algorithm 2.6 correctly computes the upper hull of n points in the plane. This algorithm runs in $O(\log^2 n)$ time using a total of $O(n \log n)$ operations.*

Proof: The correctness proof follows from a simple induction on n, assuming the correctness of the procedure to determine the upper common tangent.

Suppose that Algorithm 2.6 takes $T(n)$ parallel steps using $W(n)$ operations on an arbitrary instance of size n. We define each of $T(n)$ and $W(n)$ by a recurrence relation as follows.

Step 1 takes $O(1)$ sequential time. Step 2 takes $T(n/2)$ time using $2W(n/2)$ operations. As for step 3, determining the upper common tangent can be done in $O(\log n)$ sequential time, as indicated previously, and the combination of $UH(S_1)$ and $UH(S_2)$ into $UH(S)$ can be performed in $O(1)$ parallel time using a total of $O(n)$ operations. Therefore, $T(n)$ and $W(n)$ satisfy the following recurrences:

$$T(n) \leq T\left(\frac{n}{2}\right) + a \log n$$

$$W(n) \leq 2W\left(\frac{n}{2}\right) + bn,$$

where a and b are positive constants. The solutions of these recurrence equations are given by $T(n) = O(\log^2 n)$ and $W(n) = O(n \log n)$, and the theorem follows. □

Corollary 2.1: *The convex hull of a set of n points can be determined in $O(\log^2 n)$ time using a total of $O(n \log n)$ operations. Hence, Algorithm 2.6 is optimal.*

PRAM Model: Algorithm 2.6 requires the CREW PRAM model, since our implementation of the merging of $UH(S_1)$ and $UH(S_2)$ does. In Exercise 2.17, the reader is asked to adapt this algorithm to run on the EREW PRAM model.□

Remark 2.3: In the time-processors framework, the running time of Algorithm 2.6 is $O\left(\frac{n \log n}{p} + \log^2 n\right)$, where p is the number of processors. Therefore, this algorithm achieves an optimal speedup for all values of p such that $p \leq n/\log n$. □

2.3.3 REVIEW

The divide-and-conquer strategy constitutes a powerful, widely applicable approach for developing efficient parallel algorithms. However, a straightforward divide-and-conquer approach does not lead to optimal $O(\log n)$ time algorithms, unless the merging can be performed efficiently. As a matter of fact, we shall see in Chapter 6 how to find the upper common tangent in $O(1)$ parallel time using only a total of $O(n)$ operations. This improvement implies an optimal $O(\log n)$ time algorithm for the convex-hull problem. If the merging step of a divide-and-conquer algorithm proves difficult to speed up, the following two methods provide alternatives.

The first method uses an **n^α divide-and-conquer strategy** by partitioning the input into n^α approximately equal partitions, rather than into a small number of partitions (as we did in the previous example). This strategy can indeed be used to obtain a solution to the convex-hull problem in $O(\log n)$ time using a total of $O(n \log n)$ operations (see Exercise 2.18).

The second approach is to try to **pipeline** the operations required for the merging steps. If successful, such a strategy may enable us to perform merging in $O(1)$ time using a linear number of operations. This **pipelined** or **cascading divide-and-conquer strategy** is used to derive the optimal $O(\log n)$ time merge-sort algorithm (Chapter 4), and to obtain the WT optimal algorithms for several computational geometry problems (Chapter 6). A simple use of the pipelining technique is discussed in Section 2.5.

2.4 Partitioning

The **partitioning strategy** *consists of (1) breaking up the given problem into p independent subproblems of almost equal sizes, and then (2) solving the p subproblems concurrently*, where p is the number of processors available. In its simplest form, this strategy amounts to splitting the input into p nonoverlapping pieces, and then solving the problems associated with the p pieces concurrently. However, in most cases, we make an effort to break up the given problem into a set of independent subproblems as illustrated by the problem of merging two sorted sequences, which we address next.

Given a set S with a partial order relation $\leq$ (that is, $\leq$ is reflexive, antisymmetric, and transitive), S is **linearly ordered** or **totally ordered,** if for every pair $a, b \in S$, either $a \leq b$ or $b \leq a$.

Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ be two nondecreasing sequences whose elements are drawn from a linearly ordered set S. We consider, in this section, the problem of **merging** these two sequences into a sorted sequence—say, $C = (c_1, c_2, \ldots, c_{2n})$.

Simple linear-time sequential algorithms to solve the merging problem are well known. Our goal here is to provide a parallel solution that is based on *partitioning A and B into many pairs of subsequences such that we can obtain the sorted sequence by concurrently merging the resulting pairs of subsequences.*

Compare this strategy with the divide-and-conquer strategy, where the main work required typically lies in combining the solutions of the subproblems involved, rather than in partitioning the input.

2.4.1 A SIMPLE MERGING ALGORITHM

We start with few definitions. Let $X = (x_1, x_2, \ldots, x_t)$ be a sequence whose elements are from the set S. Let $x \in S$. The **rank** of x in X, denoted by $rank(x:X)$, is the number of elements of X that are less than or equal to x. Let $Y = (y_1,$

$y_2, \ldots, y_s$) be an arbitrary array of elements from S. **Ranking** Y in X is the problem of determining the array $rank(Y:X) = (r_1, r_2, \ldots, r_s)$, where $r_i = rank(y_i : X)$.

EXAMPLE 2.8:

Let $X = (25, -13, 26, 31, 54, 7)$ and $Y = (13, 27, -27)$. Then, $rank(Y:X) = (2, 4, 0)$. □

Without loss of generality, assume that all the elements appearing in the two sorted sequences to be merged, A and B, are *distinct* (see Exercise 2.21). In particular, no element of A appears in B.

The merging problem can be viewed as that of determining the rank of each element x from A or B in the set $A \cup B$. If $rank(x : A \cup B) = i$, then $c_i = x$, where c_i is the ith element of the desired sorted sequence. Since $rank(x:A \cup B) = rank(x:A) + rank(x:B)$, we can solve the merging problem by determining the two integer arrays $rank(A : B)$ and $rank(B : A)$.

We now proceed to describe an algorithm to determine $rank(B:A)$. The same algorithm can be used to compute $rank(A : B)$. Let b_i be an arbitrary element of B. Since A is sorted, we can find the rank of b_i in A by using the **binary search method.** This method consists of comparing b_i with the middle element of A. Based on the outcome of this comparison, the search can be restricted to the upper or lower half of A. The process is repeated until b_i is isolated between two successive entries of A—that is, $a_{j(i)} < b_i < a_{j(i)+1}$, where $rank(b_i : A) = j(i)$. Note that we have used here the fact that the elements of A and B are distinct.

The binary search algorithm just sketched determines the rank of an arbitrary element of B in A, and runs in $O(\log n)$ sequential time. We can obviously apply this method concurrently to all the elements of B to obtain a parallel algorithm for ranking B in A whose running time is $O(\log n)$, which implies that we have an $O(\log n)$ time parallel algorithm for merging two sequences, each of length n. However, the total number of operations used by such an algorithm is $O(n \log n)$, which is nonoptimal, since linear-time sequential algorithms exist, as indicated at the beginning of this section.

2.4.2 AN OPTIMAL MERGING ALGORITHM

An optimal merging algorithm can be obtained as follows. Choose approximately $n/\log n$ elements of each of A and B that partition A and B into blocks of almost equal lengths. Apply the binary search method to rank each of the chosen elements in the other sequence. This step reduces the problem into

merging pairs of subsequences, each of which has $O(\log n)$ elements. We can then apply an optimal-time sequential algorithm to each pair of subsequences to generate the sorted sequence.

For simplicity, we present the details of a slightly different algorithm. We start with the following partitioning algorithm, which forms the main component of the overall algorithm. In Algorithm 2.7 we do not assume that the lengths of the two sequences are necessarily equal. Figure 2.8 illustrates the pairs of subsequences obtained.

ALGORITHM 2.7

(Partition)

Input: *Two arrays* $A = (a_1, \ldots, a_n)$ *and* $B = (b_1, \ldots, b_m)$ *in increasing order, where both* $\log m$ *and* $k(m) = m/\log m$ *are integers.*

Output: $k(m)$ *pairs* (A_i, B_i) *of subsequences of* A *and* B *such that (1)* $|B_i| = \log m$, *(2)* $\sum_i |A_i| = n$, *and (3) each element of* A_i *and* B_i *is larger than each element of* A_{i-1} *or* B_{i-1}, *for all* $1 \le i \le k(m) - 1$.

begin

1. *Set* $j(0) := 0, j(k(m)) := n$
2. **for** $1 \le i \le k(m) - 1$ **pardo**
   - 2.1. *Rank* $b_{i \log m}$ *in* A *using the binary search method, and let* $j(i) = rank(b_{i \log m} : A)$
3. **for** $0 \le i \le k(m) - 1$ **pardo**
   - 3.1. *Set* $B_i := (b_{i \log m + 1}, \ldots, b_{(i+1) \log m})$
   - 3.2. *Set* $A_i := (a_{j(i)+1}, \ldots, a_{j(i+1)})$

   (A_i *is empty if* $j(i) = j(i+1)$)

end

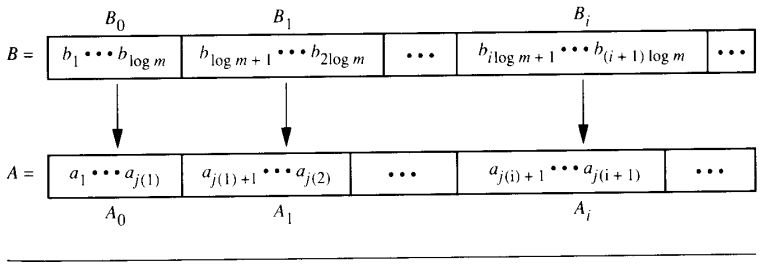


FIGURE 2.8
The partitions generated by the partition algorithm (Algorithm 2.7). Each B_i is of size $\log m$, and the A_i's could be of different sizes. The element $j(i)$ is given by $j(i) = rank(b_{i \log m} : A)$.

EXAMPLE 2.9:

Let $A = (4, 6, 7, 10, 12, 15, 18, 20)$ and let $B = (3, 9, 16, 21)$. In this case, $m = 4$ and $k(m) = 2$. Since $rank(9 : A) = 3$, we obtain the following two pairs of subsequences: (1) $A_0 = (4, 6, 7)$ and $B_0 = (3, 9)$, and (2) $A_1 = (10, 12, 15, 18, 20)$ and $B_1 = (16, 21)$. Each element of A_1 and B_1 is larger than each element in A_0 or B_0; hence, we can merge A and B by merging separately the pairs (A_0, B_0) and (A_1, B_1). □

Lemma 2.1: *Let C be the sorted sequence obtained by merging the two sorted sequences A and B, of lengths n and m, respectively. Then, Algorithm 2.7 partitions A and B into pairs of subsequences (A_i, B_i) such that $|B_i| = O(\log m)$, $\sum_i |A_i| = n$, and $C = (C_0, C_1, \cdots)$, where C_i is the sorted sequence obtained by merging A_i and B_i. Moreover, this algorithm runs in $O(\log n)$ time using a total of $O(n+m)$ operations.*

Proof: We first establish that each element in the subsequences A_i and B_i is larger than each element of A_{i-1} or B_{i-1}, for $1 \le i \le k(m) - 1$. The two smallest elements of A_i and B_i are, respectively, $a_{j(i)+1}$ and $b_{i \log m + 1}$, whereas the two largest elements of A_{i-1} and B_{i-1} are, respectively, $a_{j(i)}$ and $b_{i \log m}$. Since $rank(b_{i \log m} : A) = j(i)$, we have that $a_{j(i)} < b_{i \log m} < a_{j(i)+1}$. This result implies that $b_{i \log m + 1} > b_{i \log m} > a_{j(i)}$, and $a_{j(i)+1} > b_{i \log m}$. Therefore, each of the elements of A_i and B_i is larger than each of the elements in A_{i-1} or B_{i-1}; hence, the correctness of the algorithm follows.

The timing analysis can be done as follows. Step 1 takes $O(1)$ sequential time. Step 2 takes $O(\log n)$ time, since the binary search method is applied to all the elements in parallel. The total number of operations required to execute this step is $O((\log n) \times (m/\log m)) = O(m + n)$, since $(m \log n/\log m) < (m \log (n + m)/\log m) \le n + m$, for $n, m \ge 4$. Step 3 takes $O(1)$ parallel time using a linear number of operations. Hence, the algorithm runs in $O(\log n)$ time using a total of $O(n + m)$ operations. □

After applying Algorithm 2.7 to our merging problem (of two sequences, each of length n), we end up with an independent set of merging subproblems. This outcome is the essence of the partitioning strategy. Now, we would like to handle each merging subproblem in $O(\log n)$ time, such that the total number of operations used is proportional to the size of the subproblem. This running time can be achieved as follows.

Consider the merging subproblem corresponding to an arbitrary pair (A_i, B_i). Recall that $|B_i| = \log n$ for all indices i. If $|A_i| = O(\log n)$, we can merge the pair (A_i, B_i) in $O(\log n)$ sequential time by using an optimal sequential algorithm. Otherwise, we can apply Algorithm 2.7 to partition A_i into blocks each of which is of size $O(\log n)$ (in this case, A_i plays the role of B, and B_i plays the role of A). This step will take $O(\log \log n)$ time using $O(|A_i|)$ operations.

Therefore, we can make each of the subsequences to be of length $O(\log n)$ without asymptotically increasing the bounds on the resources. Finally, an optimal sequential algorithm for merging can be applied to each pair of subsequences to generate the desired sorted sequence. We therefore have the following theorem.

Theorem 2.4: *Let A and B be two sorted sequences, each of length n. Merging A and B can be done in $O(\log n)$ time, using a total of $O(n)$ operations.* □

PRAM Model: The binary search method is applied to the array A to rank several elements of B simultaneously in step 2 of Algorithm 2.7. Hence, concurrent-read capability is required by this algorithm; however, no concurrent-write capability is needed. The algorithm thus runs on the CREW PRAM model. □

2.4.3 REVIEW

We have presented a partitioning strategy to solve the merging problem efficiently. In Chapter 4, we shall see how to use this strategy with a parallel-search algorithm (as opposed to the binary search algorithm used before) to obtain an optimal merging algorithm that runs in $O(\log \log n)$ time. This algorithm is one of the fast optimal algorithms that does not require concurrent-write capability.

The partitioning strategy used in this section should be contrasted with the divide-and-conquer strategy. Both strategies have the same goal of decomposing the problem into a set of subproblems that can be solved in parallel. The main work in the divide-and-conquer strategy usually lies in the merging of the solutions of the subproblems, whereas the main work in the partitioning strategy lies in carefully decomposing the problem so that the solutions of the subproblems can be combined easily to generate the solution of the overall problem.

2.5 Pipelining

A **2-3 tree** is a rooted tree in which each internal node has two or three children, and every path from the root to a leaf is of the same length. The number of leaves in a 2-3 tree of height h is between 2^h and 3^h (see Exercise 2.27). Hence, if the number of leaves is n, the height of the tree is $\Theta(\log n)$.

A sorted list A of n items, $A = (a_1, a_2, \ldots, a_n)$, where $a_1 < a_2 < \cdots < a_n$, can be represented by a 2-3 tree T, where the leaves hold the data items in a left-to-right order. An internal node v will hold two data items, $L[v]$ and $M[v]$, where $L[v]$ and $M[v]$ are, respectively, the largest items stored in the first (leftmost) and the second subtrees of v. In addition, v holds a data item $R[v]$ representing the largest item in the third (rightmost) subtree of v, whenever v has a third child. The item $R[v]$ usually is not included in the definition of a 2-3 tree. We shall make use of it in a parallel insertion procedure introduced later in this section.

EXAMPLE 2.10:

Let A be the list $A = (1, 2, 3, 4, 5, 6, 7)$. A 2-3 tree representing A is shown in Fig. 2.9. We have used the notation $i : j : k$ next to each internal node v to indicate that $L[v] = i, M[v] = j$ and $R[v] = k$, whenever a rightmost subtree exists. Note that the 2-3 tree corresponding to A is not unique. □

2.5.1 BASIC OPERATIONS ON A 2-3 TREE

A 2-3 tree is a well-known data structure used to represent sets of elements subject to the operations of search, insert, and delete. More precisely, a 2-3 tree represents a dynamically changing set of elements induced by processing a sequence of instructions containing the operations of (1) searching for an element, (2) inserting an element, and (3) deleting an element, where the elements are drawn from a linearly ordered set. Notice that the set of elements represented by a 2-3 tree appear in sorted order, as discussed before,

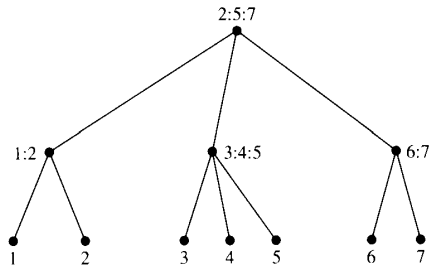


FIGURE 2.9
An example of a 2-3 tree. The notation $L[v] : M[v] : R[v]$ next to node v denotes respectively, the largest elements in the leftmost, the middle, and the rightmost (if it exists) subtrees.

and that the insert and delete operations require that the structure of the 2-3 tree be preserved after the operations are executed. Each of the search, insert, and delete operations can be performed in $O(\log n)$ time, where n is the size of the current set of elements. We next give a brief overview of the procedures related to the search and insert operations.

Let T be a 2-3 tree representing a list of n elements, $a_1 < a_2 < \cdots < a_n$. Given an element b, the problem of searching for b can be defined as the problem of finding the leaf u of T such that the data item a_i stored in u satisfies $a_i \leq b < a_{i+1}$.

This problem can be handled with a binary search method as follows. Let r be the root of T. The search for b can be restricted to one of the subtrees of r, depending on where b fits between the two data items $L[r]$ and $M[r]$. More precisely, if $b \leq L[r]$, the search continues in the leftmost subtree of r; otherwise, if $L[r] < b \leq M[r]$, the search continues in the second subtree. We handle the remaining case of $b > M[r]$ by restricting the search to the third subtree. The search procedure terminates when a leaf is reached. This process takes time proportional to the height of T.

We can insert an element b into the tree T by first applying the previous search procedure to locate b. Let u be the leaf identified by this search procedure. Node u contains an element a_i such that $a_i \leq b < a_{i+1}$. If $b = a_i$, there is nothing to be done, since b is already in the tree. Otherwise, we create a leaf u', which holds the data item b. We insert u' to the immediate right of u with the same parent—say p—as u. If p has three children, we stop here. Thus, let us assume that p has four children. We create a new node p' and allocate the children of p appropriately between p and p' (see Fig. 2.10). We can then insert the node p' into T by using the same procedure. That is, p' is tentatively assigned the same parent as p, which is in turn examined for a

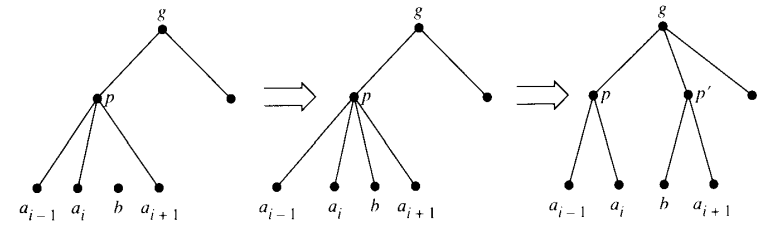


FIGURE 2.10
The process of inserting an item b into a 2-3 tree. A node containing b is created and assigned the same parent p as a_i thereby generating the middle tree. Since node p in the middle tree has four children, a new node p' is created, and is assigned the same parent as p, as illustrated in the rightmost tree.

possible violation of the restriction on the number of children. Finally, if the root r has four children, we create a new root with two children, each of which has the appropriate two children of r.

During the insertion procedure, the L, M, and R values can be adjusted appropriately without asymptotically increasing the sequential time of the procedure.

EXAMPLE 2.11:

Suppose we have to insert the data item 5 into the 2-3 tree T shown in Fig. 2.11(a). A leaf containing 5 is created with the node c as the parent (Fig. 2.11b). Since c has four children, a node c' is created whose children are the leaves containing 5 and 6 (Fig. 2.11c). But now the root has four children. Creating a new root with two children results in the 2-3 tree shown in Fig. 2.11(d). □

2.5.2 INSERTING A SEQUENCE OF ITEMS INTO A 2-3 TREE

Suppose that we are given a 2-3 tree T holding the n items $a_1 < a_2 < \cdots < a_n$. *We consider the problem of inserting a sequence of* k *items* $b_1 < b_2 < \cdots <$

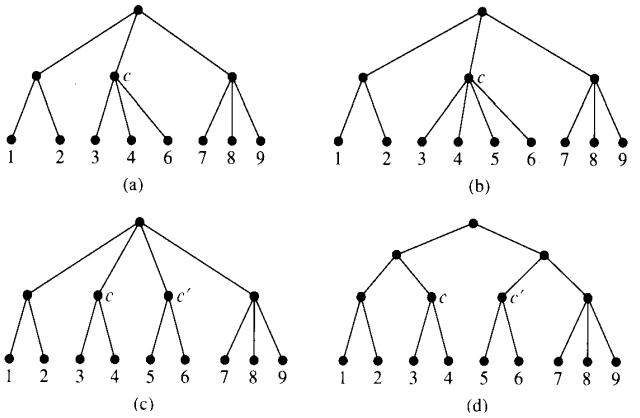


FIGURE 2.11
Steps for inserting data item 5 into a 2-3 tree. (a) A 2-3 tree. (b) and (c) Intermediate trees generated during insertion of the element 5 into the tree. (d) The final tree generated.

b_k *into* T, where k is assumed to be much smaller than n. If k is larger than n, then it is more efficient to construct the corresponding 2-3 tree from scratch (see Exercise 2.32).

Using the previous insertion procedure, we can insert the k elements, one at a time, in $O(k \log n)$ sequential time. In what follows, we describe a parallel algorithm that performs the insertion of the k elements in $O(\log n)$ time using a total of $O(k \log n)$ operations. A key component of this algorithm is the effective *pipelining* of several subsequences of insertions.

Briefly, the technique of **pipelining** *is the process of breaking up a task into a sequence of subtasks—say,* $t_1, \ldots, t_m$*—such that, once* t_1 *is completed, the sequence corresponding to a new task can begin and can proceed at the same rate as the previous task.* This process is similar to the operation of an assembly line, where a subtask could be the completion of a specific component of a system, and several systems are assembled on line.

We start with a preprocessing step that simplifies the presentation of the algorithm. We insert b_1 and b_k into T using the sequential insertion algorithm. This step ensures that each of the remaining b_i's fits between two consecutive leaves of T. This preprocessing takes $O(\log n)$ sequential time. Hence, we assume for the remainder of this section that T has been preprocessed in this fashion.

Concurrently, we locate all the elements b_i in T by applying the search procedure on each b_i separately. Hence, for each b_i we obtain a leaf containing the item $a_{i'}$ such that $a_{i'} \leq b_i < a_{i'+1}$. This search takes $O(\log n)$ parallel steps, using a total of $O(k \log n)$ operations.

PRAM Model: Locating all the b_i's in T requires a concurrent-read capability of the tree T; hence, this procedure runs on the CREW PRAM model. In Exercise 2.28, the reader is asked to show how to achieve the same performance on the EREW PRAM model. □

Let us call a **chain** the ordered set of elements among the b_i's that have to fit between two consecutive leaves of T. In particular, let B_i be the chain that has to fit between a_i and a_{i+1}, and let $|B_i| = k_i$, where $1 \leq i \leq n - 1$. Clearly, $\sum_i k_i = k - 2$ (since b_1 and b_k have been already inserted in T). We first consider the special case when $k_i \leq 1$, for all $1 \leq i \leq n - 1$.

The procedure of inserting the k elements consists of a bottom-up processing of T similar to the sequential insertion procedure, except that several nodes at the same level may be processed concurrently.

Initially, we create a leaf l_i holding b_i, and assign the appropriate parent to it, for each $1 \leq i \leq n - 1$. Some of the internal nodes at height 1 may now have more than three children, but none will have more than six. For each node v of more than three children, we create another node v' with the same parent as v, and reassign the children of v appropriately between v and v'. This step can be done concurrently for all the internal nodes at height 1 with

more than three children. This process can be repeated until we reach the root. The root is then handled just as in the sequential case.

Hence, inserting k elements into T can be performed in $O(\log n)$ time using $O(k \log n)$ operations in the case when $k_i \leq 1$, for $1 \leq i \leq n - 1$. At this point, the reader should verify that the data items stored in the nodes can be updated appropriately as we proceed with the insertion procedure. We call the algorithm just described the **simple parallel insertion algorithm.**

Consider the general case where the size k_i of each B_i is not necessarily less than or equal to 1. We apply the simple parallel insertion algorithm to the middle elements of the nonempty B_i's. The middle element of a nonempty chain $a_l, a_{l+1}, \cdots, a_f$ is the element a_s, where $s = \lceil (f + l)/2 \rceil$. Inserting these elements takes $O(\log n)$ time and, in addition, reduces the size of each chain by a factor of ½. We repeat this process on the newly formed chains. Note that the new chains can be trivially deduced from the old chains. More precisely, the chain $a_l, a_{l+1}, \ldots, a_f$ causes the creation of the two new chains $a_l, \ldots, a_{s-1}$, and $a_{s+1}, \ldots, a_f$, whenever $s - 1 \geq l$ and $s + 1 \leq f$, respectively. After $O(\log k)$ iterations, all the k elements are inserted in T. The overall time is $O(\log k \log n)$, since each iteration takes $O(\log n)$ time. However, we can obtain a faster algorithm by using the technique of *pipelining*.

Let us call each application of the simple parallel insertion procedure a **stage.** Hence, the algorithm consists of $\leq \lceil \log k \rceil$ stages such that, during the ith stage, a set of elements is inserted into the 2-3 tree T_{i-1} to obtain the 2-3 tree T_i, where initially $T_0 = T$. Each T_{i-1} is processed bottom-up, level by level. Hence, stage i can be viewed as a **wave** moving up T_{i-1}. Once the nodes at a certain level are processed, the next wave can move up to process the appropriate nodes at this level. Therefore, the different stages can be pipelined one after the other, up the tree. The running time of this method is reduced to $O(\log n + \log k) = O(\log n)$, using a total of $O(k \log n)$ operations.

PRAM Model: The parallel insertion procedure does not in itself require any simultaneous access to the same memory location. However, the method used to locate the elements b_i in T requires the CREW PRAM model. Hence, the overall insertion procedure runs on the CREW PRAM. □

2.5.3 REVIEW

Pipelining is an important parallel technique that has been used extensively in parallel processing. We shall encounter this technique in the development of an optimal $O(\log n)$ merge-sort algorithm, presented in Chapter 4, and in speeding up the divide-and-conquer algorithms for several problems in computational geometry, presented in Chapter 5.

2.6 Accelerated Cascading

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of elements drawn from a linearly ordered set S. The problem of computing the **maximum** element of X is to find an element x_i such that $x_i \geq x_j$, for all $1 \leq j \leq n$.

This basic problem can be solved by a simple linear-time sequential algorithm. On the PRAM, it offers an interesting case study for introducing the strategy of *accelerated cascading*.

In the remainder of this section, we assume that the x_i's are distinct. Otherwise, we can replace x_i by the pair (x_i, i), for each i, and extend the relation $\geq$ as follows: $(x_i, i) > (x_j, j)$ if and only if $x_i > x_j$, or $x_i = x_j$ and $i > j$.

We can determine the maximum element by using a balanced binary tree constructed on the n input elements, just as in the case of computing the sum of n elements (Algorithm 1.7). The running time of the parallel algorithm is $O(\log n)$, and the total number of operations used is $O(n)$. This parallel algorithm is optimal, since the work performed matches the sequential complexity of the problem.

We can obtain a *faster* algorithm by using a scheme based on a *doubly logarithmic-depth tree computation* (as opposed to the balanced binary tree of logarithmic depth). The main idea is to construct a balanced tree with the x_i's as the leaves such that the number of children of a node u is equal to $\lceil \sqrt{n_u} \rceil$, where n_u is the number of leaves in the subtree rooted at u. Each internal node will be used to hold the maximum element in its subtree. The condition on the number of children forces the tree to be of doubly logarithmic depth. The success of this strategy depends on the existence of a *constant-time algorithm* to perform the operation represented by an internal node—that is, a constant-time algorithm for computing the maximum of an arbitrary number of elements. Our next task is the development of such an algorithm.

2.6.1 A CONSTANT-TIME ALGORITHM FOR COMPUTING THE MAXIMUM

Let A be an array holding p elements from our linearly ordered universe S. The purpose of Algorithm 2.8 is to perform comparisons between all pairs of elements from A. The maximum element can be identified as the only element that comes out a "winner" in all its comparisons.

In the algorithm that follows, the Boolean array B holds the outcomes of all the comparisons, and the symbol $\wedge$ represents the Boolean AND operation.

ALGORITHM 2.8

(Basic Maximum)

Input: *An array A of p distinct elements.*

Output: *A Boolean array M such that $M(i) = 1$ if and only if $A(i)$ is the maximum element of A.*

begin

1. **for** $1 \le i, j \le p$ **pardo if** $(A(i) \ge A(j))$ **then** *set* $B(i, j) := 1$
   else *set* $B(i, j) := 0$
2. **for** $1 \le i \le p$ **pardo**
   Set $M(i) := B(i, 1) \wedge B(i, 2) \wedge \cdots \wedge B(i, p)$

end

When the algorithm terminates, $M(i) = 1$ if and only if $A(i)$ is the maximum element.

PRAM Model: Step 1 requires simultaneous read operations. Step 2 can be implemented in $O(1)$ time if we allow concurrent writes of the same value (see Exercise 1.5). Hence, Algorithm 2.8 can be implemented on the common CRCW PRAM in $O(1)$ time using $O(p^2)$ operations. □

Lemma 2.2: *The maximum of p elements can be computed on the common CRCW PRAM in $O(1)$ time using a total of $O(p^2)$ operations.* □

2.6.2 A DOUBLY LOGARITHMIC-TIME ALGORITHM

Given a rooted tree, the **level** of a node u is the number of edges on the path between u and the root of the tree. Hence, the level of the root is 0. We are ready to introduce the doubly logarithmic-depth tree.

For clarity, assume that $n = 2^{2^k}$, for some integer k. Define the **doubly logarithmic-depth tree** with n leaves as follows. The root of the tree has $2^{2^{k-1}}$ (that is, $\sqrt{n}$) children, each of its children has $2^{2^{k-2}}$ children, and, in general, each node at the ith level has $2^{2^{k-i-1}}$ children, for $0 \le i \le k - 1$. Each node at level k will have two leaves as children.

EXAMPLE 2.12:

The doubly logarithmic-depth tree for the case when $n = 16$ is shown in Fig. 2.12. The root has four children, and each of the other internal nodes has two children. Each internal node corresponds to computing the maximum of that node's children. □

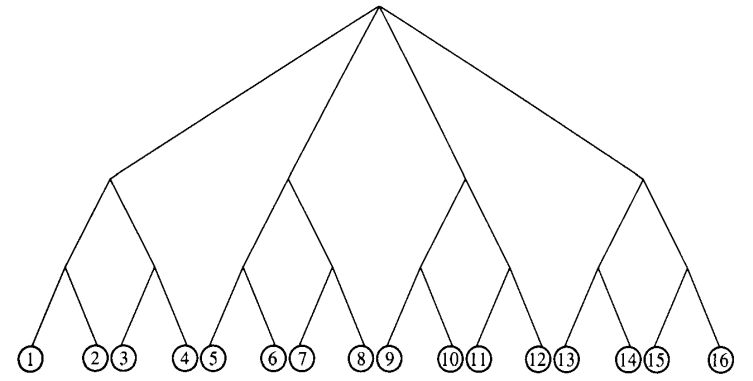


FIGURE 2.12
Doubly logarithmic-depth tree on 16 nodes.

A simple inductive argument on i shows that the number of nodes at the ith level of the doubly logarithmic-depth tree is $2^{2^k - 2^{k-i}}$, for $0 \le i < k$. The number of nodes at the kth level is $2^{2^k - 1} = n/2$. In addition, the depth of this tree is $k + 1 = \log \log n + 1$.

The doubly logarithmic-depth tree can be used for computing the maximum of n elements. Each internal node holds the maximum of the elements stored in its subtree. The algorithm proceeds level by level, bottom up, starting with the nodes at height 1. Using Algorithm 2.8, the maxima required at any given level can be computed in $O(1)$ time.

It follows that the maximum can be computed in $T(n) = O(\log \log n)$ time, so the doubly logarithmic-depth tree algorithm is *exponentially* faster than the previous $O(\log n)$ time algorithm (Algorithm 1.7).

We still need to estimate the total number of operations required by the new method. The number of operations required by the tasks at the ith level is $O((2^{2^{k-i-1}})^2)$ per node, for $0 \le i \le k$, giving a total of $O((2^{2^{k-i-1}})^2 \cdot 2^{2^k - 2^{k-i}}) = O(2^{2^k}) = O(n)$ operations per level. Hence, the total number of operations required by the overall computation is $W(n) = O(n \log \log n)$, which makes the algorithm nonoptimal.

2.6.3 MAKING THE FAST ALGORITHM OPTIMAL

In summary, we have introduced two algorithms for computing the maximum. The first, based on the logarithmic-depth binary tree, is optimal and runs in

logarithmic time, whereas the second algorithm is nonoptimal but runs in doubly logarithmic time ("very fast"). In such a case, we can try to use a strategy, called **accelerated cascading,** to combine the two algorithms into an optimal and a very fast algorithm. We first describe the accelerated-cascading strategy in general terms:

1. Start with the optimal algorithm until the size of the problem is reduced to a certain threshold value.
2. Then, shift to the fast but nonoptimal algorithm.

We now examine the adaptation of this strategy to the problem of computing the maximum of n elements.

In phase 1, we apply the binary tree algorithm, starting from the leaves and moving up to $\lceil \log \log \log n \rceil$ levels. Since the number of candidates reduces by a factor of ½ per level as we go up the binary tree, we know that the maximum is among the $n' = O(n/\log \log n)$ elements generated at the end of the binary tree algorithm. The total number of operations used so far is $O(n)$, and the corresponding time is $O(\log \log \log n)$.

During phase 2, we use the doubly logarithmic-depth tree based on the n' generated elements during phase 1. This second phase requires $O(\log \log n') = O(\log \log n)$ time and uses $W(n) = O(n' \log \log n') = O(n)$ operations. Therefore, the overall algorithm is optimal and runs in time $O(\log \log n)$.

Theorem 2.5: *Finding the maximum of n elements can be done optimally in* $O(\log \log n)$ *time on the common CRCW PRAM.* □

PRAM Model: The basic maximum algorithm (Algorithm 2.8) requires a concurrent write of the same value capability to run in constant time. Since our analysis is based on this assumption, our optimal $O(\log \log n)$ time algorithm requires the common CRCW PRAM model. A matching lower bound for the CRCW PRAM (regardless of whether it is common, arbitrary, or priority) is derived in Chapter 4 in the case where $O(n)$ processors are available. Therefore, *our* $O(\log \log n)$ *time algorithm is WT optimal for the CRCW PRAM model.*

On the other hand, $\Omega(\log n)$ time is required on the CREW PRAM regardless of the number of processors available (Chapter 10). Hence, *the balanced binary tree method yields a WT optimal EREW and CREW PRAM algorithm for computing the maximum.* □

Remark 2.4: An *important special case* of the accelerated-cascading technique is when the optimal algorithm used in phase 1 is an **optimal sequential** algorithm. In fact, we can obtain an optimal $O(\log \log n)$ time algorithm for computing the maximum of n elements by using this special case.

We partition the input into $n/\log \log n$ blocks $\{B_i\}$, each block containing approximately $\log \log n$ elements. We then use the optimal sequential algorithm to compute the maximum of each block, for all blocks concurrently. We then proceed with the doubly logarithmic-depth tree to generate the maximum within the claimed bounds. □

2.6.4 REVIEW

In this section, two general techniques were used to obtain the fast optimal algorithm to compute the maximum. The first consists of organizing the computation in a doubly logarithmic-depth tree. The second is the accelerated cascading of two algorithms, one optimal but relatively slow, the other very fast but nonoptimal. These two techniques seem to be useful for deriving very fast parallel optimal algorithms.

In an important special case, we (1) partition the input into nonoverlapping pieces, (2) solve the subproblems associated with the pieces concurrently by using an optimal sequential algorithm, and (3) combine the solutions of the subproblems by using a fast parallel algorithm.

2.7 Symmetry Breaking

Let $G = (V, E)$ be a directed cycle; that is, the indegree and the outdegree of each vertex is 1, and, for any two vertices $u, v \in V$, there exists a directed path from u to v. A ***k*-coloring** of G is a mapping $c : V \rightarrow \{0, 1, \cdots, k-1\}$, such that $c(i) \neq c(j)$ if $\langle i, j \rangle \in E$. We are interested in the problem of determining a 3-coloring of G.

A straightforward algorithm consists of traversing the cycle from an arbitrary vertex, while assigning alternate colors from $\{0, 1\}$ to adjacent vertices. A third color may be needed to terminate the cycle. This simple sequential algorithm is clearly optimal, but it does not seem to lead to a fast parallel algorithm.

The main difficulty in solving our coloring problem fast using parallelism lies in the apparent symmetry in the problem. Assigning a color to many vertices in parallel implies that these vertices have been somehow distinguished from the remaining vertices. But all vertices look alike; hence, some mechanism should be introduced to partition the vertices into classes such that each class can be assigned the same color.

This symmetric situation occurs in many other problems as well. In the remainder of this section, we present an almost constant-time algorithm to break this symmetry. Another technique for breaking symmetry using *randomization* is presented in Chapter 9.

2.7.1 A BASIC COLORING ALGORITHM

We make the following assumption regarding the input representation of the directed cycle $G = (V, E)$: The arcs of G are specified by an array S of length n such that $S(i) = j$ whenever $\langle i,j \rangle \in E$, for $1 \leq i,j \leq n$. Note that we can obtain the **predecessor** relation immediately by setting $P(S(i)) = i$, for $1 \leq i \leq n$.

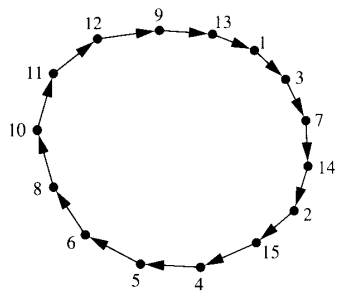
Suppose that an initial coloring c of the vertices of G is given (we can always start with $c(i) = i$ for all i). Given the binary expansion of an integer i—say, $i = i_{t-1} \cdots i_k \cdots i_1 i_0$—the **$k$th least significant bit** of i is bit i_k. We can reduce the number of colors by using the following simple procedure that takes advantage of the binary representation of the colors.

ALGORITHM 2.9
(Basic Coloring)
Input: *A directed cycle whose arcs are specified by an array S of size n and a coloring c of the vertices.*
Output: *Another coloring c' of the vertices of the cycle.*
begin
for $1 \leq i \leq n$ **pardo**
1. Set k to the least significant bit position in which $c(i)$ and $c(S(i))$ disagree.
2. Set $c'(i) := 2k + c(i)_k$
end

EXAMPLE 2.13:

Let $c(i) = i$ be the initial coloring assigned to the vertices of the directed cycle shown in Fig. 2.13. For each vertex, we list the value of k and its corresponding new color in two separate columns. The number of colors is reduced from 15 to six in this case. □

Lemma 2.3: *The output function c' generated by Algorithm 2.9 is a valid coloring whenever c is a valid coloring. The algorithm runs in $T(n) = O(1)$ time using $W(n) = O(n)$ operations.*



v	c	k	c'
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

FIGURE 2.13
The basic coloring algorithm on a directed cycle, where each arc is represented by $\langle i, S(i) \rangle$. The column c represents the initial coloring $c(i) = i$, k is the least significant bit position in which $c(i)$ and $c(S(i))$ differ, and $c'(i) = 2k + c(i)_k$ is the new coloring. The vertices $\{v\}$ are listed in the order they appear on the cycle.

Proof: Notice that, in the basic coloring procedure, the index k must always exist, since c is a coloring. Suppose that $c'(i) = c'(j)$ for some $\langle i,j \rangle \in E$. Then, $c'(i) = 2k + c(i)_k$ and $c'(j) = 2l + c(j)_l$, where k and l are determined as stated in the basic coloring algorithm. Since $c'(i) = c'(j)$, we must have $k = l$. But then $c(i)_k = c(j)_k$, which contradicts the definition of k. Hence, $c'(i) \neq c'(j)$ whenever $\langle i,j \rangle \in E$.

The bounds on $T(n)$ and $W(n)$ are obvious under the assumption that the least significant bit position in which two binary numbers differ can be found in $O(1)$ sequential time, whenever each of the two binary numbers is of size $O(\log n)$ bits. We avoid giving a justification of this assumption here, since it involves a discussion of the specific type of primitive operations allowed on each processor. □

2.7.2 A SUPERFAST 3-COLORING ALGORITHM

We start by estimating the number of colors generated by Algorithm 2.9 as a function of the number of initial colors.

Let $t > 3$ be the number of bits used to represent each of the colors in the initial coloring c. Then, each of the colors used by c' can be represented with $\lceil \log t \rceil + 1$ bits. Hence, if the number of colors in c is q, where q satisfies $2^{t-1} < q \leq 2^t$, coloring c' uses at most $2^{\lceil \log t \rceil + 1} = O(t) = O(\log q)$ colors. Therefore, the number of colors decreases exponentially in general.

The basic coloring procedure can be applied repeatedly. Further reduction of the number of colors occurs as long as the number t of bits used to represent the colors satisfies $t > \lceil \log t \rceil + 1$—that is, $t > 3$. Applying the basic coloring procedure to the case where $t = 3$ will result in a coloring with at most six colors, since the new color of i is given by $c'(i) = 2k + c(i)_k$, and hence $0 \leq c'(i) \leq 5$ (as $0 \leq k \leq 2$). We next estimate the number of iterations needed to reach this stage.

Before presenting the analysis on the number of iterations, we introduce the following notation. Let $\log^{(i)} x$ be defined by $\log^{(1)} x = \log x$, and $\log^{(i)} x = \log(\log^{(i-1)} x)$. For example, $\log^{(2)} x = \log \log n$, and $\log^{(3)} x = \log \log \log x$. Let $\log^* x = \min\{i \mid \log^{(i)} x \leq 1\}$. The function $\log^* x$ is an extremely slowly growing function that is bounded by 5 for all $x \leq 2^{65536}$.

Starting with the initial coloring $c(i) = i$ for $1 \leq i \leq n$, the first application of Algorithm 2.9 reduces the number of colors to $O(\log n)$ colors. The second application reduces the number of colors to $O(\log \log n) = O(\log^{(2)} n)$. Therefore, the number of colors will be reduced to less than or equal to six after $O(\log^* n)$ iterations.

We can reduce the number of colors to three as follows. The additional recoloring procedure consists of three iterations, each of which handles vertices of a specific color. For each $3 \leq l \leq 5$, we do the following. For each vertex i with color l, we recolor vertex i with the smallest possible color from $\{0, 1, 2\}$ (that is, smallest color different from the colors of its predecessor and its successor). Each iteration takes $O(1)$ time, using $O(n)$ operations. After the last iteration, we obtain a 3-coloring within the same asymptotic bounds.

EXAMPLE 2.14:

Let us apply the recoloring procedure to the example in Fig. 2.13. The colors of the vertices obtained after we apply Algorithm 2.9 are 0, 1, 2, 3, 4, 5. Vertex 6 is the only vertex of color 3. Since its neighbors, vertices 5 and 8, are colored 1 and 2, vertex 6 is recolored 0. Next, vertices 3 and 9 are recolored 0 and 1, respectively. Finally, vertices 13 and 14 are recolored 0 and 2, respectively. □

Theorem 2.6: *We can color the vertices of a directed cycle with three colors in $O(\log^* n)$ time, using $O(n \log^* n)$ operations.*

Proof: The procedure consists of $O(\log^* n)$ iterations of the basic coloring algorithm (Algorithm 2.9), followed by recoloring the vertices whose colors are between colors 3 and 5.

As we have seen before, the number of colors obtained after $O(\log^* n)$ iterations of Algorithm 2.9 is less than or equal to six. The recoloring procedure ensures that vertices of colors 4, 5, and 6 are recolored from the set $\{0, 1, 2\}$. The recoloring procedure works correctly because no adjacent vertices are recolored at the same time.

Obtaining the stated bounds on the resources is straightforward. □

PRAM Model: In Algorithm 2.9, simultaneous access to an entry $c(i)$ can be avoided by creating a copy of c and accessing the copy for the color of the successor. Therefore this algorithm runs on the EREW PRAM model. □

2.7.3 AN OPTIMAL 3-COLORING ALGORITHM

The previous 3-coloring algorithm in Section 2.7.2 breaks the symmetry in a directed cycle extremely fast, but uses a nonlinear number of operations. However, for all practical purposes, this algorithm takes less than six parallel steps, each of which involves simple operations on each of the vertices.

If we insist on optimality, a more involved variation of the procedure can be used. The remark that follows is needed for our optimal algorithm.

Remark 2.5: **(Sorting Integers)** Sorting n integers, each of which in the range $[0, O(\log n)]$, can be done in $O(\log n)$ time using a linear number of operations. This sorting can be accomplished by using a combination of the well-known radix-sorting algorithm and the prefix-sums algorithm. The reader is asked to provide the details in Exercise 2.45. □

Our optimal algorithm consists of (1) coloring the vertices with $O(\log n)$ colors using the basic coloring procedure, (2) sorting the vertices by their colors (hence, all vertices with the same color appear consecutively), and (3) applying a recoloring procedure successively on each set of vertices with the same color. The details are given in the following algorithm.

ALGORITHM 2.10

(3-Coloring of a Cycle)

Input: *A directed cycle of length n whose arcs are specified by an array S.*

Output: *A 3-coloring of the vertices of the cycle.*

begin

1. **for** $1 \leq i \leq n$ **pardo**
   Set $C(i) := i$
2. *Apply Algorithm 2.9 once.*
3. *Sort the vertices by their colors.*

4. **for** $i = 3$ **to** $2\lceil \log n \rceil$ **do**
   for *all vertices v of color i* **pardo**
   Color v with the smallest color from $\{0, 1, 2\}$ *that is different from the colors of its two neighbors.*

end

Theorem 2.7: *We can color the vertices of a directed cycle with three colors in* $O(\log n)$ *time, using a total of* $O(n)$ *operations.*

Proof: Note that, by Lemma 2.3 no two adjacent vertices will have the same color after the execution of step 2. Hence, no two vertices of the same color are adjacent. The recoloring performed at step 4 is therefore valid.

As for the time complexity, steps 1 and 2 take $O(1)$ time, using $O(n)$ operations, and step 3 takes $O(\log n)$ time, using $O(n)$ operations, in view of Remark 2.5. After sorting the vertices by their colors, we can assume that all the vertices with the same color are in consecutive memory locations, and that we know the locations of the first and the last vertices of each color (see Exercise 2.40). Let n_i be the number of vertices of color i. Recoloring these vertices takes $O(1)$ parallel time, using $O(n_i)$ operations. Therefore step 4 takes $O(\log n)$ time, using a total of $O(\sum_i n_i) = O(n)$ operations. □

PRAM Model: Algorithm 2.10 does not need any simultaneous memory access. Hence, it runs on the EREW PRAM model. □

2.7.4 REVIEW

A simple technique was presented in this section to perform a 3-coloring of the vertices of a cycle. This technique seems to break the symmetry present in some problems. Examples of the applications of this technique are given in the exercises at the end of this chapter, as well as in later chapters. Another technique for breaking symmetry uses randomization, a topic studied in Chapter 9.

2.8 Summary

In this chapter, we provided a collection of general methods for designing parallel algorithms. These methods are adapted for solving many of the problems encountered as we progress in this book. Our classification is somewhat arbitrary, in the sense that a given algorithm could sometimes be grouped easily under two or more of these methods. For example, the $O(\log n)$ time parallel algorithm for computing the sum of n numbers also could have been described as a divide-and-conquer algorithm.

The parallel algorithms developed for the specific combinatorial problems are commonly used as building blocks to solve more complex problems. Table 2.1 gives a summary of the most important algorithms.

TABLE 2.1
ALGORITHMS INTRODUCED IN THIS CHAPTER.

Algorithm		Section	Time	Work	Method	PRAM Model
2.1	Prefix Sums	2.1.1	$O(\log n)$	$O(n)$	Balanced binary tree	EREW
2.2	Nonrecursive Prefix Sums	2.1.2	$O(\log n)$	$O(n)$	Balanced binary tree	EREW
2.4	Pointer Jumping	2.2.1	$O(\log h)$	$O(n \log h)$	Pointer jumping	CREW
2.5	Parallel Prefix (Rooted Trees)	2.2.2	$O(\log h)$	$O(n \log h)$	Pointer jumping	CREW
2.6	Upper Hull	2.3.2	$O(\log^2 n)$	$O(n \log n)$	Divide and conquer	CREW
2.7	Partition	2.4.2	$O(\log n)$	$O(n + m)$	Partitioning	CREW
	Merging	2.4.2	$O(\log n)$	$O(n)$	Partitioning	CREW
2.8	Basic Maximum	2.6.1	$O(1)$	$O(n^2)$	All pairs comparisons	Common CRCW
	Fast, Optimal Maximum	2.6.2 and 2.6.3	$O(\log \log n)$	$O(n)$	Accelerated cascading	Common CRCW
2.9	Basic Coloring	2.7.1	$O(1)$	$O(n)$	Symmetry breaking	EREW
	Fast Coloring	2.7.2	$O(\log^* n)$	$O(n \log^* n)$	Symmetry breaking	EREW
2.10	3-Coloring of a Cycle	2.7.3	$O(\log n)$	$O(n)$	Symmetry breaking and integer sort	EREW

Exercises

2.1. Develop an optimal nonrecursive prefix-sums algorithm that is similar to Algorithm 2.2 but that does not use the auxiliary variables B and C. The input array A should hold the prefix sums when the algorithm terminates.

2.2. Consider the following divide-and-conquer algorithm for computing the prefix sums $\{s_i\}$ of a sequence $x_1, x_2, \ldots, x_n$, where n is a power of 2. Compute the prefix sums of the two subsequences $\{x_1, \ldots, x_{n/2}\}$ and $\{x_{(n/2)+1}, \ldots, x_n\}$—say, $z_1, \ldots, z_{n/2}$ and $z_{(n/2)+1}, \ldots, z_n$, respectively. Then, set $s_i = z_i$, for $1 \le i \le n/2$, and $s_i = z_i + z_{n/2}$, for $(n/2) + 1 \le i \le n$.

Write a nonrecursive version of this algorithm in the WT presentation framework. What are the time and the work required by the algorithm? What is the PRAM model needed?

2.3. Suppose we are given a set of n elements stored in an array A together with an array L such that $L(i) \in \{1, 2, \ldots, k\}$ represents the label of element $A(i)$, where k is a constant. Develop an optimal $O(\log n)$ time EREW PRAM algorithm that stores all the elements of A with label 1 into the upper part of A while preserving their initial ordering, followed by the elements labeled 2 with the same initial ordering, and so on.

2.4. Let $A = (a_1, a_2, \ldots, a_n)$ be an array of elements, and let $j_1 = 1 < j_2 < \cdots < j_s = n$ be a set of indices. Consider the problem of computing the array $B = (b_1, b_2, \ldots, b_n)$ such that $b_l = a_{j_i}$, for $j_{i-1} < l \le j_i$ and $2 \le i \le s$. For example, the array B corresponding to $A = (4, 7, 9, 6, 15)$ and $j_1 = 1 < j_2 = 3 < j_3 = 5$ is given by $B = (9, 9, 9, 15, 15)$. Develop an optimal algorithm whose running time is $O(\log n)$.

2.5. **(Segmented Prefix Sums)** We are given a sequence $A = (a_1, a_2, \ldots, a_n)$ of elements from a set S with an associative operation $*$, and a Boolean array B of length n such that $b_1 = b_n = 1$. For each $i_1 < i_2$ such that $b_{i_1} = b_{i_2} = 1$ and $b_j = 0$ for all $i_1 < j < i_2$, we wish to compute the prefix sums of the subarray $(a_{i_1+1}, \ldots, a_{i_2})$ of A. Develop an $O(\log n)$ time algorithm to compute all the corresponding prefix sums. Your algorithm should use $O(n)$ operations and should run on the EREW PRAM.

2.6. Let $A = (a_1, a_2, \ldots, a_n)$ be an array of elements drawn from a linearly ordered set. The *suffix-minima problem* is to compute for each i, where $1 \le i \le n$, the minimum element among $\{a_i, a_{i+1}, \ldots, a_n\}$. We can, in a similar fashion, define the *prefix minima*. Develop an $O(\log n)$ time algorithm to compute the prefix and the suffix minima of A using a total of $O(n)$ operations. Your algorithm should run on the EREW PRAM.

2.7. Given an array $A = (a_1, \ldots, a_n)$, where the elements come from a linearly ordered set S, and given two elements $x, y \in S$, show how to store all the elements a_i from A that are between x and y in consecutive memory locations. Your algorithm should run in $O(\log n)$ time using $O(n)$ operations. Specify the PRAM model used.

2.8. Let x be an indeterminate and let n be a positive integer. Consider the problem of computing the polynomials $y_i = x^i$, for $1 \le i \le n$. Show how to compute all the y_i's in $O(\log n)$ time using $O(n)$ operations.

2.9. Given n independent jobs with processing times $\{t_1, t_2, \ldots, t_n\}$, determine a schedule of each job on a set of m machines such that the finish time is minimum. A job can be split to run on different machines. Your algorithm should run in $O(\log n)$ time using a total of $O(n)$ operations.

2.10. Consider a cycle $C = (v_1, v_2, \cdots, v_n)$ with an additional set E of edges between the vertices of C such that, for each vertex v_i, there exists at most one edge in E incident on v_i. Consider the problem of determining whether or not it is possible to draw all the edges in E inside the cycle C without any two of them crossing. Develop an $O(\log n)$ time algorithm to solve this problem. The total number of operations used must be $O(n)$. Your algorithm should run on the EREW PRAM model.

2.11. Develop an $O(\log n \log m)$ time algorithm to compute A^i, for all $1 \le i \le m$, where A is an $n \times n$ matrix. What is the amount of work required?

2.12. Let F be a forest of rooted directed trees specified by an array S of length n such that $S(i) = j$ if and only if (i, j) is an arc in F. Show that any algorithm to determine the root of the tree containing node i, for $1 \le i \le n$, will take $\Omega(\log n)$ time on the EREW PRAM even if the maximum height of any tree is very small.

2.13. Let A be a Boolean array of size n.

a. Develop an $O(1)$ time CRCW PRAM algorithm to find the smallest index k such that $A(k) = 1$. The total number of operations must be $O(n)$. *Hint*: Use a $\sqrt{n}$ divide-and-conquer strategy. Start by solving the subproblems.

b. How fast can you solve this problem on the CREW PRAM? Your algorithm must use $O(n)$ operations.

2.14. Given an array $A = (a_1, a_2, \ldots, a_n)$ whose elements are drawn from a linearly ordered set, the *left match* of a_i, where $1 \le i \le n$, is the element a_k, if it exists, such that k is the maximum index satisfying $1 \le k < i$ and $a_k < a_i$. Similarly, we can define the right match of a_i. The problem of finding the right and left matches of all the elements in A is called the problem of *all nearest smaller values* (ANSV).

Show how to solve the ANSV problem in $O(1)$ time using $O(n^2)$ operations. *Hint*: Use Exercise 2.13.

2.15. *Given a set of n open intervals (x_i, y_i) with $x_i < y_i$, where $1 \le i \le n$, consider the problem of packing these intervals on a minimum number of horizontal lines such that no two intervals on the same horizontal line overlap. Assume that the $2n$ points $\{x_i, y_i\}_{i=1}^{n}$ are sorted in nondecreasing order, with ties broken according to the index i. Develop an $O(\log n)$ time algorithm to achieve this. What is the total number of operations used? *Hint*: Adjust the sequence of the $2n$ points and, for each y_i, determine a closest $x_j > y_i$.

2.16. Show that the problem of finding the ordered list of vertices defining the convex hull of n points in the plane requires $\Omega(n \log n)$ operations. *Hint*: Consider the set of points (x_i, x_i^2), where $1 \le i \le n$.

2.17. Show how to modify Algorithm 2.6 such that it runs on the EREW PRAM model without asymptotically increasing the bounds on $T(n)$ and $W(n)$.

2.18. * An optimal $O(\log n)$ time algorithm to solve the convex-hull problem can be developed with the following divide-and-conquer strategy.

Let S be the set of points. Partition S into $S_1, S_2, \ldots, S_{\sqrt{n}}$, each of size approximately $\sqrt{n}$, by sorting the points by their x coordinates. Compute $CH(S_i)$ recursively, and then combine all the $CH(S_i)$ into a solution.

a. Let $T_{i,j}$ be the upper common tangent between $CH(S_i)$ and $CH(S_j)$. Develop an algorithm that determines which points of $CH(S_i)$, if any, belong to $CH(S)$. Your algorithm should run in $O(\log n)$ time using $O(\sqrt{n} \log n)$ operations. *Hint*: Let L_i be the tangent of smallest slope in $\{T_{i,1}, T_{i,2}, \ldots, T_{i,i-1}\}$, and let M_i be the tangent of largest slope in $\{T_{i,i+1}, T_{i,i+2}, \ldots, T_{i,\sqrt{n}}\}$.

b. Develop the recurrence equations for the running time $T(n)$ and the total number of operations $W(n)$ required by the overall algorithm. Show that $T(n) = O(\log n)$ and $W(n) = O(n \log n)$.

2.19. We have already introduced, in Exercise 2.6, the problem of computing the prefix and the suffix minima of an array A of n elements whose elements are drawn from a linearly ordered set.

a. Use Exercise 2.14 to design an $O(1)$ time algorithm for computing the prefix and the suffix minima of A, using a total of $O(n^2)$ operations.

b. *Use a $\sqrt{n}$ divide-and-conquer strategy to obtain an $O(\log \log n)$ time algorithm. The total number of operations used must be $O(n)$. Specify the PRAM model needed.

2.20. Let A be an $n \times n$ lower triangular matrix such that $n = 2^k$. Assume that A is nonsingular. Let A be partitioned into blocks, each of size $\frac{n}{2} \times \frac{n}{2}$, as follows:

$$\begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix}$$

a. Show that the inverse of A is given by the following:

$$\begin{bmatrix} A_{11}^{-1} & 0 \\ -A_{22}^{-1}A_{21}A_{11}^{-1} & A_{22}^{-1} \end{bmatrix}$$

b. Develop a divide-and-conquer algorithm to compute A^{-1} in $O(\log^2 n)$ time. What is the total number of operations used? Assume that the addition, multiplication, and division of two real numbers are primitive operations on your PRAM.

2.21. Let A and B be two sorted sequences whose elements are not necessarily distinct. Show how to use Algorithm 2.7 to merge A and B without increasing the resources required.

2.22. Develop an algorithm using the partitioning strategy outlined in Section 2.4 to merge two sequences of lengths n and m, where $n \neq m$. What is the running time of your algorithm? What is the total number of operations?

2.23. Sketch a solution to the processor allocation problem required to implement Algorithm 2.7 on the CREW PRAM model.

2.24. Let A be an arbitrary array whose n elements are drawn from a linearly ordered set S, and let $x \in S$.

a. Show how to determine $rank(x : A)$ in $O(\log n)$ time using $O(n)$ operations. Your algorithm should run on the EREW PRAM model.

b. Suppose A is sorted. How fast can you determine $rank(x : A)$ with $o(n)$ operations? Specify the PRAM model used.

2.25. Let A and B be two sorted arrays, each of length n. Suppose that we are given the integer array $rank(A : B)$. Develop an optimal $O(\log n)$ time algorithm to compute $rank(B : A)$. Your algorithm should run on the EREW PRAM model.

2.26. Let $B = (b_1, b_2, \ldots, b_k)$ be a sequence of elements stored in the global memory of an n-processor EREW PRAM.

a. Develop an algorithm to distribute a copy of B to each of the local memories of the n processors in $O(\log n)$ time, assuming that $k = O(\log n)$.

b. Generalize your algorithm to handle t such sequences that must be distributed to all the processors.

2.27. Let T be a 2-3 tree of height h. Show that the number of vertices of T is between $2^{h+1} - 1$ and $(3^{h+1} - 1)/2$, and that the number of leaves is between 2^h and 3^h.

2.28. Let T be a 2-3 tree with n leaves. You wish to search for a given set of k elements in T, where $k < n$. Show how to accomplish this search in $O(\log n)$ time on the EREW PRAM model.

2.29. Show how to delete a given sorted sequence of k elements from a 2-3 tree with n leaves in $O(\log n)$ time, using a total of $O(k \log n)$ operations. Use the CREW PRAM model if you wish.

2.30. Let A be an arbitrary array of size n, and let B be a nondecreasing array of size n whose elements are drawn from $\{1, 2, \ldots, n\}$.

a. Develop an $O(\log n)$ time EREW PRAM algorithm to compute the array C such that $c_i = a_{b_i}$, for $1 \leq i \leq n$. You must use a linear number of operations.

b. Suppose that each a_i of A is replaced with a linear block of size $\lceil \log n \rceil$. Show how to generate the corresponding $n \times \lceil \log n \rceil$ array C in $O(\log n)$ time optimally on the EREW PRAM.

2.31. Solve the processor allocation problem corresponding to the algorithm given in Section 2.5 to insert a sequence of k elements in a 2-3 tree with n leaves.

2.32. Develop an algorithm that, given a sorted array A of n elements, constructs a 2-3 tree T to represent A.

a. Specify T so as to make the correspondence between A and T one to one.

b. Show how to build such a tree optimally on the CREW PRAM model. What is the running time of your algorithm?

2.33. Let T be a 2-3 tree with n leaves holding the items $a_1 < a_2 < \ldots < a_n$. Suppose you wish to insert k elements, $b_1 < b_2 < \ldots < b_k$. One approach would be to build a new 2-3 tree with the corresponding $n + k$ leaves without using the given tree T at all. Develop such an algorithm and state its running time and the total number of operations used. Compare with the insertion algorithm described in Section 2.5.2.

2.34. Let $T_0, T_1, \cdots, T_k$ be 2-3 trees such that, for any $0 \leq i < j \leq k$, if a_i is a leaf stored in T_i and a_j is a leaf stored in T_j, then $a_i < a_j$. Develop an $O(\log n + \log k)$ time algorithm to combine $T_0, T_1, \cdots, T_k$ into a new 2-3 tree. Your algorithm should not use any simultaneous memory access.

2.35. Develop an algorithm similar to Algorithm 2.2 to compute the maximum of n elements using a doubly logarithmic-depth tree. Assume that $n = 2^{2^k}$, for some positive integer k.

2.36. Show how to compute the maximum of n elements in $O(1)$ parallel time using n^{1+c} processors, where c is an arbitrary positive constant. Which PRAM model do you need?

2.37. Suppose that we have an algorithm A to solve a given problem P of size n in $O(\log n)$ time on the PRAM using $O(n \log n)$ operations. On the other hand, an algorithm B exists that reduces the size of P by a constant fraction in $O(\log n/\log \log n)$ time using $O(n)$ operations without altering the solution. Derive an $O(\log n)$ time algorithm to solve P using $O(n)$ operations.

2.38. You are given an array of colors $A = (a_1, a_2, \ldots, a_n)$ drawn from k colors $\{c_1, c_2, \ldots, c_k\}$, where k is a constant. You wish to compute k indices $i_1, i_2, \ldots, i_k$, for each element a_i, such that i_j is the index of the closest element to the left of a_i whose color is c_j. If no such element exists, then set $i_j = 0$. Show how to solve this problem in $O(\log n)$ time using a total of $O(n)$ operations on the EREW PRAM.

2.39. You are given a sorted array $A = (a_1, a_2, \ldots, a_n)$ such that each a_i is labeled l_i, where $l_i \in \{1, 2, \ldots, m\}$, $m = O(\log n)$.

a. Develop an $O(\log n)$ time CREW PRAM algorithm to determine, for each $1 \leq i \leq m$, the minimum element of A with label l_i. Your algorithm must use $O(n)$ operations. Do not use any sorting algorithms.

b. What if $m = O(\log \log n)$? Which PRAM model do you need?

2.40. Develop a routine to get pointers to the first vertex and to the last vertex of the same color after step 3 of Algorithm 2.9. Your procedure should run in $O(\log n)$ time using a total of $O(n)$ operations on the EREW PRAM model.

2.41. *Let $T = (V, E)$ be a rooted tree with $V = \{1, 2, \ldots, n\}$, specified by the pairs $(i, p(i))$, $1 \leq i \leq n$, where $p(i)$ is the parent of i. If r is the root, then $p(r) = 0$.

a. Develop a CREW PRAM algorithm to color T with three colors in $O(\log^* n)$ using $O(n \log^* n)$ operations. Can you derive an optimal algorithm?

b. A *pseudoforest* is a directed graph in which each vertex has an outdegree ≤ 1. Generalize your tree-coloring algorithm from part (a) to handle pseudo forests.

2.42. Given a pseudoforest (as defined in Exercise 2.41), you wish to assign to each vertex v a label $l(v)$ such that all the vertices in the same weakly connected component (that is, a connected component in the undirected version of the graph) get the same label. Develop an $O(\log n)$ time algorithm to accomplish this task. What is the total number of operations required by your algorithm?

2.43. Let $T = (V, E)$ be a rooted tree specified by the pairs $(i, p(i))$, $1 \leq i \leq n$, where $p(i)$ is the parent of i. Show that T can always be 2-colored. Develop an $O(\log n)$ time CREW PRAM algorithm to find such a coloring.

2.44. *Given a set of n integers, each colored with one color from $\{1, 2, \ldots, k\}$, where $k \leq \log n$, show how to obtain the prefix sums of the elements of a given color i, in the order they appear, for all i. Your algorithm should run in $O(\log n)$ time using a total of $O(n)$ operations on the EREW PRAM.

2.45. *Suppose we are given n integers in the range $[0, \log n - 1]$. Develop an $O(\log n)$ time EREW PRAM sorting algorithm that uses $O(n)$ operations. Can you generalize your algorithm to the case when the range is

$[0, m], m \geq \log n$? What is the corresponding time? *Hint*: Use radix sort, and start by computing the number of elements equal to i, for each i. Use your answer to Exercise 2.44.

2.46. *Given an undirected graph $G = (V, E)$ such that $|V| = n$ and the maximum degree of any vertex v is bounded by a constant δ, show how to color G with $\delta + 1$ colors in $O(\log^* n)$ time using $O(n \log^* n)$ operations. *Hint*: Decompose the graph into $O(\delta)$ pseudoforests. Color each pseudoforest with three colors as in Exercise 2.41, and recolor G appropriately.

Bibliographic Notes

The parallel algorithm for computing the prefix sums was developed by Ladner and Fischer [13]. The design of efficient parallel algorithms based on balanced binary trees was emphasized in [6], where this technique was applied to several scheduling problems. The pointer jumping technique was used in several of the early parallel graph algorithms; for example, see [10, 16, 11]. Wyllie [21] used this technique on linked lists to solve the parallel prefix problem. The divide-and-conquer approach for computing the planar convex hull was originally proposed by Shamos [17] for its sequential computation. Valiant [19] gave the first (nonoptimal) $O(\log \log n)$ time algorithm for computing the maximum of n elements, and introduced the partitioning strategy to solve the merging problem on the parallel comparison-tree model (to be introduced in Chapter 4). The optimal $O(\log \log n)$ time algorithm to compute the maximum, and an optimal $O(\log n)$ time merging algorithm, were given in [18]. The accelerated-cascading technique was introduced in [4]. Pipelining is a classical technique that can be traced back to the early days of computing. Our algorithm to insert a sequence of elements into a 2-3 tree is taken from [14]. The technique to break the symmetry in coloring the vertices of a cycle was introduced by Cole and Vishkin in [5], and was later simplified in [8]. A similar technique was developed independently by Wagner and Han [20]. The solutions to Exercises 2.13 and 2.15 can be found in [7] and [12], respectively. Exercises 2.14 and 2.19 are taken from [3]. The divide-and-conquer strategy sketched in Exercise 2.20 was given in [9]. The $\sqrt{n}$ divide-and-conquer strategy to solve the convex-hull problem described in Exercise 2.18 was developed independently in [1] and [2]. Exercises 2.28 and 2.29 are taken from [14]. Solutions to Exercises 2.41 and 2.46 can be found in [8], and the solutions to Exercises 2.44 and 2.45 appear in [5, 15].

References

1. Aggarwal, A., B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293-327, 1988.
2. Atallah, M. J., and M. T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492-507, 1986.
3. Berkman, O., B. Schieber, and U. Vishkin. Some doubly logarithmic optimal parallel algorithms based on finding all nearest smaller values. Technical Report UMIACS-TR-88-79, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1988.

4. Cole, R., and U. Vishkin. Approximate coin tossing with applications to list, tree and graph problems. In *Proceedings Twenty-Seventh Annual IEEE Symposium on Foundations of Computer Science*, Toronto, Canada, 1986, pages 478-491. IEEE Press, Piscataway, NJ.
5. Cole, R., and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32-53, 1986.
6. Dekel, E., and S. Sahni. Parallel scheduling algorithms. *Operations Research*, 31(1):24-49, 1983.
7. Fich, F. E., P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Computing*, 17(3):606-627, 1988.
8. Goldberg, A., S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings Nineteenth Annual ACM Symposium on Theory of Computing*, New York, NY, 1987, pages 315-324.
9. Heller, D. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20(4):740-777, 1978.
10. Hirschberg, D. S. Parallel algorithms for the transitive closure and the connected components problems. In *Proceedings Eighth Annual ACM Symposium on Theory of Computing*, Hershey, PA, 1976, pages 55-57. ACM Press, New York.
11. JáJá, J. Graph connectivity problems on parallel computers. Technical Report CS-78-05, Department of Computer Science, Pennsylvania State University, University Park, PA, 1978.
12. JáJá, J., and S-C. Chang. Parallel algorithms for channel routing in the knock-knee model. *SIAM J. Computing*, 20 (2):228-245, 1991.
13. Ladner, R. E., and M. J. Fischer. Parallel prefix computation. *JACM*, 27(4):831-838, 1980.
14. Paul, W., U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceedings Tenth ICALP*, Barcelona, Spain, 1983, Volume 154, pages 597-609. Springer-Verlag.
15. Rajasekaran, S., and J. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Computing*, 18(3):594-607, 1989.
16. Savage, C. *Parallel Algorithms for Graph Theoretic Problems*. PhD thesis, Computer Science Department, University of Illinois, Urbana, IL, 1978.
17. Shamos, M. I. *Computational Geometry*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1978.
18. Shiloach, Y., and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88-102, 1981.
19. Valiant, L. G. Parallelism in comparison problems. *SIAM J. Computing*, 4(3):348-355, 1975.
20. Wagner, W., and Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem. In *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, 1986, pages 924-930.
21. Wyllie, J. C. *The Complexity of Parallel Computations*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, 1979.