Chapter 13

PRAM ALGORITHMS

13.1 INTRODUCTION

So far our discussion of algorithms has been confined to single-processor computers. In this chapter we study algorithms for parallel machines (i.e., computers with more than one processor). There are many applications in day-to-day life that demand real-time solutions to problems. For example, weather forecasting has to be done in a timely fashion. In the case of severe hurricanes or snowstorms, evacuation has to be done in a short period of time. If an expert system is used to aid a physician in surgical procedures, decisions have to be made within seconds. And so on. Programs written for such applications have to perform an enormous amount of computation. In the forecasting example, large-sized matrices have to be operated on. In the medical example, thousands of rules have to be tried. Even the fastest single-processor machines may not be able to come up with solutions within tolerable time limits. Parallel machines offer the potential of decreasing the solution times enormously.

Example 13.1 Assume that you have 5 loads of clothes to wash. Also assume that it takes 25 minutes to wash one load in a washing machine. Then, it will take 125 minutes to wash all the clothes using a single machine. On the other hand, if you had 5 machines, washing could be completed in just 25 minutes! In this example, if there are p washing machines and p loads of clothes, then the washing time can be cut down by a factor of p compared to having a single machine. Here we have assumed that every machine takes exactly the same time to wash. If this assumption is invalid, then the washing time will be dictated by the slowest machine. □

Example 13.2 As another example, say there are 100 numbers to be added and there are two persons A and B. Person A can add the first 50 numbers. At the same time B can add the next 50 numbers. When they are done, one of them can add the two individual sums to get the final answer. So, two people can add the 100 numbers in almost half the time required by one. □

The idea of parallel computing is very similar. Given a problem to solve, we partition the problem into many subproblems; let each processor work on a subproblem; and when all the processors are done, the partial solutions are combined to arrive at the final answer. If there are p processors, then potentially we can cut down the solution time by a factor of p. We refer to any algorithm designed for a single-processor machine as a *sequential algorithm* and any designed for a multiprocessor machine as a *parallel algorithm.*

Definition 13.1 Let π be a given problem for which the best-known sequential algorithm has a run time of $S'(n)$, where n is the problem size. If a parallel algorithm on a p-processor machine runs in time $T'(n,p)$, then the *speedup* of the parallel algorithm is defined to be $\frac{S'(n)}{T'(n,p)}$.

If the best-known sequential algorithm for π has an asymptotic run time of $S(n)$ and if $T(n,p)$ is the asymptotic run time of a parallel algorithm, then the *asymptotic speedup* of the parallel algorithm is defined to be $\frac{S(n)}{T(n,p)}$. If $\frac{S(n)}{T(n,p)} = \Theta(p)$, then the algorithm is said to have *linear speedup.* □

Note: In this book we use the terms "speedup" and "asymptotic speedup" interchangeably. Which one is meant is clear from the context.

Example 13.3 For the problem of Example 13.2, the 100 numbers can be added sequentially in 99 units of time. Person A can add 50 numbers in 49 units of time. At the same time, B can add the other 50 numbers. In another unit of time, the two partial sums can be added; this means the parallel run time is 50. So the speedup of this parallel algorithm is $\frac{99}{50} = 1.98$, which is very nearly equal to 2! □

Example 13.4 There are many sequential sorting algorithms such as heap sort (Section 2.4.2) that are optimal and run in time $\Theta(n \log n)$, n being the number of keys to be sorted. Let $\mathcal{A}$ be an n-processor parallel algorithm that sorts n keys in $\Theta(\log n)$ time and let $\mathcal{B}$ be an n^2-processor algorithm that also sorts n keys in $\Theta(\log n)$ time.

Then, the speedup of $\mathcal{A}$ is $\frac{\Theta(n \log n)}{\Theta(\log n)} = \Theta(n)$. On the other hand, the speedup of $\mathcal{B}$ is also $\frac{\Theta(n \log n)}{\Theta(\log n)} = \Theta(n)$. Algorithm $\mathcal{A}$ has linear speedup, whereas $\mathcal{B}$ does not have a linear speedup. □

Definition 13.2 If a p-processor parallel algorithm for a given problem runs in time $T(n,p)$, the *total work done* by this algorithm is defined to be $pT(n,p)$. The *efficiency* of the algorithm is defined to be $\frac{S(n)}{pT(n,p)}$, where $S(n)$ is the asymptotic run time of the best known sequential algorithm for solving the same problem. Also, the parallel algorithm is said to be *work-optimal* if $pT(n,p) = O(S(n))$. $\Box$

Note: A parallel algorithm is work-optimal if and only if it has linear speedup. Also, the efficiency of a work-optimal parallel algorithm is $\Theta(1)$.

Example 13.5 Let w be the time needed to wash one load of clothes on a single machine in Example 13.1. Also let n be the total number of loads to wash. A single machine will take time nw. If there are p machines, the washing time is $\lceil \frac{n}{p} \rceil w$. Thus the speedup is $n/\lceil \frac{n}{p} \rceil$. This speedup is $> \frac{p}{2}$ if $n \geq p$. So, the asymptotic speedup is $\Omega(p)$ and hence the parallel algorithm has linear speedup and is work-optimal. Also, the efficiency is $\frac{nw}{p\lceil \frac{n}{p} \rceil w}$. This is $\Theta(1)$ if $n \geq p$. $\Box$

Example 13.6 For the algorithm $\mathcal{A}$ of Example 13.4, the total work done is $n\Theta(\log n) = \Theta(n \log n)$. Its efficiency is $\frac{\Theta(n \log n)}{\Theta(n \log n)} = \Theta(1)$. Thus, $\mathcal{A}$ is work-optimal and has a linear speedup. The total work done by algorithm $\mathcal{B}$ is $n^2\Theta(\log n) = \Theta(n^2 \log n)$ and its efficiency is $\frac{\Theta(n \log n)}{\Theta(n^2 \log n)} = \Theta(\frac{1}{n})$. As a result, $\mathcal{B}$ is not work-optimal! $\Box$

Is it possible to get a speedup of more than p for any problem on a p-processor machine? Assume that it is possible (such a speedup is called a *superlinear speedup*). In particular, let π be the problem under consideration and S be the best-known sequential run time. If there is a parallel algorithm on a p-processor machine whose speedup is better than p, it means that the parallel run time T satisfies $T < \frac{S}{p}$; that is, $pT < S$. Note that a single step of the parallel algorithm can be simulated on a single processor in time $\leq p$. Thus the whole parallel algorithm can be simulated sequentially in time $\leq pT < S$. This is a contradiction since by assumption S is the run time of the best-known sequential algorithm for solving π!

The preceding discussion is valid only when we consider asymptotic speedups. When the speedup is defined with respect to the actual run times on the sequential and parallel machines, it is possible to obtain superlinear speedup. Two of the possible reasons for such an anomaly are (1) p processors have more aggregate memory than one and (2) the cache-hit frequency may be better for the parallel machine as the p-processors may have more aggregate cache than does one processor.

One way of solving a given problem in parallel is to explore many techniques (i.e., algorithms) and identify the one that is the most parallelizable. To achieve a good speedup, it is necessary to parallelize every component of the underlying technique. If a fraction f of the technique cannot be parallelized (i.e., has to be run serially), then the maximum speedup that can be obtained is limited by f. Amdahl's law (proof of which is left as an exercise) relates the maximum speedup achievable with f and p (the number of processors used) as follows.

Lemma 13.1 Maximum speedup = $\frac{1}{f+\frac{1-f}{p}}$. $\square$

Example 13.7 Consider some technique for solving a problem π. Assume that $p = 10$. If $f = 0.5$ for this technique, then the maximum speedup that can be obtained is $\frac{1}{0.5+\frac{1-0.5}{10}} = \frac{20}{11}$, which is less than 2! If $f = 0.1$, then the maximum speedup is $\frac{10}{1.9}$, which is slightly more than 5! Finally, if $f = 0.01$, then the maximum speedup is $\frac{10}{1.09}$, which is slightly more than 9! $\square$

EXERCISES

1. Algorithms $\mathcal{A}$ and $\mathcal{B}$ are parallel algorithms for solving the selection problem (Section 3.6). Algorithm $\mathcal{A}$ uses $n^{0.5}$ processors and runs in time $\Theta(n^{0.5})$. Algorithm $\mathcal{B}$ uses n processors and runs in $\Theta(\log n)$ time. Compute the works done, speedups, and efficiencies of these two algorithms. Are these algorithms work-optimal?

2. Mr. Ultrasmart claims to have found an algorithm for selection that runs in time $\Theta(\log n)$ using $n^{3/4}$ processors. Is this possible?

3. Prove Amdahl's law.

13.2 COMPUTATIONAL MODEL

The sequential computational model we have employed so far is the RAM (random access machine). In the RAM model we assume that any of the following operations can be performed in one unit of time: addition, subtraction, multiplication, division, comparison, memory access, assignment, and so on. This model has been widely accepted as a valid sequential model. On the other hand when it comes to parallel computing, numerous models have been proposed and algorithms have been designed for each such model.

An important feature of parallel computing that is absent in sequential computing is the need for interprocessor communication. For example, given any problem, the processors have to communicate among themselves and agree on the subproblems each will work on. Also, they need to communicate to see whether every one has finished its task, and so on. Each machine

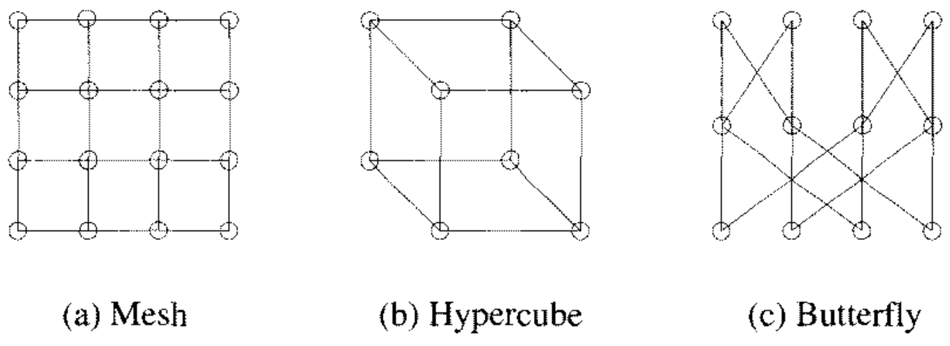


Figure 13.1 Examples of fixed connection machines

or processor in a parallel computer can be assumed to be a RAM. Various parallel models differ in the way they support interprocessor communication. Parallel models can be broadly categorized into two: fixed connection machines and shared memory machines.

A fixed connection network is a graph $G(V, E)$ whose nodes represent processors and whose edges represent communication links between processors. Usually we assume that the degree of each node is either a constant or a slowly increasing function of the number of nodes in the graph. Examples include the mesh, hypercube, butterfly, and so on (see Figure 13.1). Interprocessor communication is done through the communication links. Any two processors connected by an edge in G can communicate in one step. In general two processors can communicate through any of the paths connecting them. The communication time depends on the lengths of these paths (at least for small packets). More details of these models are provided in Chapters 14 and 15.

In shared memory models [also called PRAMs (Parallel Random Access Machines)], a number (say p) of processors work synchronously. They communicate with each other using a common block of global memory that is accessible by all. This global memory is also called *common* or *shared* memory (see Figure 13.2). Communication is performed by writing to and/or reading from the common memory. Any two processors i and j can communicate in two steps. In the first step, processor i writes its message into memory cell j, and in the second step, processor j reads from this cell. In contrast, in a fixed connection machine, the communication time depends on the lengths of the paths connecting the communicating processors.

Each processor in a PRAM is a RAM with some local memory. A single step of a PRAM algorithm can be one of the following: arithmetic operation (such as addition, division, and so on.), comparison, memory access (local or global), assignment, etc. The number (m) of cells in the global memory is typically assumed to be the same as p. But this need not always be the case.

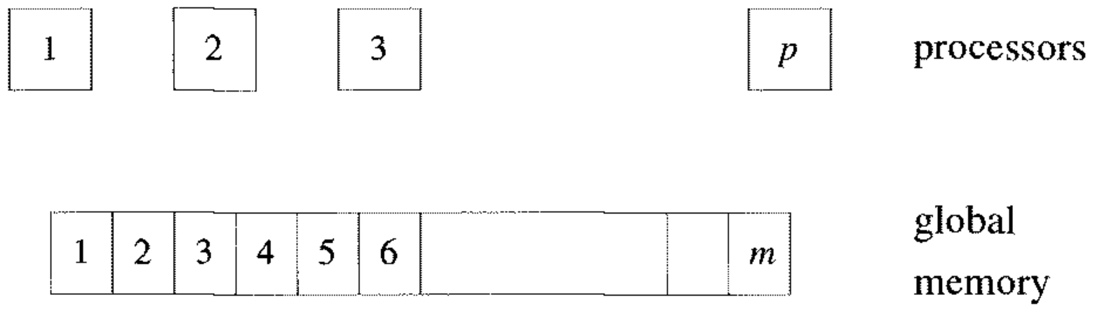


Figure 13.2 A parallel random access machine

In fact we present algorithms for which m is much larger or smaller than p. We also assume that the input is given in the global memory and there is space for the output and for storing intermediate results. Since the global memory is accessible by all processors, access conflicts may arise. What happens if more than one processor tries to access the same global memory cell (for the purpose of reading from or writing into)? There are several ways of resolving read and write conflicts. Accordingly, several variants of the PRAM arise.

EREW (Exclusive Read and Exclusive Write) PRAM is the shared memory model in which no concurrent read or write is allowed on any cell of the global memory. Note that ER or EW does not preclude different processors simultaneously accessing different memory cells. For example, at a given time step, processor one might access cell five and at the same time processor two might access cell 12, and so on. But processors one and two cannot access memory cell ten, for example, at the same time. CREW (Concurrent Read and Exclusive Write) PRAM is a variation that permits concurrent reads but not concurrent writes. Similarly one could also define the ERCW model. Finally, the CRCW PRAM model allows both concurrent reads and concurrent writes.

In a CREW or CRCW PRAM, if more than one processor tries to read from the same cell, clearly, they will read the same information. But in a CRCW PRAM, if more than one processor tries to write in the same cell, then possibly they may have different messages to write. Thus there has to be an additional mechanism to determine which message gets to be written. Accordingly, several variants of the CRCW PRAM can be derived. In a *common* CRCW PRAM, concurrent writes are permitted in any cell only if all the processors conflicting for this cell have the same message to write. In an *arbitrary* CRCW PRAM, if there is a conflict for writing, one of the processors will succeed in writing and we don't know which one. Any algorithms designed for this model should work no matter which processors succeed in the event of conflicts. The *priority* CRCW PRAM lets the processor with the highest priority succeed in the case of conflicts. Typically each processor is assigned a (static) priority to begin with.

Example 13.8 Consider a 4-processor machine and also consider an operation in which each processor has to read from the global cell $M[1]$. This operation can be denoted as

```
Processor $i$ (in parallel for $1 \leq i \leq 4$) does:
    Read $M[1]$;
```

This concurrent read operation can be performed in one unit of time on the CRCW as well as on the CREW PRAMs. But on the EREW PRAM, concurrent reads are prohibited. Still, we can perform this operation on the EREW PRAM making sure that at any given time no two processors attempt to read from the same memory cell. One way of performing this is as follows: processor 1 reads $M[1]$ at the first time unit; processor 2 reads $M[1]$ at the second time unit; and processors 3 and 4 read $M[1]$ at the third and fourth time units, respectively. The total run time is four. Better algorithms for general cases are considered in later sections (see Section 13.6, Exercise 11).

Now consider the operation in which each processor has to access $M[1]$ for writing at the same time. Since only one message can be written to $M[1]$, one has to assume some scheme for resolving contentions. This operation can be denoted as

```
Processor $i$ (in parallel for $1 \leq i \leq 4$) does:
    Write $M[1]$;
```

Again, on the CRCW PRAM, this operation can be completed in one unit of time. On the CREW and EREW PRAMs, concurrent writes are prohibited. However, these models can simulate the effects of a concurrent write. Consider our simple example of four processors trying to write in $M[1]$. Simulating a common CRCW PRAM requires the four processors to verify that all wish to write the same value. Following this, processor 1 can do the writing. Simulating a priority CRCW PRAM requires the four processors to first determine which has the highest priority, and then the one with this priority does the write. Other models may be similarly simulated. Exercise 12 of Section 13.6 deals with more general concurrent writes. □

Note that any algorithm that runs on a p-processor EREW PRAM in time $T(n,p)$, where n is the problem size, can also run on a p-processor CREW PRAM or a CRCW PRAM within the same time. But a CRCW PRAM algorithm or a CREW PRAM algorithm may not be implementable on an

```
Processor $i$ (in parallel for $1 \le i \le n$) does:
    if $(A[i] = 1)$ then $A[0] := A[i]$;
```

Algorithm 13.1 Computing the boolean OR in $O(1)$ time

EREW PRAM preserving the asymptotic run time. In Example 13.8, we saw that the implementation of a single concurrent write or concurrent read step takes much more time on the EREW PRAM. Likewise, a p-processor CRCW PRAM algorithm may not be implementable on a p-processor CREW PRAM preserving the asymptotic run time. It turns out that there is a strict hierarchy among the variants of the PRAM in terms of their computational power. For example, a CREW PRAM is strictly more powerful than an EREW PRAM. This means that there is at least one problem that can be solved in asymptotically less time on a CREW PRAM than on an EREW PRAM, given the same number of processors. Also, any version of the CRCW PRAM is more powerful than a CREW PRAM as is demonstrated by Example 13.9.

Example 13.9 $A[0] = A[1] || A[2] || \cdots || A[n]$ is the Boolean (or logical) OR of the n bits $A[1:n]$. $A[0]$ is easily computed in $O(n)$ time on a RAM. Algorithm 13.1 shows how $A[0]$ can be computed in $\Theta(1)$ time using an n-processor CRCW PRAM.

Assume that $A[0]$ is zero to begin with. In the first time step, processor i, for $1 \le i \le n$, reads memory location $A[i]$ and proceeds to write a 1 in memory location $A[0]$ if $A[i]$ is a 1. Since several of the $A[i]$'s may be 1, several processors may write to $A[0]$ concurrently. Hence the algorithm cannot be run (as such) on an EREW or CREW PRAM. In fact, for these two models, it is known that the parallel complexity of the Boolean OR problem is $\Omega(\log n)$ no matter how many processors are used. Note that the algorithm of Algorithm 13.1 works on all three varieties of the CRCW PRAM. □

Theorem 13.1 The boolean OR of n bits can be computed in $O(1)$ time on an n-processor common CRCW PRAM. □

There exists a hierarchy among the different versions of the CRCW PRAM also. Common, arbitrary, and priority form an increasing hierarchy of computing power. Let EREW$(p, T(n,p))$ denote the set of all problems that can be solved using a p-processor EREW PRAM in time $T(n,p)$ (n being the problem size). Similarly define CREW$(p, T(n,p))$ and CRCW$(p, T(n,p))$. Then,

$$\text{EREW}(p, T(n,p)) \subset \text{CREW}(p, T(n,p)) \subset \text{Common CRCW}(p, T(n,p))$$

$$\subset \text{Arbitrary CRCW}(p, T(n,p)) \subset \text{Priority CRCW}(p, T(n,p))$$

All the algorithms developed in this chapter for the PRAM model assume some relationship between the problem size n and the number of processors p. For example, the CRCW PRAM algorithm of Algorithm 13.1 solves a problem of size n using n processors. In practice, however, a problem of size n is solved on a computer with a constant number p of processors. All the algorithms designed under some assumptions about the relationship between n and p can also be used when fewer processors are available as there is a general slow-down lemma for the PRAM model.

Let $\mathcal{A}$ be a parallel algorithm for solving problem π that runs in time T using p processors. The slow-down lemma concerns the simulation of the same algorithm on a p'-processor machine (for $p' < p$).

Each step of algorithm $\mathcal{A}$ can be simulated on the p'-processor machine (call it $\mathcal{M}$) in time $\leq \lceil \frac{p}{p'} \rceil$, since a processor of $\mathcal{M}$ can be in charge of simulating $\lceil \frac{p}{p'} \rceil$ processors of the original machine. Thus, the simulation time on $\mathcal{M}$ is $\leq T \lceil \frac{p}{p'} \rceil$. Therefore, the total work done on $\mathcal{M}$ is $\leq p'T \lceil \frac{p}{p'} \rceil$ $\leq pT + p'T = O(pT)$. This results in the following lemma.

Lemma 13.2 [Slow-down lemma] Any parallel algorithm that runs on a p-processor machine in time T can be run on a p'-processor machine in time $O\left(\frac{pT}{p'}\right)$, for any $p' < p$. □

Since no such slow-down lemma is known for the models of Chapters 14 and 15, we need to develop different algorithms when the number of processors changes relative to the problem size. So, in Chapters 14 and 15 we develop algorithms under different assumptions about the relationship between n and p.

Example 13.10 Algorithm 13.1 runs in $\Theta(1)$ time using n processors. Using the slow-down lemma, the same algorithm also runs in $\Theta(\log n)$ time using $\frac{n}{\log n}$ processors; it also runs in $\Theta(\sqrt{n})$ time using $\sqrt{n}$ processors; and so on. When $p = 1$, the algorithm runs in time $\Theta(n)$, which is the same as the run time of the best sequential algorithm! □

Note: In Chapters one through nine we presented various algorithm design techniques and demonstrated how they can be applied to solve several specific problems. In the domain of parallel algorithms also some common ideas have been repeatedly employed to design algorithms over a wide variety of models. In Chapters 13, 14, and 15 we consider the PRAM, mesh, and hypercube models, respectively. In particular, we study the following problems: prefix computation, list ranking, selection, merging, sorting, some basic graph problems, and convex hull. For each of these problems a common theme is used to solve it on the three different models. In Chapter 13 we present full details of these common themes. In Chapters 14 and 15 we only point out the differences in implementation.

EXERCISES

1. Present an $O(1)$ time n-processor common CRCW PRAM algorithm for computing the boolean AND of n bits.

2. Input is an array of n elements. Give an $O(1)$ time, n-processor common CRCW PRAM algorithm to check whether the array is in sorted order.

3. Solve the boolean OR and AND problems on the CREW and EREW PRAMs. What are the time and processor bounds of your algorithms?

4. The array A is an array of n keys, where each key is an integer in the range $[1, n]$. The problem is to decide whether there are any repeated elements in A. Show how you do this in $O(1)$ time on an n-processor CRCW PRAM. Which version of the CRCW PRAM are you using?

5. Can Exercise 4 be solved in $O(1)$ time using n processors on any of the PRAMs if the keys are arbitrary? How about if there are n^2 processors?

6. The string matching problem takes as input a text t and a pattern p, where t and p are strings from an alphabet Σ. The problem is to determine all the occurrences of p in t. Present an $O(1)$ time PRAM algorithm for string matching. Which PRAM are you using and what is the processor bound of your algorithm?

7. The algorithm $\mathcal{A}$ is a parallel algorithm that has two components. The first component runs in $\Theta(\log\log n)$ time using $\frac{n}{\log\log n}$ EREW PRAM processors. The second component runs in $\Theta(\log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors. Show that the whole algorithm can be run in $\Theta(\log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors.

13.3 FUNDAMENTAL TECHNIQUES AND ALGORITHMS

In this section we introduce two basic problems that arise in the parallel solution of numerous problems. The first problem is known as the *prefix computation problem* and the second one is called the *list ranking problem.*

13.3.1 Prefix Computation

Let Σ be any domain in which the binary *associative* operator $\oplus$ is defined. An operator $\oplus$ is said to be associative if for any three elements x, y, and z from Σ, $((x \oplus y) \oplus z) = (x \oplus (y \oplus z))$; that is, the order in which the operation $\oplus$ is performed does not matter. It is also assumed that $\oplus$ is unit time computable and that Σ is closed under this operation; that is, for any $x, y \in \Sigma$, $x \oplus y \in \Sigma$. The prefix computation problem on Σ has as input n elements from Σ, say, $x_1, x_2, \ldots, x_n$. The problem is to compute the n elements $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$. The output elements are often referred to as the *prefixes.*

Example 13.11 Let Σ be the set of integers and $\oplus$ be the usual addition operation. If the input to the prefix computation problem is $3, -5, 8, 2, 5, 4$, the output is $3, -2, 6, 8, 13, 17$. As another example, let Σ be the set of integers and $\oplus$ be the multiplication operation. If $2, 3, 1, -2, -4$ is the input, the output is $2, 6, 6, -12, 48$. □

Example 13.12 Let Σ be the set of all integers and $\oplus$ be the minimum operator. Note that the minimum operator is associative. If the input to the prefix computation problem is $5, 8, -2, 7, -11, 12$, the output is $5, 5, -2, -2, -11, -11$. In particular, the last element output is the minimum among all the input elements. □

The prefix computation problem can be solved in $O(n)$ time sequentially. Any sequential algorithm for this problem needs $\Omega(n)$ time. Fortunately, work-optimal algorithms are known for the prefix computation problem on many models of parallel computing. We present a CREW PRAM algorithm that uses $\frac{n}{\log n}$ processors and runs in $O(\log n)$ time. Note that the work done by such an algorithm is $O(n)$ and hence the algorithm has an efficiency of $\Theta(1)$ and is work-optimal. Also, the speedup of this algorithm is $\Theta(n/\log n)$.

We employ the divide-and-conquer strategy to devise the prefix algorithm. Let the input be $x_1, x_2, \ldots, x_n$. Without loss of generality assume that n is an integral power of 2. We first present an n-processor and $O(\log n)$ time algorithm (Algorithm 13.2).

Step 0. If $n = 1$, one processor outputs x_1.

Step 1. Let the first $n/2$ processors recursively compute the prefixes of $x_1, x_2, \ldots, x_{n/2}$ and let $y_1, y_2, \ldots, y_{n/2}$ be the result. At the same time let the rest of the processors recursively compute the prefixes of $x_{n/2+1}, x_{n/2+2}, \ldots, x_n$ and let $y_{n/2+1}, y_{n/2+2}, \ldots, y_n$ be the output.

Step 2. Note that the first half of the final answer is the same as $y_1, y_2, \ldots, y_{n/2}$. The second half of the final answer is $y_{n/2} \oplus y_{n/2+1}, y_{n/2} \oplus y_{n/2+2}, \ldots, y_{n/2} \oplus y_n$.
Let the second half of the processors read $y_{n/2}$ concurrently from the global memory and update their answers. This step takes $O(1)$ time.

Algorithm 13.2 Prefix computation in $O(\log n)$ time

Example 13.13 Let $n = 8$ and $p = 8$. Let the input to the prefix computation problem be $12, 3, 6, 8, 11, 4, 5, 7$ and let $\oplus$ be addition. In step 1, processors 1 to 4 compute the prefix sums of $12, 3, 6, 8$ to arrive at $12, 15, 21, 29$. At the same time processors 5 to 8 compute the prefix sums of $11, 4, 5, 7$ to obtain $11, 15, 20, 27$. In step 2, processors 1 to 4 don't do anything. Processors 5 to 8 update their results by adding 29 to every prefix sum and get $40, 44, 49, 56$. □

What is the time complexity of Algorithm 13.2? Let $T(n)$ be the run time of Algorithm 13.2 on any input of size n using n processors. Step 1 takes $T(\frac{n}{2})$ time and step 2 takes $O(1)$ time. So, we get the following recurrence relation for $T(n)$:

$$T(n) \;=\; T\left(\frac{n}{2}\right) + O(1), \quad T(1) = 1$$

This solves to $T(n) = O(\log n)$. Note that in defining the run time of a parallel divide-and-conquer algorithm, it is essential to quantify it with the number of processors used.

Algorithm 13.2 is not work-optimal for the prefix computation problem since the total work done by this algorithm is $\Theta(n \log n)$, whereas the run time of the best-known sequential algorithm is $\Theta(n)$. A work-optimal algorithm can be obtained by decreasing the number of processors used to $\frac{n}{\log n}$,

---

Step 1. Processor i ($i = 1, 2, \ldots, \frac{n}{\log n}$) in parallel computes the prefixes of its $\log n$ assigned elements $x_{(i-1)\log n+1}, x_{(i-1)\log n+2}, \ldots, x_{i \log n}$. This takes $O(\log n)$ time. Let the results be $z_{(i-1)\log n+1}, z_{(i-1)\log n+2}, \ldots, z_{i \log n}$.

Step 2. A total of $\frac{n}{\log n}$ processors collectively employ Algorithm 13.2 to compute the prefixes of the $\frac{n}{\log n}$ elements $z_{\log n}, z_{2\log n}, z_{3\log n}, \ldots, z_n$. Let $w_{\log n}, w_{2\log n}, w_{3\log n}, \ldots, w_n$ be the result.

Step 3. Each processor updates the prefixes it computed in step 1 as follows. Processor i computes and outputs $w_{(i-1)\log n} \oplus z_{(i-1)\log n+1}, w_{(i-1)\log n} \oplus z_{(i-1)\log n+2}, \ldots, w_{(i-1)\log n} \oplus z_{i\log n}$, for $i = 2, 3, \ldots, \frac{n}{\log n}$. Processor 1 outputs $z_1, z_2, \ldots, z_{\log n}$ without any modifications.

Algorithm 13.3 Work-optimal logarithmic time prefix computation

while keeping the asymptotic run time the same. The number of processors used can be decreased to $\frac{n}{\log n}$ as follows. We first reduce the number of inputs to $\frac{n}{\log n}$, apply the non-work-optimal Algorithm 13.2 to compute the prefixes of the reduced input, and then finally compute all the n prefixes. Every processor will be in charge of computing $\log n$ final answers. If the input is $x_1, x_2, \ldots, x_n$ and the output is $y_1, y_2, \ldots, y_n$, let processor i be in charge of the outputs $y_{(i-1)\log n+1}, y_{(i-1)\log n+2}, \ldots, y_{i\log n}$, for $i = 1, 2, \ldots, \frac{n}{\log n}$. The detailed algorithm appears as Algorithm 13.3. The correctness of the algorithm is clear. Step 1 takes $O(\log n)$ time. Step 2 takes $O(\log(\frac{n}{\log n})) = O(\log n)$ time (using Algorithm 13.2). Finally, step 3 also takes $O(\log n)$ time. Thus we get the following theorem.

Theorem 13.2 Prefix computation on an n-element input can be performed in $O(\log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors. □

Example 13.14 Let the input to the prefix computation be $5, 12, 8, 6, 3, 9, 11, 12, 1, 5, 6, 7, 10, 4, 3, 5$ and let $\oplus$ stand for addition. Here $n = 16$ and $\log n = 4$. Thus in step 1, each of the four processors computes prefix sums on four numbers each. In step 2, prefix sums on the local sums is computed, and in step 3, the locally computed results are updated. Figure 13.3 illustrates these three steps. □

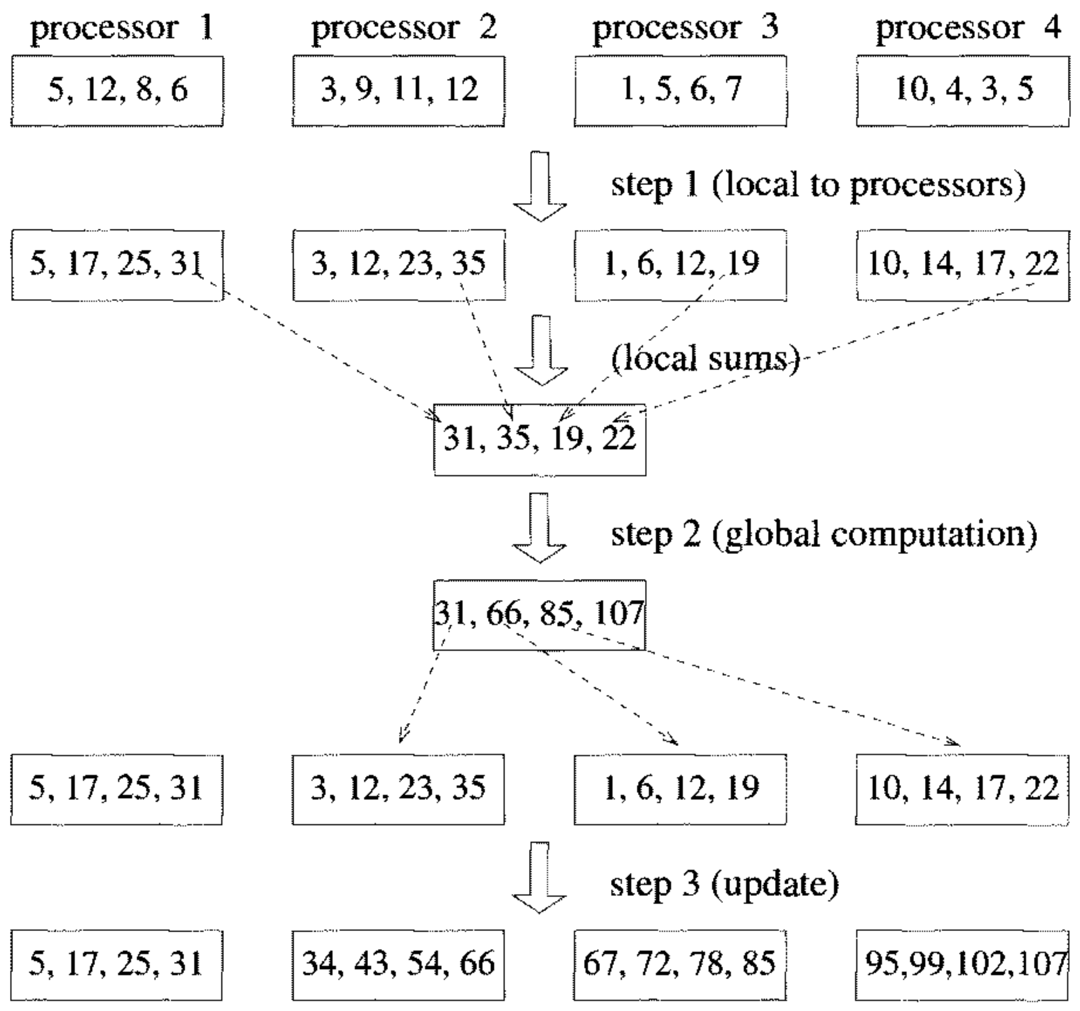


Figure 13.3 Prefix computation – an example

13.3.2 List Ranking

List ranking plays a vital role in the parallel solution of several graph problems. The input to the problem is a list given in the form of an array of nodes. A node consists of some data and a pointer to its right neighbor in the list. The nodes themselves need not occur in any order in the input. The problem is to compute for each node in the list the number of nodes to its right (also called the *rank of the node*). Since the data contained in

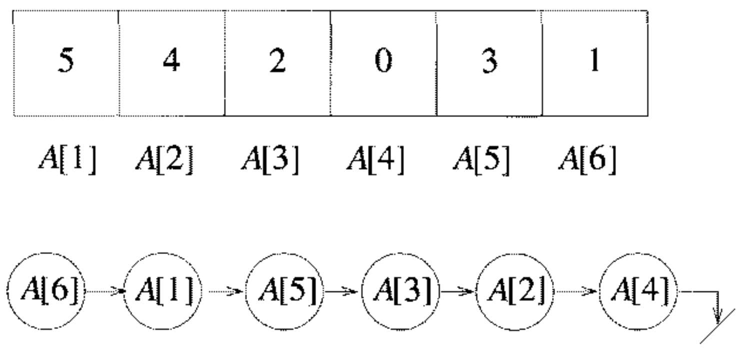


Figure 13.4 Input to the list ranking problem and the corresponding list

any node is irrelevant to the list ranking problem, we assume that each node contains only a pointer to its right neighbor. The rightmost node's pointer field is zero.

Example 13.15 Consider the input $A[1 : 6]$ of Figure 13.4. The right neighbor of node $A[1]$ is $A[5]$. The right neighbor of node $A[2]$ is $A[4]$. And so on. Node $A[4]$ is the rightmost node; hence its rank is zero. Node $A[2]$ has rank 1 since the only node to its right is $A[4]$. Node $A[5]$ has rank 3 since the nodes $A[3], A[2]$, and $A[4]$ are to its right. In this example, the left-to-right order of the list nodes is given by $A[6], A[1], A[5], A[3], A[2], A[4]$. □

List ranking can be done sequentially in linear time. First, the list head is determined by examining $A[1 : n]$ to identify the unique i, $1 \leq i \leq n$, such that $A[j] \neq i$, $1 \leq j \leq n$. Node $A[i]$ is the head. Next, a left-to-right scan of the list is made and nodes are assigned the ranks $n-1, n-2, \ldots, 0$ in this order. In this section, we develop two parallel algorithms for list ranking. The first is an n-processor $O(\log n)$ time EREW PRAM algorithm and the second is an $\frac{n}{\log n}$-processor $\widetilde{O}(\log n)$ time EREW PRAM algorithm. The speedups of both the algorithms are $\Theta(n/\log n)$. The efficiency of the first algorithm is $\frac{\Theta(n)}{\Theta(n \log n)} = \Theta(1/\log n)$, whereas the efficiency of the second algorithm is $\frac{\Theta(n)}{\Theta(n)} = \Theta(1)$. Thus the second algorithm is work-optimal but the first algorithm is not.

Deterministic list ranking

One of the crucial ideas behind these parallel algorithms is *pointer jumping.* To begin with, each node in the list points to its immediate right neighbor (see Figure 13.5(a)). In one step of pointer jumping, the right neighbor of every node is modified to be the right neighbor of its right neighbor (see Figure 13.5(b)). Note that if we have n processors (one processor per node), this can be done in $O(1)$ time. Now every node points to a node that was originally a distance of 2 away. In the next step of pointer jumping every node will point to a node that was originally a distance of 4 away. And so on. (See Figure 13.5(c) and (d).) Since the length of the list is n, within $\lceil \log n \rceil$ pointer jumping steps, every node will point to the end of the list.

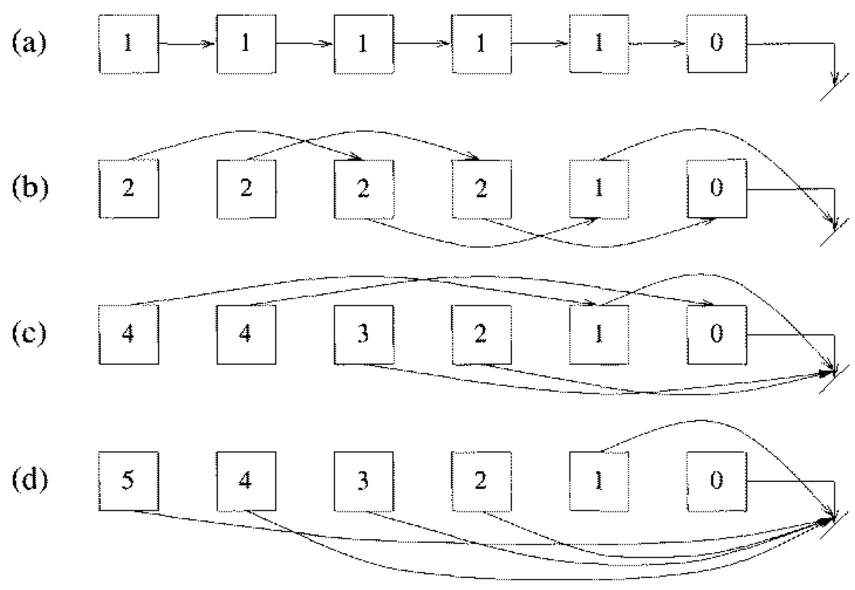


Figure 13.5 Pointer jumping applied to list ranking

In each step of pointer jumping a node also collects information as to how many nodes are between itself and the node it newly points to. This information is easy to accumulate as follows. To start with, set the rank field of each node to 1 except for the rightmost node whose rank field is zero. Let $Rank[i]$ and $Neighbor[i]$ stand for the rank field and the right neighbor of node i. At any step of pointer jumping, $Rank[i]$ is modified to $Rank[i] + Rank[Neighbor[i]]$, in parallel for all nodes other than those with $Neighbor[] = 0$. This is followed by making i point to $Neighbor[Neighbor[i]]$. The complete algorithm is given in Algorithm 13.4. Processor i is associated with the node $A[i]$, $1 \leq i \leq n$.

Example 13.16 For the input of Figure 13.4, Figure 13.6 walks through the steps of Algorithm 13.4. To begin with, every node has a rank of one except for node 4. When $q = 1$, for example, node 1's *Rank* field is changed to two, since its right neighbor (i.e., node 5) has a rank of one. Also, node 1's *Neighbor* field is changed to the neighbor of node 5, which is node 3. And so on. □

```
for $q := 1$ to $\lceil \log n \rceil$ do
    Processor $i$ (in parallel for $1 \leq i \leq n$) does:
        if (Neighbor[$i$] $\neq 0$) then
        {
            $Rank[i] := Rank[i] + Rank[Neighbor[i]]$;
            $Neighbor[i] := Neighbor[Neighbor[i]]$;
        }
```

Algorithm 13.4 An $O(n \log n)$ work list ranking algorithm

Neighbor						*Rank*						
5	4	2	0	3	1	1	1	1	0	1	1	to begin with
3	0	4	0	2	5	2	1	2	0	2	2	$q = 1$
4	0	0	0	0	2	4	1	2	0	3	4	$q = 2$
0	0	0	0	0	0	4	1	2	0	3	5	$q = 3$

Figure 13.6 Algorithm 13.4 working on the input of Figure 13.4

Definition 13.3 Let A be an array of nodes corresponding to a list. Also let node i have a real weight w_i associated with it and let $\oplus$ be any associative binary operation defined on the weights. The *list prefix computation* is the problem of computing, for each node in the list, $w_i \oplus w_{i_1} \oplus w_{i_2} \oplus \cdots \oplus w_{i_k}$, where $i_1, i_2, \ldots, i_k$ are the nodes to the right of i. □

Note that the list ranking problem corresponds to a list prefix sums computation, where each node has a weight of 1 except for the rightmost node whose weight is zero. Algorithm 13.4 can be easily modified to compute list prefixes without any change in the processor and time bounds.

Randomized list ranking

Next we present a work-optimal randomized algorithm for list ranking. Each processor is in charge of computing the rank of $\log n$ nodes in the input. Processor i is assigned the nodes $A[(i-1)\log n+1], A[(i-1)\log n+2], \ldots, A[i\log n]$. The algorithm runs in stages. In any stage, a fraction of the existing nodes is selected and eliminated (or *spliced out*). When a node i is spliced out, the relevant information about this node is stored so that in the future its correct rank can be determined. When the number of remaining nodes is two, the list ranking problem is solved trivially. From the next stage on, spliced-out nodes get inserted back (i.e., *spliced in*). When a node is spliced in, its correct rank will also be determined. Nodes are spliced in in the reverse order in which they were spliced out. The splicing-out process is depicted in Algorithm 13.5.

Node insertion is also done in stages. When a node x is spliced in, its correct rank can be determined as follows: If $LNeighbor[x]$ was the pointer stored when it was spliced out, the rank of x is the current rank of $LNeighbor[x]$ minus the rank that was stored when x was spliced out. Pointers are also adjusted to take into account the fact that now x has been inserted (see Figure 13.7).

We show that the total number s of stages of splicing-out is $\widetilde{O}(\log n)$. If a node gets spliced out in stage q, then it'll be spliced in in stage $2s-q+1$. So the overall structure of the algorithm is as follows: In stages $1, 2, \ldots, s$, nodes are successively spliced out. Stage s is such that there are only two nodes left and one of them is spliced out. In stage $s+1$, the node that was spliced out in stage s is spliced in. In stage $s+2$, the nodes that were spliced out in stage $s-1$ are spliced in. And so on. Following the last stage, we know the ranks of all the nodes in the original list.

The nodes spliced out in any stage are such that (1) from among the nodes associated with any processor, at most one node is selected and (2) no two adjacent nodes of the list are selected. Since in any stage, processor q considers only one node, at most one of its nodes is spliced out. Also realize that no two adjacent nodes from the list are spliced out in any stage. This is because a processor with a *head* proceeds to splice the chosen node only if the right neighbor's processor does not have a *head*. Therefore, the time spent by any processor in a given stage is only $O(1)$.

To compute the total run time of the algorithm, we only have to compute the value of s, the number of stages. This can be done if we can estimate the number of nodes that will be spliced out in any stage. If q is any processor, in a given stage its chosen node x is spliced out with probability at least $\frac{1}{4}$. The reasons are (1) the probability for q to come up with a *head* is $\frac{1}{2}$ and (2)

Step 1. Doubly link the list. With n processors, this can be done in $O(1)$ time as follows. Processor i is associated with node $A[i]$ (for $1 \leq i \leq n$). In one step, processor i writes i in memory cell $Neighbor[i]$ so that in the next step the processor associated with the node $A[Neighbor[i]]$ will know its left neighbor. Using the slow-down lemma, this can also be done in $O(\log n)$ time using $\frac{n}{\log n}$ processors. Let $LNeighbor[i]$ and $RNeighbor[i]$ stand for the left and right neighbors of node $A[i]$. To begin with, the rank field of each node is as shown in Figure 13.5(a).

Step 2. while (the number of remaining nodes is > 2) **do**

{

> **Step a.** Processor q ($1 \leq q \leq \frac{n}{\log n}$) considers the next unspliced node (call it x) associated with it. It flips a two-sided coin. If the outcome is a *tail*, the processor becomes idle for the rest of the stage. In the next stage it again attempts to splice out x. On the other hand, if the coin flip results in a *head*, it checks whether the right neighbor of x is being considered by the corresponding processor. If the right neighbor of x is being considered and the coin flip of that processor is also a *head*, processor q gives up and is idle for the rest of the stage. If not, q decides to splice out x.
>
> **Step b.** When node x is spliced out, processor q stores in node x the stage number, the pointer $LNeighbor[x]$, and $Rank[LNeighbor[x]]$. $Rank[LNeighbor[x]]$ at this time is the number of nodes between $LNeighbor[x]$ and x. Processor q also sets $Rank[LNeighbor[x]] := Rank[LNeighbor[x]] + Rank[x]$;. Finally it sets $RNeighbor[LNeighbor[x]] := RNeighbor[x]$; and $LNeighbor\ [RNeighbor[x]] := LNeighbor[x]$;.

}

Algorithm 13.5 Splicing out nodes

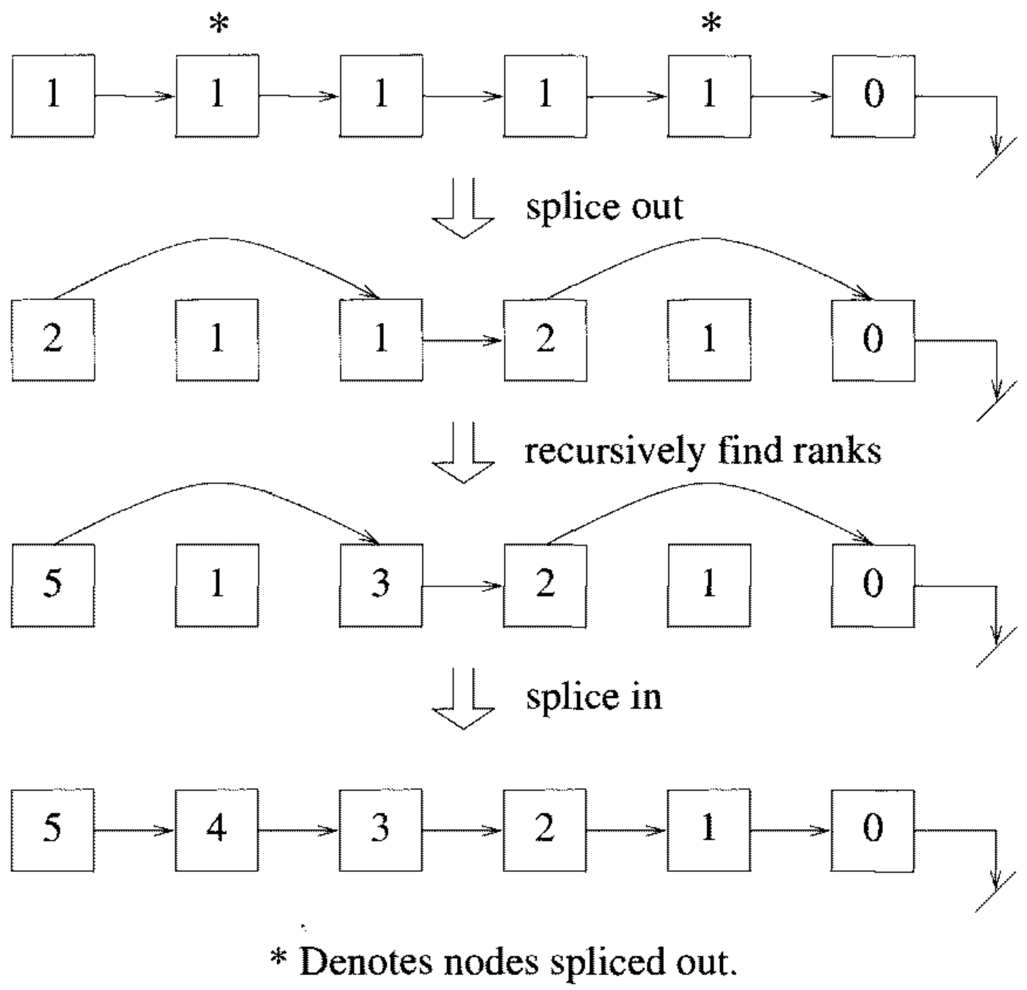


Figure 13.7 Splicing in and splicing out nodes. Only right links of nodes are shown.

the probability that the right neighbor of x (call it y) either has not been chosen or, if chosen, y's processor has a tail is $\geq \frac{1}{2}$. Since events 1 and 2 are independent, the claim follows.

Every processor begins the algorithm with $\log n$ nodes, and in every stage it has a probability of $\geq \frac{1}{4}$ of splicing out a node. Thus it follows that the expected value of s is $\leq 4 \log n$. We can also use Chernoff bounds (Equation 1.2 with parameters $12\alpha \log n$ and $\frac{1}{4}$ with $\epsilon = \frac{2}{3}$) to show that the value of s is $\leq 12\alpha \log n$ with probability $> (1 - n^{-\alpha})$ for any $\alpha \geq 1$. As a result we get the following theorem.

Theorem 13.3 List ranking on a list of length n can be performed in $\widetilde{O}(\log n)$ time using $\frac{n}{\log n}$ EREW PRAM processors.

EXERCISES

1. There is some data in cell M_1 of the global memory. The goal is to make copies of this data in cells $M_2, M_3, \ldots, M_n$. Show how you accomplish this in $O(\log n)$ time using n EREW PRAM processors.

2. Present an $O(\log n)$ time $\frac{n}{\log n}$ processor EREW PRAM algorithm for the problem of Exercise 1.

3. Show that Theorem 13.2 holds for the EREW PRAM also.

4. Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$. Present an $O(\log n)$ time $\frac{n}{\log n}$-processor CREW PRAM algorithm to evaluate the polynomial f at a given point y.

5. The *segmented prefix* problem is defined as follows: Array A has n elements from some domain Σ. Array $B[1 : n]$ is a Boolean array with $B[1] = 1$. Define a *segment* of A to be $A[i : j]$, where $B[i] = 1, B[j] = 1$, and $B[k] = 0,\ i < k < j$. As a convention assume that $B[n + 1] = 1$. The problem is to perform several independent prefix computations on A, one for each segment. Show how to solve this problem in $O(\log n)$ time on a $\frac{n}{\log n}$-processor CREW PRAM.

6. If $k_1, k_2, \ldots, k_n$ are from Σ and $\oplus$ is a binary associative operator on Σ, the *suffix computation problem* is to output $k_n,\ k_{n-1} \oplus k_n,\ \ldots,\ k_1 \oplus k_2 \oplus \cdots \oplus k_n$. Show how you'll solve this problem in $O(\log n)$ time on an $\frac{n}{\log n}$-processor CREW PRAM as well as EREW PRAM.

7. The inputs are an array A of n elements and an element x. The goal is to rearrange the elements of A such that all the elements of A that are less than or equal to x appear first (in successive cells) followed by

the rest of the elements. Give an $O(\log n)$-time $\frac{n}{\log n}$-processor CREW PRAM algorithm for this problem.

8. Array A is an array of n elements, where each element has a label of zero or one. The problem is to rearrange A so that all the elements with a zero label appear first, followed by all the others. Show how to perform this rearrangement in $O(\log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors.

9. Let A be an array of n keys. The rank of a key x in A is defined to be one plus the number of elements in A that are less than x. Given A and an x, show how you compute the rank of x using $\frac{n}{\log n}$ CREW PRAM processors. Your algorithm should run in $O(\log n)$ time.

10. Show how you modify Algorithm 13.4 and the randomized list ranking algorithm so they solve the list prefix computation problem.

11. Array A is an array of nodes representing a list. Each node has a label of either zero or one. You are supposed to split the list in two, the first list containing all the elements with a zero label and the second list consisting of all the rest of the nodes. The original order of nodes should be preserved. For example, if x is a node with a zero label and the next right node with a zero label is z, then x should have z as its right neighbor in the first list created. Present an $\widetilde{O}(\log n)$ time algorithm for this problem. You can use up to $\frac{n}{\log n}$ CREW PRAM processors.

12. Present an $O(\log n)$ time $\frac{n^2}{\log n}$-processor CREW PRAM algorithm to multiply an $n \times n$ matrix with an $n \times 1$ column vector. How will you solve the same problem on an EREW PRAM? You can use $O(n^2)$ global memory. (*Hint:* See Exercise 1.)

13. Show how to multiply two $n \times n$ matrices using $\frac{n^3}{\log n}$ CREW PRAM processors and $O(\log n)$ time.

14. Strassen's algorithm for matrix multiplication was introduced in Section 3.7. Using the same technique, design a divide-and-conquer algorithm for matrix multiplication that uses only $n^{\log_2 7}$ CREW PRAM processors and runs in $O(\log n)$ time.

15. Prove that two $n \times n$ boolean matrices can be multiplied in $O(1)$ time on any of the CRCW PRAMs. What is the processor bound of your algorithm?

16. In Section 10.3, a divide-and-conquer algorithm was presented for inverting a triangular matrix. Parallelize this algorithm on the CREW

PRAM to get a run time of $O(\log^2 n)$. What is the processor bound of your algorithm?

17. A *tridiagonal* matrix has nonzero elements only in the diagonal and its two neighboring diagonals (one below and one above). Present an $O(\log n)$ time $\frac{n}{\log n}$-processor CREW PRAM algorithm to solve the system of linear equations $Ax = b$, where A is an $n \times n$ tridiagonal matrix and x (unknown) and b are $n \times 1$ column vectors.

18. An optimal algorithm for FFT was given in Section 9.3. Present a work-optimal parallelization of this algorithm on the CREW PRAM. Your algorithm should run in $O(\log n)$ time.

19. Let X and Y be two sorted arrays with n elements each. Show how you merge these in $O(1)$ time on the common CRCW PRAM. How many processors do you use?

13.4 SELECTION

The problem of selection was introduced in Section 3.6. Recall that this problem takes as input a sequence of n keys and an integer i, $1 \le i \le n$, and outputs the ith smallest key from the sequence. Several algorithms were presented for this problem in Section 3.6. One of these algorithms (Algorithm 3.19) has a worst-case run time of $O(n)$ and hence is optimal. In this section we study the parallel complexity of selection. We start by presenting algorithms for many special cases. Finally, we give an $\widetilde{O}(\log n)$ time $\frac{n}{\log n}$-processor common CRCW PRAM algorithm. Since the work done in this algorithm is $\widetilde{O}(n)$, it is work-optimal.

13.4.1 Maximal Selection with n^2 Processors

Here we consider the problem of selection for $i = n$; that is, we are interested in finding the maximum of n given numbers. This can be done in $O(1)$ time using an n^2-processor CRCW PRAM.

Let $k_1, k_2, \ldots, k_n$ be the input. The idea is to perform all pairs of comparisons in one step using n^2 processors. If we name the processors p_{ij} (for $1 \le i, j \le n$), processor p_{ij} computes $x_{ij} = (k_i < k_j)$. Without loss of generality assume that all the keys are distinct. Even if they are not, they can be made distinct by replacing key k_i with the tuple (k_i, i) (for $1 \le i \le n$); this amounts to appending each key with only a $(\log n)$-bit number. Of all the input keys, there is only one key k which when compared with every other key, would have yielded the same bit zero. This key can be identified using the boolean OR algorithm (Algorithm 13.1) and is the maximum of all. The resultant algorithm appears as Algorithm 13.6.

Step 0. If $n = 1$, output the key.

Step 1. Processor p_{ij} (**for** each $1 \le i, j \le n$ **in parallel**) computes $x_{ij} = (k_i < k_j)$.

Step 2. The n^2 processors are grouped into n groups $G_1, G_2, \ldots, G_n$, where G_i $(1 \le i \le n)$ consists of the processors $p_{i1}, p_{i2}, \ldots, p_{in}$. Each group G_i computes the boolean OR of $x_{i1}, x_{i2}, \ldots, x_{in}$.

Step 3. If G_i computes a zero in step 2, then processor p_{i1} outputs k_i as the answer.

Algorithm 13.6 Finding the maximum in $O(1)$ time

Steps 1 and 3 of this algorithm take one unit of time each. Step 2 takes $O(1)$ time (see Theorem 13.1). Thus the whole algorithm runs in $O(1)$ time; this implies the following theorem.

Theorem 13.4 The maximum of n keys can be computed in $O(1)$ time using n^2 common CRCW PRAM processors. □

Note that the speedup of Algorithm 13.6 is $\frac{\Theta(n)}{1} = \Theta(n)$. Total work done by this algorithm is $\Theta(n^2)$. Hence its efficiency is $\frac{\Theta(n)}{\Theta(n^2)} = \Theta(1/n)$. Clearly, this algorithm is not work-optimal!

13.4.2 Finding the Maximum Using n Processors

Now we show that maximal selection can be done in $O(\log\log n)$ time using n common CRCW PRAM processors. The technique to be employed is divide-and-conquer. To simplify the discussion, we assume n is a perfect square (when n is not a perfect square, replace $\sqrt{n}$ by $\lceil\sqrt{n}\rceil$ in the following discussion).

Let the input sequence be $k_1, k_2, \ldots, k_n$. We are interested in developing an algorithm that can find the maximum of n keys using n processors. Let $T(n)$ be the run time of this algorithm. We partition the input into $\sqrt{n}$ parts, where each part consists of $\sqrt{n}$ keys. Allocate $\sqrt{n}$ processors to each part so that the maximum of each part can be computed in parallel. Since the recursive maximal selection of each part involves $\sqrt{n}$ keys and an equal

---

Step 0. If $n = 1$, return k_1.

Step 1. Partition the input keys into $\sqrt{n}$ parts $K_1, K_2, \ldots, K_{\sqrt{n}}$ where K_i consists of $k_{(i-1)\sqrt{n}+1}, k_{(i-1)\sqrt{n}+2}, \ldots, k_{i\sqrt{n}}$. Similarly partition the processors so that P_i $(1 \leq i \leq \sqrt{n})$ consists of the processors $p_{(i-1)\sqrt{n}+1}, p_{(i-1)\sqrt{n}+2}, \ldots, p_{i\sqrt{n}}$. Let P_i find the maximum of K_i recursively (for $1 \leq i \leq \sqrt{n}$).

Step 2. If $M_1, M_2, \ldots, M_{\sqrt{n}}$ are the group maxima, find and output the maximum of these maxima employing Algorithm 13.6.

Algorithm 13.7 Maximal selection in $O(\log \log n)$ time

number of processors, this can be done in $T(\sqrt{n})$ time. Let $M_1, M_2, \ldots, M_{\sqrt{n}}$ be the group maxima. The answer we are supposed to output is the maximum of these maxima. Since now we only have $\sqrt{n}$ keys, we can find the maximum of these employing all the n processors (see Algorithm 13.7).

Step 1 of this algorithm takes $T(\sqrt{n})$ time and step 2 takes $O(1)$ time (c.f. Theorem 13.4). Thus $T(n)$ satisfies the recurrence

$$T(n) = T(\sqrt{n}) + O(1)$$

which solves to $T(n) = O(\log \log n)$. Therefore the following theorem arises.

Theorem 13.5 The maximum of n keys can be found in $O(\log \log n)$ time using n common CRCW PRAM processors. $\square$

Total work done by Algorithm 13.7 is $\Theta(n \log \log n)$ and its efficiency is $\frac{\Theta(n)}{\Theta(n \log \log n)} = \Theta(1/\log \log n)$. Thus this algorithm is not work-optimal.

13.4.3 Maximal Selection Among Integers

Consider again the problem of finding the maximum of n given keys. If each one of these keys is a bit, then the problem of finding the maximum reduces to computing the boolean OR of n bits and hence can be done in $O(1)$ time using n common CRCW PRAM processors (see Algorithm 13.1). This raises the following question: What can be the maximum magnitude of each key if we desire a constant time algorithm for maximal selection using n processors? Answering this question in its full generality is beyond the

```
for $i := 1$ to $2c$ do
{
    Step 1. Find the maximum of all the alive keys with
    respect to their $i$th parts. Let $M$ be the maximum.

    Step 2. Delete each alive key whose $i$th part is $< M$.
}
Output one of the alive keys.
```

Algorithm 13.8 Integer maximum

scope of this book. Instead we show that if each key is an integer in the range $[0, n^c]$, where c is a constant, maximal selection can be done work-optimally in $O(1)$ time. Speedup of this algorithm is $\Theta(n)$ and its efficiency is $\Theta(1)$.

Since each key is of magnitude at most n^c, it follows that each key is a binary number with $\leq c \log n$ bits. Without loss of generality assume that every key is of length exactly equal to $c \log n$. (We can add leading zero bits to numbers with fewer bits.) Suppose we find the maximum of the n keys only with respect to their $\frac{\log n}{2}$ most significant bits (MSBs) (see Figure 13.8). Let M be the maximum value. Then, any key whose $\frac{\log n}{2}$ MSBs do not equal M can be dropped from future consideration since it cannot possibly be the maximum. After this step, many keys can potentially survive. Next we compute the maximum of the remaining keys with respect to their next $\frac{\log n}{2}$ MSBs and drop keys that cannot possibly be the maximum. We repeat this basic step $2c$ times (once for every $\frac{\log n}{2}$ bits in the input keys). One of the keys that survives the very last step can be output as the maximum. Refer to the $\frac{\log n}{2}$ MSBs of any key as its first part, the next most significant $\frac{\log n}{2}$ bits as its second part, and so on. There are $2c$ parts for each key. The $2c$th part may have less than $\frac{\log n}{2}$ bits. The algorithm is summarized in Algorithm 13.8. To begin with, all the keys are *alive*.

We now show that step 1 of Algorithm 13.8 can be completed in $O(1)$ time using n common CRCW PRAM processors. Note that if a key has at most $\frac{\log n}{2}$ bits, its maximum magnitude is $\sqrt{n} - 1$. Thus each step of Algorithm 13.8 is nothing but the task of finding the maximum of n keys, where each key is an integer in the range $[0, \sqrt{n} - 1]$. Assign one processor to

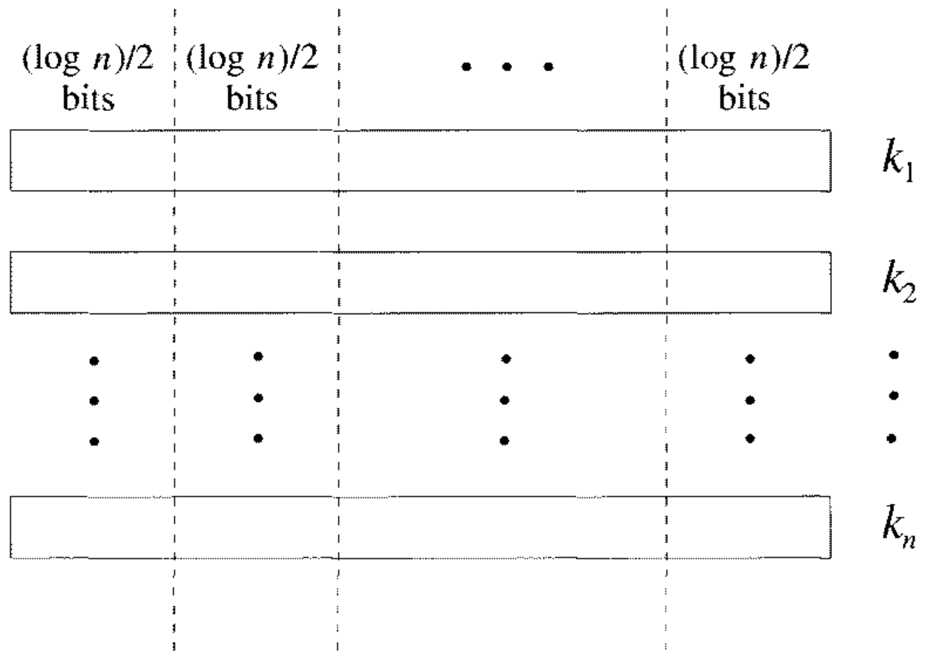


Figure 13.8 Finding the integer maximum

each key. Make use of $\sqrt{n}$ global memory cells (which are initialized to $-\infty$). Call these cells $M_0, M_1, \ldots, M_{\sqrt{n}-1}$. In one parallel write step, if processor i has a key k_i, then it tries to write k_i in M_{k_i}. For example, if processor i has a key valued 10, it will attempt to write 10 in M_{10}. After this write step, the problem of computing the maximum of the n keys reduces to computing the maximum of the contents of $M_0, M_1, \ldots, M_{\sqrt{n}-1}$. Since these are only $\sqrt{n}$ numbers, their maximum can be found in $O(1)$ time using n processors (see Theorem 13.4). As a result we get the following theorem.

Theorem 13.6 The maximum of n keys can be found in $O(1)$ time using n CRCW PRAM processors provided the keys are integers in the range $[0, n^c]$ for any constant c. □

Example 13.17 Consider the problem of finding the maximum of the following four four-bit keys: $k_1 = 1010, k_2 = 1101, k_3 = 0110$, and $k_4 = 1100$. Here $n = 4$, $c = 2$, and $\log n = 2$. In the first basic step of Algorithm 13.8, the maximum of the four numbers with respect to their MSB is 1. Thus k_3 gets eliminated. In the second basic step, the maximum of k_1, k_2, and k_4 with respect to their second part (i.e., second MSB) is found. As a result, k_1 is dropped. In the third basic step, no key gets eliminated. Finally, in the fourth basic step, k_4 is deleted to output k_2 as the maximum. □

13.4.4 General Selection Using n^2 Processors

Let $X = k_1, k_2, \ldots, k_n$ be a given sequence of distinct keys and say we are interested in selecting the ith smallest key. The *rank* of any key x in X is defined to be one plus the number of keys of X that are less than x. If we have $\frac{n}{\log n}$ CREW PRAM processors, we can compute the rank of any given key x in $O(\log n)$ time (see Section 13.3, Exercise 9).

If we have $\frac{n^2}{\log n}$ processors, we can group them into $G_1, G_2, \ldots, G_n$ such that each G_j has $\frac{n}{\log n}$ processors. G_j computes the rank of k_j in X (for $1 \leq j \leq n$) using the algorithm of Section 13.3, Exercise 9. This will take $O(\log n)$ time. One of the processors in the group whose rank was i will output the answer. Thus we get the following theorem.

Theorem 13.7 Selection can be performed in $O(\log n)$ time using $\frac{n^2}{\log n}$ CREW PRAM processors. □

The algorithm of Theorem 13.7 has a speedup of $\frac{\Theta(n)}{\Theta(\log n)} = \Theta(n/\log n)$. Its efficiency is $\frac{\Theta(n)}{\Theta(n^2)} = \Theta(1/n)$; that is, the algorithm is not work-optimal!

13.4.5 A Work-Optimal Randomized Algorithm (*)

In this section we show that selection can be done in $\widetilde{O}(\log n)$ time using $\frac{n}{\log n}$ common CRCW PRAM processors. The randomized algorithm chooses a random sample (call it S) from X of size $n^{1-\epsilon}$ (for some suitable ϵ) and selects two elements of S as splitters. A choice of $\epsilon = 0.6$ suffices. Let l_1 and l_2 be the splitters. The keys l_1 and l_2 are such that the element to be selected has a value between l_1 and l_2 with high probability (abbreviated w.h.p.). In addition, the number of keys of X that have a value in the range $[l_1, l_2]$ is small, $\widetilde{O}(n^{(1+\epsilon)/2}\sqrt{\log n})$ to be specific.

Having chosen l_1 and l_2, we partition X into X_1, X_2, and X_3, where $X_1 = \{x \in X | x < l_1\}$, $X_2 = \{x \in X | l_1 \leq x \leq l_2\}$, and $X_3 = \{x \in X | x > l_2\}$. While performing this partitioning, we also count the size of each part. If $|X_1| < i \leq |X_1| + |X_2|$, the element to be selected lies in X_2. If this is the case, we proceed further. If not, we start all over again. We can show that the ith smallest element of X will indeed belong to X_2 with high probability and also that $|X_2| = N = \widetilde{O}(n^{(1+\epsilon)/2}\sqrt{\log n})$. The element to be selected will be the $(i - |X_1|)$th smallest element of X_2.

The preceding process of sampling and elimination is repeated until the number of remaining keys is $\leq n^{0.4}$. After this, we perform an appropriate selection from out of the remaining keys using the algorithm of Theorem 13.7. More details of the algorithm are given in Algorithm 13.9. To begin with, each input key is *alive*. There are $\frac{n}{\log n}$ processors and each processor gets $\log n$ keys. Concentration (in steps 3 and 7) refers to collecting the relevant keys and putting them in successive cells in global memory (see Section 13.3, Exercise 8).

Let a *stage* refer to one run of the **while** loop. The number of samples in any given stage is binomial with parameters N and $N^{-\epsilon}$. Thus the expected number of sample keys is $N^{1-\epsilon}$. Using Chernoff bounds, we can show that $|S|$ is $\widetilde{O}(N^{1-\epsilon})$.

Let S be a sample of s elements from a set X of n elements. Let $r_j =$ rank(select$(j, S), X)$. Here rank(x, X) is defined to be one plus the number of elements in X that are less than x, and select(j, S) is defined to be the jth smallest element of S. The following lemma provides a high probability confidence interval for r_j.

Lemma 13.3 For every α, Prob. $\left(|r_j - j\frac{n}{s}| > \sqrt{3\alpha}\frac{n}{\sqrt{s}}\sqrt{\log n}\right) < n^{-\alpha}$. □

For a proof of this lemma, see the references supplied at the end of this chapter. Using this lemma, we can show that only $\widetilde{O}(N^{(1+\epsilon)/2}\sqrt{\log N})$ keys survive at the end of any stage, where N is the number of alive keys at the beginning of this stage. This in turn implies there are only $\widetilde{O}(1)$ stages in the algorithm.

Broadcasting in any parallel machine is the operation of sending a specific information to a specified set of processors. In the case of a CREW PRAM, broadcasting can be done in $O(1)$ time with a concurrent read operation. In Algorithm 13.9, steps 1 and 2 take $O(\log n)$ time each. In steps 3 and 6, concentration can be done using a prefix sums computation followed by a write. Thus step 3 takes $O(\log n)$ time. Also, the sample size in steps 3 and 6 is $\widetilde{O}(n^{0.4})$. Thus these keys can be sorted in $O(\log n)$ time using a simple algorithm (given in Section 13.6). Alternatively, the selections performed in steps 3 and 6 can be accomplished using the algorithm of Theorem 13.7. Two prefix sums computations are done in step 5 for a total of $O(\log n)$ time. Therefore, each stage of the algorithm runs in $\widetilde{O}(\log n)$ time and the whole algorithm also terminates in time $\widetilde{O}(\log n)$; this implies the following theorem.

Theorem 13.8 Selection from out of n keys can be performed in $\widetilde{O}(\log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors. □

EXERCISES

1. Present an $O(\log \log n)$ time algorithm for finding the maximum of n arbitrary numbers using $\frac{n}{\log \log n}$ common CRCW PRAM processors.

```
$N := n$; // $N$ at any time is the number of live keys
while ($N > n^{0.4}$) do
{
    Step 1. Each live key is included in the random sample $S$ with
    probability $\frac{1}{N^\epsilon}$. This step takes $\log n$ time and with high proba-
    bility, $O(N^{1-\epsilon})$ keys (from among all the processors) are in the
    random sample.

    Step 2. All processors perform a prefix sums operation to com-
    pute the number of keys in the sample. Let $q$ be this num-
    ber. Broadcast $q$ to all the processors. If $q$ is not in the range
    $[0.5N^{1-\epsilon}, 1.5N^{1-\epsilon}]$, go to step 1.

    Step 3. Concentrate and sort the sample keys.

    Step 4. Select keys $l_1$ and $l_2$ from $S$ with ranks $\left\lceil \frac{iq}{N} \right\rceil - d\sqrt{q \log N}$
    and $\left\lceil \frac{iq}{N} \right\rceil + d\sqrt{q \log N}$, respectively, $d$ being a constant $> \sqrt{3\alpha}$.
    Broadcast $l_1$ and $l_2$ to all the processors. The key to be selected
    has a value in the range $[l_1, l_2]$ w.h.p.

    Step 5. Count the number $r$ of live keys that are in the range
    $[l_1, l_2]$. Also count the number of live keys that are $< l_1$. Let this
    count be $t$. Broadcast $r$ and $t$ to all the processors. If $i$ is not in
    the interval $(t, t + r]$ or if $r$ is $\neq O(N^{(1+\epsilon)/2}\sqrt{\log N})$, go to step
    1; else kill (i.e., delete) all the live keys with a value $< l_1$ or $> l_2$
    and set $i := i - t$ and $N := r$.
}
Step 6. Concentrate and sort the live keys. Identify and output the $i$th
smallest key.
```

Algorithm 13.9 A work-optimal randomized selection algorithm

2. Show that prefix minima computation can be performed in $O(\log\log n)$ time using $\frac{n}{\log\log n}$ common CRCW PRAM processors.

3. Given an array A of n elements, we would like to find the largest i such that $A[i] = 1$. Give an $O(1)$ time algorithm for this problem on an n-processor common CRCW PRAM.

4. Algorithm 13.6 runs in time $O(1)$ using n^2 processors. Show how to modify this algorithm so that the maximum of n elements can be found in $O(1)$ time using $n^{1+\epsilon}$ processors for any fixed $\epsilon > 0$.

5. If k is any integer > 1, the kth quantiles of a sequence X of n numbers are defined to be those $k-1$ elements of X that evenly divide X. For example, if $k = 2$, there is only one quantile, namely, the median of X. Show that the kth quantiles of any given X can be computed in $\widetilde{O}(\log k \log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors.

6. Present an $\widetilde{O}(1)$ time n-processor algorithm for finding the maximum of n given arbitrary numbers. (*Hint*: Employ random sampling along the same lines as in Algorithm 13.9.)

7. Given an array A of n elements, the problem is to find any element of A that is greater than or equal to the median. Present an $\widetilde{O}(1)$ time algorithm for this problem. You can use a maximum of $\log^2 n$ CRCW PRAM processors.

8. The distinct elements problem was posed in Section 13.2, Exercises 4 and 5. Assume that the elements are integers in the range $[0, n^c]$, where c is a constant. Show how to solve the distinct elements problem in $O(1)$ time using n CRCW PRAM processors. (*Hint*: You can use $O(n^c)$ global memory.)

9. Show how to reduce the space bound of the above algorithm to $O(n^{1+\epsilon})$ for any fixed $\epsilon > 0$. (*Hint*: Use the idea of radix reduction (see Figure 13.8).)

10. If X is a sorted array of elements and x is any element, we can make use of binary search to check whether $x \in X$ in $O(\log n)$ time sequentially. Assume that we have k processors, where $k > 1$. Can the search be done faster? One way of making use of all the k processors is to partition X into k nearly equal parts. Each processor is assigned a part. A processor then compares x with the two endpoints of the part assigned to it to check whether x falls in its part. If no part has x, then the answer is immediate. If $x \in X$, only one part survives. In the next step, all the k processors can work on the surviving part in a similar manner. This is continued until the position of x is pinpointed.

The preceding algorithm is called a *k-ary search*. What is the run time of a k-ary search algorithm? Show that if there are n^ϵ CREW PRAM processors (for any fixed $\epsilon > 0$), we can check whether $x \in X$ in $O(1)$ time.

13.5 MERGING

The problem of merging is to take two sorted sequences as input and produce a sorted sequence of all the elements. This problem was studied in Chapter 3 and an $O(n)$ time algorithm (Algorithm 3.8) was presented. Merging is an important problem. For example, an efficient merging algorithm can lead to an efficient sorting algorithm (as we saw in Chapter 3). The same is true in parallel computing also. In this section we study the parallel complexity of merging.

13.5.1 A Logarithmic Time Algorithm

Let $X_1 = k_1, k_2, \ldots, k_m$ and $X_2 = k_{m+1}, k_{m+2}, \ldots, k_{2m}$ be the input sorted sequences to be merged. Assume without loss of generality that m is an integral power of 2 and that the keys are distinct. Note that the merging of X_1 and X_2 can be reduced to computing the rank of each key k in $X_1 \cup X_2$. If we know the rank of each key, then the keys can be merged by writing the key whose rank is i into global memory cell i. This writing will take only one time unit if we have $n = 2m$ processors.

For any key k, let its rank in X_1 (X_2) be denoted as r_k^1 (r_k^2). If $k = k_j \in X_1$, then note that $r_k^1 = j$. If we allocate a single processor π to k, π can perform a binary search (see Algorithms 3.2 and 3.3) on X_2 and figure out the number q of keys in X_2 that are less than k. Once q is known, π can compute k's rank in $X_1 \cup X_2$ as $j + q$. If k belongs to X_2, a similar procedure can be used to compute its rank in $X_1 \cup X_2$. In summary, if we have $2m$ processors (one processor per key), merging can be completed in $O(\log m)$ time.

Theorem 13.9 Merging of two sorted sequences each of length m can be completed in $O(\log m)$ time using m CREW PRAM processors. □

Since two sorted sequences of length m each can be sequentially merged in $\Theta(m)$ time, the speedup of the above algorithm is $\frac{\Theta(m)}{\Theta(\log m)} = \Theta(m/\log m)$; its efficiency is $\frac{\Theta(m)}{\Theta(m \log m)} = \Theta(1/\log m)$. This algorithm is not work-optimal!

13.5.2 Odd-Even Merge

Odd-even merge is a merging algorithm based on divide-and-conquer that yields itself to efficient parallelization. If $X_1 = k_1, k_2, \ldots, k_m$ and $X_2 = k_{m+1}, k_{m+2}, \ldots, k_{2m}$ (where m is an integral power of 2) are the two sorted sequences to be merged, then Algorithm 13.10) uses $2m$ processors.

Step 0. If $m = 1$, merge the sequences with one comparison.

Step 1. Partition X_1 and X_2 into their odd and even parts. That is, partition X_1 into $X_1^{odd} = k_1, k_3, \ldots, k_{m-1}$ and $X_1^{even} = k_2, k_4, \ldots, k_m$. Similarly, partition X_2 into X_2^{odd} and X_2^{even}.

Step 2. Recursively merge X_1^{odd} with X_2^{odd} using m processors. Let $L_1 = \ell_1, \ell_2, \ldots, \ell_m$ be the result. Note that $X_1^{odd}, X_1^{even}, X_2^{odd}$, and X_2^{even} are in sorted order. At the same time merge X_1^{even} with X_2^{even} using the other m processors to get $L_2 = \ell_{m+1}, \ell_{m+2}, \ldots, \ell_{2m}$.

Step 3. *Shuffle* L_1 and L_2; that is, form the sequence $L = \ell_1, \ell_{m+1}, \ell_2, \ell_{m+2}, \ldots, \ell_m, \ell_{2m}$. Compare every pair (ℓ_{m+i}, ℓ_{i+1}) and interchange them if they are out of order. That is, compare ℓ_{m+1} with ℓ_2 and interchange them if need be, compare ℓ_{m+2} with ℓ_3 and interchange them if need be, and so on. Output the resultant sequence.

Algorithm 13.10 Odd-even merge algorithm

Example 13.18 Let $X_1 = 2, 5, 8, 11, 13, 16, 21, 25$ and $X_2 = 4, 9, 12, 18, 23, 27, 31, 34$. Figure 13.9 shows how the odd-even merge algorithm can be used to merge these two sorted sequences. □

Let $M(m)$ be the run time of Algorithm 13.10 on two sorted sequences of length m each using $2m$ processors. Then, step 1 takes $O(1)$ time. Step 2 takes $M(m/2)$ time. Step 3 takes $O(1)$ time. This yields the following recurrence relation: $M(m) = M(m/2) + O(1)$ which solves to $M(m) = O(\log m)$. Thus we arrive at the following theorem.

Theorem 13.10 Two sorted sequences of length m each can be merged in $O(\log m)$ time using $2m$ EREW PRAM processors. □

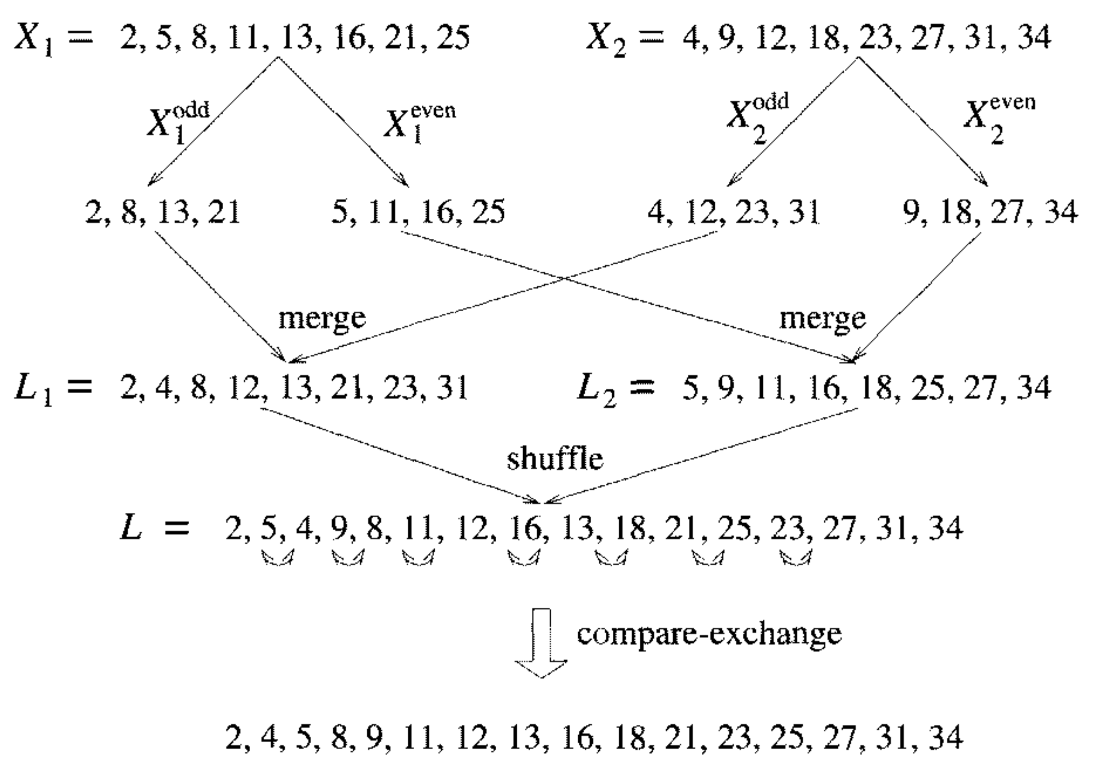


Figure 13.9 Odd-even merge – an example

The correctness of the merging algorithm can be established using the *zero-one principle.* The validity of this principle is not proved here.

Theorem 13.11 [Zero-one principle] If any *oblivious* comparison-based sorting algorithm sorts an arbitrary sequence of n zeros and ones correctly, then it will also sort any sequence of n arbitrary keys. □

A comparison-based sorting algorithm is said to be *oblivious* if the sequence of cells to be compared in the algorithm is prespecified. For example, the next pair of cells to be compared cannot depend on the outcome of comparisons made in the previous steps.

Example 13.19 Let $k_1, k_2, \ldots, k_n$ be a sequence of bits. One way of sorting this sequence is to count the number z of zeros in the sequence, followed by writing z zeros and $n - z$ ones in succession. The zero-one principle cannot be applied to this algorithm since the algorithm is not comparison based.

Also, the quicksort algorithm (Section 3.5), even though comparison based, is not oblivious. The reason is as follows. At any point in the algorithm, the next pair of cells to be compared depends on the number of keys in each of the two parts. For example, if there are only two keys in the first part, these two cells are compared next. On the other hand, if there are ten elements in the first part, then the comparison sequence is different.

Note that merging is a special case of sorting and also the odd-even merge algorithm is oblivious. This is because the sequence of cells to be compared is always the same. Thus the zero-one principle can be applied to odd-even merge. □

Theorem 13.12 Algorithm 13.10 correctly merges any two sorted sequences of arbitrary numbers.

Proof: The correctness of Algorithm 13.10 can be proved using the zero-one principle. Let X_1 and X_2 be sorted sequences of zeros and ones with $|X_1| = |X_2| = m$. Both X_1 and X_2 have a sequence of zeros followed by a sequence of ones. Let q_1 (q_2) be the number of zeros in X_1 (X_2, respectively). The number of zeros in X_1^{odd} is $\lceil q_1/2 \rceil$ and the number of zeros in X_1^{even} is $\lfloor q_1/2 \rfloor$. Thus the number of zeros in L_1 is $z_1 = \lceil q_1/2 \rceil + \lceil q_2/2 \rceil$ and the number of zeros in L_2 is $z_2 = \lfloor q_1/2 \rfloor + \lfloor q_2/2 \rfloor$.

The difference between z_1 and z_2 is at most 2. This difference is exactly two if and only if both q_1 and q_2 are odd. In all the other cases the difference is ≤ 1. Assume that $|z_1 - z_2| = 2$. The other cases are similar. L_1 has two more zeros than L_2. When these two are shuffled in step 3, L contains a sequence of zeros, followed by 10 and then by a sequence of ones. The only unsorted portion in L (also called the *dirty sequence*) will be 10. When the final comparison and interchange is performed in step 3, the dirty sequence and the whole sequence are sorted. □

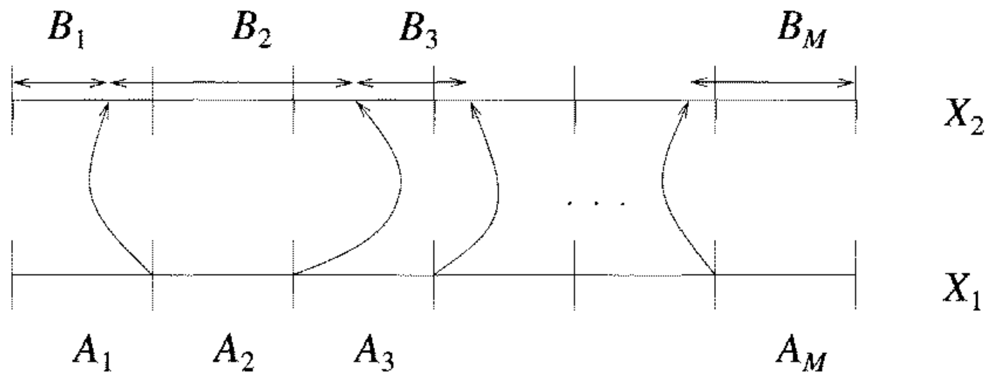


Figure 13.10 A work-optimal merging algorithm

13.5.3 A Work-Optimal Algorithm

In this section we show how to merge two sorted sequences with m elements each in logarithmic time using only $\frac{m}{\log m}$ processors. This algorithm reduces the original problem into $O(\frac{m}{\log m})$ subproblems, where each subproblem is that of merging two sorted sequences each of length $O(\log m)$. Each such subproblem can be solved using the sequential algorithm (Algorithm 3.8) of Chapter 3 in $O(\log m)$ time.

Thus the algorithm is complete if we describe how to reduce the original problem into $O(\frac{m}{\log m})$ subproblems. Let X_1 and X_2 be the sequences to be merged. Partition X_1 into $\frac{m}{\log m}$ parts, where there are $\log m$ keys in each part. Call these parts $A_1, A_2, \ldots, A_M$, where $M = \frac{m}{\log m}$. Let the largest key in A_i be ℓ_i (for $i = 1, 2, \ldots, M$). Assign a processor to each of these ℓ_i's. The processor associated with ℓ_i performs a binary search on X_2 to find the correct (i.e., sorted) position of ℓ_i in X_2. This induces a partitioning of X_2 into M parts. Note that some of these parts could be empty (see Figure 13.10). Let the corresponding parts of X_2 be $B_1, B_2, \ldots, B_M$. Call B_i the *corresponding subset of A_i in X_2*.

Now, the merge of X_1 and X_2 is nothing but the merge of A_1 and B_1, followed by the merge of A_2 and B_2, and so on. That is, merging X_1 and X_2 reduces to merging A_i with B_i for $i = 1, 2, \ldots, M$. We know that the size of each A_i is $\log m$. But the sizes of the B_i's could be very large (or very small). How can we merge A_i with B_i? We can use the idea of partitioning one more time.

Let A_i and B_i be an arbitrary pair. If $|B_i| = O(\log m)$, they can be merged in $O(\log m)$ time using one processor. Consider the case when $|B_i|$ is $\omega(\log m)$. Partition B_i into $\lceil \frac{|B_i|}{\log m} \rceil$ parts, where each part has at most $\log m$

successive keys of B_i. Allocate one processor to each part so that the processor can find the corresponding subset of this part in A_i in $O(\log\log m)$ time. As a result, the problem of merging A_i and B_i has been reduced to $\lceil \frac{|B_i|}{\log m} \rceil$ subproblems, where each subproblem is that of merging two sequences of length $O(\log m)$.

The number of processors used is $\sum_{i=1}^{M} \lceil \frac{|B_i|}{\log m} \rceil$, which is $\leq 2M$. Thus we conclude the following.

Theorem 13.13 Two sorted sequences of length m each can be merged in $O(\log m)$ time using $\frac{2m}{\log m}$ CREW PRAM processors. □

13.5.4 An $O(\log\log m)$-Time Algorithm

Now we present a very fast algorithm for merging. This algorithm can merge two sorted sequences in $O(\log\log m)$ time, where m is the number of elements in each of the two sequences. The number of processors used is $2m$. The basic idea behind this algorithm is the same as the one used for the algorithm of Theorem 13.13. In addition we employ the divide-and-conquer technique.

X_1 and X_2 are the given sequences. Assume that the keys are distinct. The algorithm reduces the problem of merging X_1 and X_2 into $N \leq 2\sqrt{m}$ subproblems, where each subproblem is that of merging two sorted sequences of length $O(\sqrt{m})$. This reduction is completed in $O(1)$ time using m processors. If $T(m)$ is the run time of the algorithm using $2m$ processors, then $T(m)$ satisfies the recurrence relation $T(m) = T(O(\sqrt{m})) + O(1)$ whose solution is $O(\log\log m)$. Details of the algorithm are given in Algorithm 13.11.

The correctness of Algorithm 13.11 is quite clear; we infer this theorem.

Theorem 13.14 Two sorted sequences of length m each can be merged in $O(\log\log m)$ time using $2m$ CREW PRAM processors. □

The above algorithm has a speedup of $\frac{\Theta(m)}{\Theta(\log\log m)} = \Theta(m/\log\log m)$ which is very close to m. Its efficiency is $\Theta(1/\log\log m)$, and hence the algorithm is not work-optimal!

EXERCISES

1. Modify Algorithm 13.11 so that it uses only $\frac{m}{\log\log m}$ CREW PRAM processors and merges X_1 and X_2 in $O(\log\log m)$ time.

2. A sequence $K = k_1, k_2, \ldots, k_n$ is said to be *bitonic* either (1) if there is a $1 \leq j \leq n$ such that $k_1 \leq k_2 \leq \cdots k_j \geq k_{j+1} \geq \cdots \geq k_n$ or

---

Step 1. Partition X_1 into $\sqrt{m}$ parts with $\sqrt{m}$ elements each. Call these parts $A_1, A_2, \ldots, A_{\sqrt{m}}$. Let the largest key in A_i be ℓ_i (for $i = 1, 2, \ldots, \sqrt{m}$). Assign $\sqrt{m}$ processors to each of these ℓ_i's. The processors associated with ℓ_i perform a $\sqrt{m}$-ary search on X_2 to find the correct (i.e., sorted) position of ℓ_i in X_2 in $O(1)$ time (see Section 13.4, Exercise 10). This induces a partitioning of X_2 into $\sqrt{m}$ parts. Note that some of these parts could be empty (see Figure 13.10). Let the corresponding parts of X_2 be $B_1, B_2, \ldots, B_{\sqrt{m}}$. The subset B_i is the corresponding subset of A_i in X_2.

Step 2. Now, the merge of X_1 and X_2 is nothing but the merge of A_1 and B_1, followed by the merge of A_2 and B_2, and so on. That is, merging X_1 and X_2 reduces to merging A_i with B_i for $i = 1, 2, \ldots, \sqrt{m}$. We know that the size of each A_i is $\sqrt{m}$. But the sizes of the B_i's could be very large (or very small). To merge A_i with B_i, we can use the idea of partitioning one more time.

Let A_i and B_i be an arbitrary pair. If $|B_i| = O(\sqrt{m})$, we can merge them in $O(1)$ time using an m^ϵ-ary search. Consider the case when $|B_i|$ is $\omega(\sqrt{m})$. Partition B_i into $\lceil \frac{|B_i|}{\sqrt{m}} \rceil$ parts, where each part has at most $\sqrt{m}$ successive keys of B_i. Allocate $\sqrt{m}$ processors to each part so that the processors can find the corresponding subset of this part in A_i in $O(1)$ time. As a result the problem of merging A_i and B_i has been reduced to $\lceil \frac{|B_i|}{\sqrt{m}} \rceil$ subproblems, where each subproblem is that of merging two sequences of length $O(\sqrt{m})$.

The number of processors used is $\sum_{i=1}^{\sqrt{m}} \sqrt{m} \lceil \frac{|B_i|}{\sqrt{m}} \rceil$, which is $\leq 2m$.

Algorithm 13.11 Merging in $O(\log \log m)$ time

(2) a cyclic shift of K satisfies 1. For example, $3, 8, 12, 17, 24, 15, 9, 6$ and $21, 35, 19, 16, 8, 5, 1, 15, 17$ are bitonic. If K is a bitonic sequence with n elements (for n even), let $a_i = \min \{k_i, k_{i+n/2}\}$ and $b_i = \max \{k_i, k_{i+n/2}\}$. Also let $L(K) = \min \{k_1, k_{1+n/2}\}$, $\min \{k_2, k_{2+n/2}\}$, $\ldots, \min \{k_{n/2}, k_n\}$ and $H(K) = \max \{k_1, k_{1+n/2}\}$, $\max \{k_2, k_{2+n/2}\}, \ldots,$ $\max \{k_{n/2}, k_n\}$. Show that:

(a) $L(K)$ and $H(K)$ are both bitonic.

(b) Every element of $L(K)$ is smaller than any element of $H(K)$. In other words, to sort K, it suffices to sort $L(K)$ and $H(K)$ separately and output one followed by the other.

The above properties suggest a divide-and-conquer algorithm for sorting a given bitonic sequence. Present the details of this algorithm together with an analysis of the time and processor bounds. Show how to make use of the resultant sorting algorithm to merge two given sorted sequences. Such an algorithm is called the *bitonic merger*.

3. Given two sorted sequences of length n each. How will you merge them in $O(1)$ time using n^2 CREW PRAM processors?

13.6 SORTING

Given a sequence of n keys, recall that the problem of sorting is to rearrange this sequence into either ascending or descending order. In this section we study several algorithms for parallel sorting. If we have n^2 processors, the rank of each key can be computed in $O(\log n)$ time comparing, in parallel, all possible pairs (see the proof of Theorem 13.7). Once we know the rank of each key, in one parallel write step they can be written in sorted order (the key whose rank is i is written in cell i). Thus we have the following theorem.

Theorem 13.15 We can sort n keys in $O(\log n)$ time using n^2 CREW PRAM processors. □

The work done by the preceding algorithm is $O(n^2 \log n)$. On the other hand we have seen several sequential algorithms with run times of $O(n \log n)$ (Chapter 3) and have also proved a matching lower bound (Chapter 10). The preceding algorithm is not work-optimal.

13.6.1 Odd-Even Merge Sort

Odd-even merge sort employs the classical divide-and-conquer strategy. Assume for simplicity that n is an integral power of two and that the keys are distinct. If $X = k_1, k_2, \ldots, k_n$ is the given sequence of n keys, it is partitioned into two subsequences $X'_1 = k_1, k_2, \ldots, k_{n/2}$ and $X'_2 = k_{n/2+1}, k_{n/2+2}, \ldots, k_n$ of equal length. X'_1 and X'_2 are sorted recursively assigning $n/2$ processors to each. The two sorted subsequences (call them X_1 and X_2, respectively) are then finally merged.

The preceding description of the algorithm is exactly the same as that of merge sort. The difference between the two algorithms lies in how the two subsequences X_1 and X_2 are merged. In the merging algorithm used in Section 3.4, the minimum elements from the two sequences are compared and the minimum of these two is output. This step continues until the two sequences are merged. As is seen, this process seems to be inherently sequential in nature. Instead, we employ the odd-even merge algorithm (Algorithm 13.10) of Section 13.5.

Theorem 13.16 We can sort n arbitrary keys in $O(\log^2 n)$ time using n EREW PRAM processors.

Proof: The sorting algorithm is described in Algorithm 13.12. It uses n processors. Define $T(n)$ to be the time taken by this algorithm to sort n keys using n processors. Step 1 of this algorithm takes $O(1)$ time. Step 2 runs in $T(n/2)$ time. Finally, step 3 takes $O(\log n)$ time (c.f. Theorem 13.10). Therefore, $T(n)$ satisfies $T(n) = O(1) + T(n/2) + O(\log n) = T(n/2) + O(\log n)$, which solves to $T(n) = O(\log^2 n)$. □

Example 13.20 Consider the problem of sorting the 16 numbers 25, 21, 8, 5, 2, 13, 11, 16,23, 31, 9, 4,18, 12, 27, 34 using 16 processors. In step 1 of Algorithm 13.12, the input is partitioned into two: $X'_1 = 8, 21, 8, 5, 2, 13, 11, 16$, and $X'_2 = 23, 31, 9, 4, 18, 12, 27, 34$. In step 2, processors 1 to 8 work on X'_1, recursively sort it, and obtain $X_1 = 2, 5, 8, 11, 13, 16, 21, 25$. At the same time processors 9 to 16 work on X'_2, sort it, and obtain $X_2 = 4, 9, 12, 18, 23, 27, 31,$ 34. In step 3, X_1 and X_2 are merged as shown in Example 13.18 to get the final result: 2, 4, 5, 8,9, 11, 12, 13,16, 18, 21, 23,25, 27, 31, 34. □

The work done by Algorithm 13.12 is $\Theta(n \log^2 n)$. Therefore, its efficiency is $\Theta(1/\log n)$. It has a speedup of $\Theta(n/\log n)$.

13.6.2 An Alternative Randomized Algorithm

We can get the result of Theorem 13.16 using the randomized selection algorithm of Section 13.4. Theorem 13.8 states that selection from out of n keys can be performed in $\widetilde{O}(\log n)$ time using $\frac{n}{\log n}$ processors. Assume that there are n processors. The median k of the n given keys can be found in $\widetilde{O}(\log n)$ time. Having found the median, partition the input into two parts.

Step 0. If $n \leq 1$, return X.

Step 1. Let $X = k_1, k_2, \ldots, k_n$ be the input. Partition the input into two: $X_1' = k_1, k_2, \ldots, k_{n/2}$ and $X_2' = k_{n/2+1}, k_{n/2+2}, \ldots, k_n$.

Step 2. Allocate $n/2$ processors to sort X_1' recursively. Let X_1 be the result. At the same time employ the other $n/2$ processors to sort X_2' recursively. Let X_2 be the result.

Step 3. Merge X_1 and X_2 using Algorithm 13.10 and $n = 2m$ processors.

Algorithm 13.12 Odd-even merge sort

The first part X_1' contains all the input keys $\leq k$ and the second part X_2' contains all the rest of the keys. The parts X_1' and X_2' are sorted recursively with $n/2$ processors each. The output is X_1' in sorted order followed by X_2' in sorted order. If $T(n)$ is the sorting time of n keys using n processors, we have $T(n) = T(n/2) + \widetilde{O}(\log n)$, which solves to $T(n) = \widetilde{O}(\log^2 n)$.

Theorem 13.17 Sorting n keys can be performed in $\widetilde{O}(\log^2 n)$ time using n CREW PRAM processors. □

13.6.3 Preparata's Algorithm

Preparata's algorithm runs in $O(\log n)$ time and uses $n \log n$ CREW PRAM processors. This is a recursive divide-and-conquer algorithm wherein the rank of each input key is computed and the keys are output according to their ranks (see the proof of Theorem 13.15). Let $k_1, k_2, \ldots, k_n$ be the input sequence. Preparata's algorithm partitions the input into $\log n$ parts $K_1, K_2, \ldots, K_{\log n}$, where there are $\frac{n}{\log n}$ keys in each part. If k is any key in the input, its rank in the input is computed as follows. First, the rank r_i of k in K_i is computed for each i, $1 \leq i \leq \log n$. Then, the total rank of k is computed as $\sum_{i=1}^{\log n} r_i$. One of the results that it makes use of is Theorem 13.14.

The details of Preparata's algorithm are given in Algorithm 13.13. Let $T(n)$ be the run time of Preparata's algorithm using $n \log n$ processors. Clearly, step 1 takes $T(n/\log n)$ time and steps 2 and 3 together take

Step 0. If n is a small constant, sort the keys using any algorithm and quit.

Step 1. Partition the given n keys into $\log n$ parts, with $n/\log n$ keys in each part. Sort each part recursively and separately in parallel, assigning n processors to each part. Let $S_1, S_2, \ldots, S_{\log n}$ be the sorted sequences.

Step 2. Merge S_i with S_j for $1 \leq i, j \leq \log n$ in parallel. This can be done by allocating $n/\log n$ processors to each pair (i, j). That is, using $n \log n$ processors, this step can be accomplished in $O(\log\log n)$ time with Algorithm 13.11. As a by-product of this merging step, we have computed the rank of each key in each one of the S_i's $(1 \leq i \leq \log n)$.

Step 3. Allocate $\log n$ processors to compute the rank of each key in the original input. This is done in parallel for all the keys by adding the $\log n$ ranks computed (for each key) in step 2. This can be done in $O(\log\log n)$ time using the prefix computation algorithm (see Algorithm 13.3). Finally, the keys are written out in the order of their ranks.

Algorithm 13.13 Preparata's sorting algorithm

$O(\log\log n)$ time. Thus we have

$$T(n) \;=\; T(n/\log n) \;+\; O(\log\log n)$$

which can be solved by repeated substitution to get $T(n) = O(\log n)$. Also, the number of processors used in each step is $n\log n$. We get the following.

Theorem 13.18 Any n arbitrary keys can be sorted in $O(\log n)$ time using $n\log n$ CREW PRAM processors. □

Applying the slow-down lemma (Lemma 13.2) to the above theorem, we infer a corollary.

Corollary 13.1 Any n general keys can be sorted in $O(t\log n)$ time using $n\log n/t$ CREW PRAM processors, for any $t \geq 1$. □

Preparata's algorithm does the same total work as the odd-even merge sort. But its speedup is $\Theta(n)$, which is better than that of the odd-even merge sort. Efficiency of both the algorithms is the same; i.e., $\Theta(1/\log n)$.

13.6.4 Reischuk's Randomized Algorithm (*)

This algorithm uses n processors and runs in time $\widetilde{O}(\log n)$. Thus its efficiency is $\frac{\Theta(n\log n)}{\widetilde{\Theta}(n\log n)} = \widetilde{\Theta}(1)$; i.e., the algorithm is work-optimal with high probability! The basis for this algorithm is Preparata's sorting scheme and the following theorem. (For a proof see the references at the end of this chapter.)

Theorem 13.19 We can sort n keys, where each key is an integer in the range $[0, n(\log n)^c]$ (c is any constant) in $\widetilde{O}(\log n)$ time using $\frac{n}{\log n}$ CRCW PRAM processors. □

Reischuk's algorithm runs in the same time bound as Preparata's (with high probability) but uses only n processors. The idea is to randomly sample $N = \frac{n}{\log^4 n}$ keys from the input and sort these using a non-work-optimal algorithm like Preparata's. The sorted sample partitions the original problem into $N+1$ independent subproblems of nearly equal size, and all these subproblems can be solved easily. These ideas are made concrete in Algorithm 13.14.

Step 2 of Algorithm 13.14 can be done using $N\log N \leq N\log n$ processors in $O(\log N) = O(\log n)$ time (c.f. Theorem 13.18). In step 3, the partitioning of X can be done using binary search and the integer sort algorithms

---

Step 1. $N = n/(\log^4 n)$ processors randomly sample a key (each) from $X = k_1, k_2, \ldots, k_n$, the given input sequence.

Step 2. Sort the N keys sampled in step 1 using Preparata's algorithm. Let $l_1, l_2, \ldots, l_N$ be the sorted sequence.

Step 3. Let $K_1 = \{k \in X | k \leq l_1\}$; $K_i = \{k \in X | l_{i-1} < k \leq l_i\}$, $i = 2, 3, \ldots, N$; and $K_{N+1} = \{k \in X | k > l_N\}$. Partition the given input X into K_i's as defined. This is done by first finding the part each key belongs to (using binary search in parallel). Now partitioning the keys reduces to sorting the keys according to their part numbers.

Step 4. For $1 \leq i \leq N+1$ in parallel sort K_i using Preparata's algorithm.

Step 5. Output sorted(K_1), sorted(K_2), $\ldots$, sorted(K_{N+1}).

Algorithm 13.14 Work-optimal randomized algorithm for sorting

(c.f. Theorem 13.19). If there is a processor associated with each key, the processor can perform a binary search in $l_1, l_2, \ldots, l_N$ to figure out the part number the key belongs to. Note that the part number of each key is an integer in the range $[1, N+1]$. Therefore the keys can be sorted according to their part numbers using Theorem 13.19.

Thus step 3 can be performed in $\widetilde{O}(\log n)$ time, using $\leq n$ processors. With high probability, there will be no more than $O(\log^5 n)$ keys in each of the K_i's $(1 \leq i \leq N)$. The proof of this fact is left as an exercise. Within the same processor and time bounds, we can also count $|K_i|$ for each i. In step 4, each K_i can be sorted in $O(\log |K_i|)$ time using $|K_i| \log |K_i|$ processors. Also K_i can be sorted in $(\log |K_i|)^2$ time using $|K_i|$ processors (see Corollary 13.1). So step 4 can be completed in $(\max_i \log |K_i|)^2$ time using n processors. If $\max_i |K_i| = O(\log^5 n)$, step 4 takes $O((\log \log n)^2)$ time. Thus we have proved the following.

Theorem 13.20 We can sort n general keys using n CRCW PRAM processors in $\widetilde{O}(\log n)$ time. □

EXERCISES

1. In step 3 of Algorithm 13.12, we could employ the merging algorithm of Algorithm 13.11. If so, what would be the run time of Algorithm 13.12? What would be the processor bound?

2. If we have n numbers to sort and each number is a bit, one way of sorting X could be to make use of prefix computation algorithms as in Section 13.3, Exercise 8. This amounts to counting the number of zeros and the number of ones. If z is the number of zeros, we output z zeros followed by $n - z$ ones. Using this idea, design an $O(\log n)$ time algorithm to sort n numbers, where each number is an integer in the range $[0, \log n - 1]$. Your algorithm should run in $O(\log n)$ time using no more than $\frac{n}{\log n}$ CREW PRAM processors. Recall that n numbers in the range $[0, n^c]$ can be sequentially sorted in $O(n)$ time (the corresponding algorithm is known as the radix sort).

3. Make use of the algorithm designed in the previous problem together with the idea of radix sorting to show that n numbers in the range $[0, (\log n)^c]$ can be sorted in $O(\log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors.

4. Given two sets A and B of size n each (in the form of arrays), the goal is to check whether the two sets are disjoint or not. Show how to solve this problem:

   (a) In $O(1)$ time using n^2 CRCW PRAM processors
   (b) In $\widetilde{O}(\log n)$ time using n CREW PRAM processors

5. Sets A and B are given such that $|A| = n$, $|B| = m$, and $n \geq m$. Show that we can determine whether A and B are disjoint in $\widetilde{O}((\log n)(\log m))$ time using $\frac{n}{\log n}$ CREW PRAM processors.

6. Show that if a set X of n keys is partitioned using a random sample of size s (as in Reischuk's algorithm), the size of each part is $\widetilde{O}\left(\frac{n}{s}\log n\right)$.

7. Array A is an almost sorted array of n elements. It is given that the position of each key is at most a distance of d from its final sorted position, where d is a constant. Give an $O(1)$ time n-processor EREW PRAM algorithm to sort A. Prove the correctness of your algorithm using the zero-one principle.

8. The original algorithm of Reischuk was recursive and had the following steps:

   (a) Select a random sample of size $\sqrt{n}-1$ and sort it using Theorem 13.15.
   (b) Partition the input into $\sqrt{n}$ parts making use of the sorted sample (similar to step 3 of Algorithm 13.14).
   (c) Assign a linear number of processors to each part and recursively sort each part in parallel.
   (d) Output the sorted parts in the correct order.

   See if you can analyze the run time of this algorithm.

9. It is known that prefix sums computation can be done in time $O(\frac{\log n}{\log\log n})$ using $\frac{n \log\log n}{\log n}$ CRCW PRAM processors, provided the numbers are integers in the range $[0, n^c]$ for any constant c. Assuming this result, show that sorting can be done in time $O(\frac{\log n}{\log\log n})$ time using n^2 CRCW PRAM processors.

10. Adopt the algorithms of Exercise 7 and Section 13.4, Exercise 10, and the $O(\log n/\log\log n)$ time algorithm for integer prefix sums computation to show that n numbers can be sorted in $\widetilde{O}(\frac{\log n}{\log\log n})$ time using $n(\log n)^\epsilon$ CRCW PRAM processors (for any constant $\epsilon > 0$).

11. The *random access read (RAR)* operation in a parallel machine is defined as follows: Each processor wants to read a data item from some other processor. In the case of a PRAM, it is helpful to assume that

each processor has an associated part of the global memory, and reading from a processor means reading from the corresponding part of shared memory. It may be the case that several processors want to read from the same processor. Note that on the CRCW PRAM or on the CREW PRAM, a RAR operation can be performed in one unit of time. Devise an efficient algorithm for RAR on the EREW PRAM. (*Hint:* If processor i wants to read from processor j, create a tuple (j, i) corresponding to this request. Sort all the tuples (j, i) in lexicographic order.)

12. We can define a *random access write (RAW)* operation similar to RAR as follows. Every processor has an item of data to be sent to some other processor (that is, an item of data to be written in the shared memory part of some other processor). Several processors might want to write in the same part and hence a resolution scheme (such as common, priority, etc.) is also supplied. On the CRCW PRAM (with the same resolution scheme), this can be done in one unit of time. Develop efficient algorithms for RAW on the CREW and EREW PRAMs. (*Hint:* Make use of sorting (see Exercise 11).)

13.7 GRAPH PROBLEMS

We consider the problems of transitive closure, connected components, minimum spanning tree, and all-pairs shortest paths in this section. Efficient sequential algorithms for these problems were studied in Chapters 6 and 10. We begin by introducing a general framework for solving these problems.

Definition 13.4 Let M be an $n \times n$ matrix with nonnegative integer coefficients. Let $\widetilde{M}$ be a matrix defined as follows:

$$\widetilde{M}(i,i) = 0 \qquad \text{for every } i$$

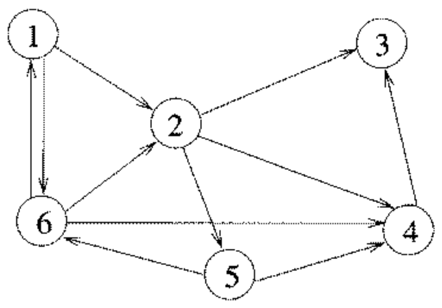
$$\widetilde{M}(i,j) = \min \{M_{i_0 i_1} + M_{i_1 i_2} + \cdots + M_{i_{k-1} i_k}\} \qquad \text{for every } i \neq j$$

where $i_0 = i$, $i_k = j$, and the minimum is taken over all sequences $i_0, i_1, \ldots, i_k$ of elements from the set $\{1, 2, \ldots, n\}$. □

Example 13.21 Let $G(V, E)$ be a directed graph with $V = \{1, 2, \ldots, n\}$. Define M as $M(i,j) = 0$ if either $i = j$ or there is a directed edge from node i to node j in G, and $M(i,j) = 1$ otherwise. For this choice of M, it is easy to see that $\widetilde{M}(i,j) = 0$ if and only if there is a directed path from node i to node j in G.

In Figure 13.11, a directed graph is shown together with its M and $\widetilde{M}$. $\widetilde{M}(1,5)$ is zero since $M_{12} + M_{25} = 0$. Similarly, $\widetilde{M}(2,1)$ is zero since $M_{25} +$

$M_{56} + M_{61} = 0$. On the other hand, $\widetilde{M}(3,1)$ is one since for every choice of $i_1, i_2, \ldots, i_{k-1}$, the sum $M_{i_0 i_1} + M_{i_1 i_2} + \cdots + M_{i_{k-1} i_k}$ is > 0. □



M

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

$\widetilde{M}$

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

Figure 13.11 An example graph and its M and $\widetilde{M}$

Theorem 13.21 $\widetilde{M}$ can be computed from an $n \times n$ matrix M in $O(\log n)$ time using $n^{3+\epsilon}$ common CRCW PRAM processors, for any fixed $\epsilon > 0$.

Proof: We make use of $O(n^3)$ global memory. In particular we use the variables $m[i,j]$ for $1 \leq i,j \leq n$ and $q[i,j,k]$ for $1 \leq i,j,k \leq n$. The algorithm to be employed is given in Algorithm 13.15.

Initializing $m[\,]$ takes n^2 time. Step 1 of Algorithm 13.15 takes $O(1)$ time using n^3 processors. In step 2, n^2 different $m[i,j]$'s are computed. The computation of a single $m[i,j]$ involves computing the minimum of n numbers and hence can be completed in $O(1)$ time using n^2 CRCW PRAM processors (Theorem 13.4). In fact this minimum can also be computed in $O(1)$ time using $n^{1+\epsilon}$ processors for any fixed $\epsilon > 0$ (Section 13.4, Exercise 4). In summary, step 2 can be completed in $O(1)$ time using $n^{3+\epsilon}$ common

```
$m[i,j] := M[i,j]$ for $1 \leq i,j \leq n$ in parallel;
for $r := 1$ to $\log n$ do
{
    Step 1. In parallel set $q[i,j,k] := m[i,j] + m[j,k]$ for
    $1 \leq i,j,k \leq n$.

    Step 2. In parallel set $m[i,j] :=$ min $\{q[i,1,j],$
    $q[i,2,j], \ldots, q[i,n,j]\}$ for $1 \leq i,j \leq n$.
}

Put $\widetilde{M}(i)(i) := 0$ for all $i$ and $\widetilde{M}(i)(j) := m[i,j]$ for $i \neq j$.
```

Algorithm 13.15 Computation of $\widetilde{M}$

CRCW PRAM processors. Thus the **for** loop runs in $O(\log n)$ time. The final computation of $\widetilde{M}$ also can be done in $O(1)$ time using n^2 processors. □

The correctness of Algorithm 13.15 can be proven by induction on r. We can show that the value of $m[i,j]$ at the end of the rth iteration of the **for** loop is min $\{M_{i_0 i_1} + M_{i_1 i_2} + \cdots + M_{i_{k-1} i_k}\}$, where $i = i_0$, $j = i_k$, and the minimum is taken over all the sequences $i_0, i_1, \ldots, i_k$ of elements of $\{1, 2, \ldots, n\}$ such that $k \leq 2^r$. Algorithm 13.15 can be specialized to solve several problems including the transitive closure, connected components, minimum spanning tree, and so on.

Theorem 13.22 The transitive closure matrix of an n-vertex directed graph can be computed in $O(\log n)$ time using $n^{3+\epsilon}$ common CRCW PRAM processors.

Proof: If M is defined as in Example 13.21, the transitive closure of G can be easily obtained once $\widetilde{M}$ is computed. In accordance with Theorem 13.21, $\widetilde{M}$ can be computed within the stated resource bounds. □

Theorem 13.23 The connected components of an n-vertex graph can be determined in $O(\log n)$ time using $n^{3+\epsilon}$ common CRCW PRAM processors.

Proof: Define $M(i)(j)$ to be zero if either $i = j$ or i and j are connected by an edge; $M(i)(j)$ is one otherwise. Nodes i and j are in the same connected component if and only if $\widetilde{M}(i)(j) = 0$. □

Theorem 13.24 A minimum spanning tree for an n-vertex weighted graph $G(V, E)$ can be computed in $O(\log n)$ time using $n^{5+\epsilon}$ common CRCW PRAM processors.

Proof: The algorithm is a parallelization of Kruskal's sequential algorithm (see Section 4.5). In Kruskal's algorithm, the edges in the given graph G are sorted according to nondecreasing edge weights. A forest F of trees is maintained. To begin with, F consists of $|V|$ isolated nodes. The edges are processed from the smallest to the largest. An edge (u, v) gets included in F if and only if (u, v) connects two different trees in F.

In parallel, the edges can be sorted in $O(\log n)$ time using n^2 processors (Theorem 13.15). Let $e_1, e_2, \ldots, e_n$ be the edges of G. For each edge $e_i = (u, v)$, we can decide, in parallel, whether it will belong to the final tree as follows. Find the transitive closure of the graph G_i that has V as its node set and whose edges are $e_1, e_2, \ldots, e_{i-1}$. The e_i will get included in the final spanning tree if and only if u and v are not in the same connected component of G_i.

Thus, using Theorem 13.23, the test as to whether an edge belongs to the final answer can be performed in $O(\log n)$ time given $n^{3+\epsilon}$ processors. Since there are at most n^2 edges, the result follows. □

13.7.1 An Alternative Algorithm for Transitive Closure

Now we show how to compute the transitive closure of a given directed graph $G(V, E)$ in $O(\log^2 n)$ time using $\frac{n^3}{\log n}$ CREW PRAM processors. In Section 10.3 (Lemma 10.9) we showed that if A is the adjacency matrix of G, then the transitive closure matrix M is given by $M = I + A + A^2 + \cdots + A^{n-1}$. Along the same lines, we can also show that $M = (I + A)^n$. A proof by induction will establish that for any k, $1 \le k \le n$, $(I + A)^k(i)(j) = 1$ if and only if there is a directed path from node i to node j of length $\le k$.

Thus M can be computed by evaluating $(I + A)^n$. $(I + A)^n$ can be rewritten as $(I + A)^{2^{\lceil \log n \rceil}}$. Therefore, computing M reduces to a sequence of $\lceil \log n \rceil$ matrix squarings (or multiplications). Since two matrices can be multiplied in $O(\log n)$ time using $\frac{n^3}{\log n}$ CREW PRAM processors (see Section 13.3, Exercise 12), we have the following theorem.

Theorem 13.25 The transitive closure of an n-node directed graph can be computed in $O(\log^2 n)$ time using $\frac{n^3}{\log n}$ CREW PRAM processors. □

13.7.2 All-Pairs Shortest Paths

An $O(n^3)$ time algorithm was developed in Section 5.3 for the problem of identifying the shortest path between every pair of vertices in a given weighted directed graph. The basic principle behind this algorithm was to define $A^k(i,j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k and then to infer that

$$A^k(i,j) = \min\ \{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\}, \quad k \geq 1.$$

The same paradigm can be used to design a parallel algorithm as well. The importance of the above relationship between A^k and A^{k-1} is that the computation of A^k from A^{k-1} corresponds to matrix multiplication, where min and addition take the place of addition and multiplication, respectively. Under this interpretation of matrix multiplication, the problem of all-pairs shortest paths reduces to computing $A^n = A^{2^{\lceil \log n \rceil}}$. We get this theorem.

Theorem 13.26 The all-pairs shortest-paths problem can be solved in $O(\log^2 n)$ time using $\frac{n^3}{\log n}$ CREW PRAM processors. □

EXERCISES

1. Compute the speedup, total work done, and efficiency for each of the algorithms given in this section.

2. Let $G(V, E)$ be a directed acyclic graph (dag). The *topological sort* of G is defined to be a linear ordering of the vertices of G such that if (u, v) is an edge of G, then u appears before v in the linear ordering. Show how to employ the general paradigm introduced in this section to obtain an $O(\log n)$ time algorithm for topological sort using $n^{3+\epsilon}$ common CRCW PRAM processors.

3. Present an efficient parallelization of Prim's algorithm for minimum spanning trees (see Section 4.5).

4. Present an efficient parallel algorithm to check whether a given undirected graph is acyclic. Analyze the processor and time bounds.

5. If G is any undirected graph, G^k is defined as follows: There will be an edge between nodes i and j in G^k if and only if there is a path of length k in G between i and j. Present an $O(\log n \log k)$ time algorithm to compute G^k from G. You can use a maximum of $\frac{n^3}{\log n}$ CREW PRAM processors.

6. Present an efficient parallel minimum spanning tree algorithm for the special case when the edge weights are zero and one.

7. Present an efficient parallelization of the Bellman and Ford algorithm (see Section 5.4).

13.8 COMPUTING THE CONVEX HULL

In this section we revisit the problem of constructing the convex hull of n points in 2D in clockwise order. The technique to be used is the same as the one we employed sequentially. The parallel algorithm will have a run time of $O(\log n)$ using n CREW PRAM processors. Note that in Chapter 10 we proved a lower bound of $\Omega(n \log n)$ for the convex hull problem and hence the parallel algorithm to be studied is work-optimal.

The sequential algorithm was based on divide-and-conquer (see Section 3.8.4). It computed the upper hull and the lower hull of the given point set separately. Thus let us restrict our discussion to the computation of the upper hull only. The given points were partitioned into two halves on the basis of their x-coordinate values. All points with an x-coordinate $\leq$ the median formed the first part. The rest of the points belonged to the second part. Upper hulls were recursively computed for the two halves. These two hulls were then merged by finding the line of tangent.

We adopt a similar technique in parallel. First, the points with the minimum and maximum x-coordinate values are identified. This can be done using the prefix computation algorithm in $O(\log n)$ time and $\frac{n}{\log n}$ processors. Let p_1 and p_2 be these points. All the points which are to the left of the line segment $\langle p_1, p_2 \rangle$ are separated from those which are to the right. This separation also can be done using a prefix computation. Points of the first (second) kind contribute to the upper (lower) hull. The computations of the upper hull and the lower hull are done independently. From here on we only consider the computation of the upper hull. By "input" we mean all the points that are to the left of $\langle p_1, p_2 \rangle$. We denote the number of such points by N.

Sort the input points according to their x-coordinate values. This can be done in $\widetilde{O}(\log N)$ time using N processors. In fact there are deterministic algorithms with the same time and processor bounds as well (see the references at the end of this chapter). This sorting is done only once in the computation of the upper hull. Let $q_1, q_2, \ldots, q_N$ be the sorted order of these points. The recursive algorithm for computing the upper hull is given in Algorithm 13.16. An upper hull is maintained in clockwise order as a list. We refer to the first element in the list as the leftmost point and the last element as the rightmost point.

We show that step 3 can be performed in $O(1)$ time using N processors. Step 4 also can be completed in $O(1)$ time. If $T(N)$ is the run time of Algorithm 13.16 for finding the upper hull on an input of N points using N

Step 0. If $N \le 2$, solve the problem directly.

Step 1. Partition the input into two halves with $q_1, q_2, \ldots, q_{N/2}$ in the first half and $q_{N/2+1}, q_{N/2+2}, \ldots, q_N$ in the second half.

Step 2. Compute the upper hull of each half (in clockwise order) recursively assigning $\frac{N}{2}$ processors to each half. Let H_1 and H_2 be the upper hulls.

Step 3. Find the line of tangent (see Figure 3.9) between the two upper hulls. Let $\langle u, v\rangle$ be the tangent.

Step 4. Drop all the points of H_1 that are to the right of u. Similarly, drop all the points to the left of v in H_2. The remaining part of H_1, the tangent, and the remaining part of H_2 form the upper hull of the given input set.

Algorithm 13.16 Parallel convex hull algorithm

processors, then we have

$$T(N) = T(N/2) + O(1)$$

which solves to $T(N) = O(\log N)$. The number of processors used is N.

The only part of the algorithm that remains to be specified is how to find the tangent $\langle u, v\rangle$ in $O(1)$ time using N processors. First start from the middle point p of H_1. Here the middle point refers to the middle element of the corresponding list. Find the tangent of p with H_2. Let $\langle p, q\rangle$ be the tangent. Using $\langle p, q\rangle$, we can determine whether u is to the left of, equal to, or to the right of p in H_1. A k-ary search (for some suitable k) in this fashion on the points of H_1 will reveal u. Use the same procedure to isolate v.

Lemma 13.4 Let H_1 and H_2 be two upper hulls with at most m points each. If p is any point of H_1, its tangent q with H_2 can be found in $O(1)$ time using m^ϵ processors for any fixed $\epsilon > 0$.

Proof. If q' is any point in H_2, we can check whether q' is to the left of, equal to, or to the right of q in $O(1)$ time using a single processor (see Figure 3.10). If $\angle pq'x$ is a right turn and $\angle pq'y$ is a left turn, then q is to the right of q'; if $\angle pq'x$ and $\angle pq'y$ are both right turns, then, $q' = q$; otherwise q is to the left of q'. Thus if we have m processors, we can assign one processor to each point of H_2 and identify q in $O(1)$ time. The identification of q can also be done using an m^ϵ-ary search (see Section 13.4, Exercise 10) in $O(1)$ time and m^ϵ processors, for any fixed $\epsilon > 0$. □

Lemma 13.5 If H_1 and H_2 are two upper hulls with at most m points each, their common tangent can be computed in $O(1)$ time using m processors.

Proof. Let $u \in H_1$ and $v \in H_2$ be such that $\langle u, v\rangle$ is the line of tangent. Also let p be an arbitrary point of H_1 and let $q \in H_2$ be such that $\langle p, q\rangle$ is a tangent of H_2. Given p and q, we can check in $O(1)$ time whether u is to the left of, equal to, or to the right of p (see Figure 3.11). If $\langle p, q\rangle$ is also tangential to H_1, then $p = u$. If $\angle xpq$ is a left turn, then u is to the left of p; else u is to the right of p. This suggests an m^ϵ-ary search for u. For each point p of H_1 chosen, we have to determine the tangent from p to H_2 and then decide the relative positioning of p with respect to u. Thus indeed we need $m^\epsilon \times m^\epsilon$ processors to determine u in $O(1)$ time. If we choose $\epsilon = 1/2$, we can make use of all the m processors. □

The following theorem summarizes these findings.

Theorem 13.27 The convex hull of n points in the plane can be computed in $O(\log n)$ time using n CREW PRAM processors. □

Algorithm 13.16 has a speedup of $\Theta(n)$; its efficiency is $\Theta(1)$.

EXERCISES

1. Show that the vertices of the convex hull of n given points can be identified in $O(1)$ time using a common CRCW PRAM.

2. Present an $O(\frac{\log n}{\log\log n})$ time CRCW PRAM algorithm for the convex hull problem. How many processors does your algorithm use?

3. Present an $O(\log n)$ time n-processor CREW PRAM algorithm to compute the area of the convex hull of n given points in 2D.

4. Given a simple polygon and a point p, the problem is to check whether p is internal to the polygon. Present an $O(\log n)$ time $\frac{n}{\log n}$-processor CREW PRAM algorithm for this problem.

5. Present an $O(1)$ time algorithm to check whether any three of n given points are colinear. You can use up to n^3 CRCW PRAM processors. Can you decrease the processor bound further?

6. Assume that n points in 2D are given in sorted order with respect to the polar angle subtended at x, where x is the point with the lowest y-coordinate. Present an $O(\log n)$-time CREW PRAM algorithm for finding the convex hull. What is the processor bound of your algorithm?

7. Given two points $p = (x_1, y_1)$ and $q = (x_2, y_2)$ in the plane, p is said to *dominate* q if $x_1 \geq x_2$ and $y_1 \geq y_2$. The *dominance counting problem* is defined as follows. Given two sets $X = \{p_1, p_2, \ldots, p_m\}$ and $Y = \{q_1, q_2, \ldots, q_n\}$ of points in the plane, determine for each point p_i the number of points in Y that are dominated by p_i. Present an $O(\log(m+n))$ time algorithm for dominance counting. How many processors does your algorithm use?

13.9 LOWER BOUNDS

In this section we present some lower bounds on parallel computation related to comparison problems such as sorting, finding the maximum, merging, and so on. The parallel model assumed is called the *parallel comparison tree (PCT)*. This model can be thought of as the parallel analog of the comparison tree model introduced in Section 10.1.

A PCT with p processors is a tree wherein at each node, at most p pairs of comparisons are made (at most one pair per processor). Depending on the outcomes of all these comparisons, the computation proceeds to an appropriate child of the node. Whereas in the sequential comparison tree each node can have at most two children, in the PCT the number of children for any node can be more than two (depending on p). The external nodes of a PCT represent termination of the algorithm. Associated with every path from the root to an external node is a unique permutation. As there are $n!$ different possible permutations of n items and any one of these might be the correct answer for the given sorting problem, the PCT must have at least $n!$ external nodes. A typical computation for a given input on a PCT proceeds as follows. We start at the root and perform p pairs of comparisons. Depending on the outcomes of these comparisons (which in turn depend on the input), we branch to an appropriate child. At this child we perform p more pairs of comparisons. And so on. This continues until we reach an external node, at which point the algorithm terminates and the correct answer is obtained from the external node reached.

Example 13.22 Figure 13.12 shows a PCT with two processors that sorts three given numbers k_1, k_2, and k_3. Rectangular nodes are external nodes that give the final answers. At the root of this PCT, two comparisons are made and hence there are four possible outcomes. There is a child for the root corresponding to each of these outcomes. For example, if both of the comparisons made at the root yielded "yes," then clearly the sorted order of the keys is k_1, k_2, k_3. On the other hand if the root comparisons yielded "yes" and "no," respectively, then k_1 is compared with k_3, and depending on the outcome, the final permutation is obtained. The depth of this PCT and hence the worst-case run time of this parallel algorithm is two. □

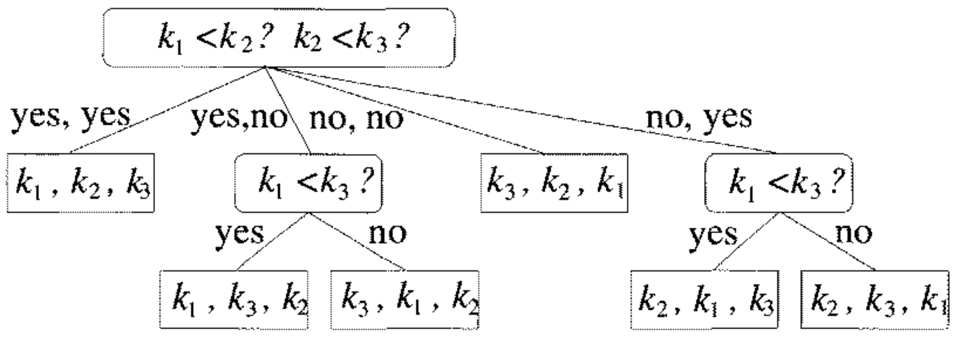


Figure 13.12 PCT with two processors that sorts three numbers

The *worst-case time* of any algorithm on a PCT is the maximum depth of any external node. The *average case time* is the average depth of an external node, all possible paths being equally likely. Note that in a PCT, while computing the time of any algorithm, we only take into account comparisons. Any other operation such as the addition of numbers, data movement, and so on, is assumed to be free. Also, at any node of the PCT, p pairs of comparisons are performed in one unit of time. As a consequence of these assumptions, any comparison problem can be solved in $O(1)$ time, given enough processors. Since a PCT is more powerful than any of the PRAMs, lower bounds derived for the PCT hold for the PRAMs as well.

Example 13.23 Suppose we are given n numbers from a linear order. There are only $\binom{n}{2}$ pairs of comparisons that can ever be made. Therefore, if $p = \binom{n}{2}$, all these comparisons can be made in one unit of time, and as a result we can solve the following problems in one unit of time: selection, sorting, and so on. (Note that a PCT charges only for the comparisons made.) □

13.9.1 A lower bound on average-case sorting

If $p \geq \binom{n}{2}$, sorting can be done in $O(1)$ time on a PCT (see Example 13.23). So assume that $p < \binom{n}{2}$. The lower bound follows from two lemmas.

The first lemma relates the average depth of an external node to the degree (i.e., the maximum number of children of any node) and the number of external nodes.

Lemma 13.6 [Shannon] A tree with degree d and ℓ external nodes has an average depth of at least $\frac{\log \ell}{\log d}$. $\square$

We can apply the above lemma to obtain a lower bound for sorting, except that we don't know what d is. Clearly, ℓ has to be at least $n!$. Note that at each node of a PCT, we make p pairs of comparisons. So, there can be as many as 2^p possible outcomes. (The first pair of comparison can yield either "yes" or "no"; independently, the second comparison can yield "yes" or "no"; and so on.) If we substitute this value for d, Lemma 13.6 yields a lower bound of $\frac{n \log n}{p}$.

A better lower bound is achieved by noting that not all of the 2^p possible outcomes at any node of the PCT are feasible. As an example, take $p \geq 3$; the three comparisons made at a given node are $x : y$, $y : z$, and $x : z$. In this case, it is impossible for the three outcomes to be "yes," "yes," and "no."

To obtain a better estimate of d, we introduce a graph. This graph has n nodes, one node per input number. Such a graph G_v is conceived of for each node v in the PCT. For each pair $x : y$ of comparisons made at the PCT node v, we draw an edge between x and y. Thus G_v has p edges and is undirected. We can *orient* (i.e., give a direction to) each edge of G_v depending on the outcome of the corresponding comparison. Say we direct the edge from x to y if $x > y$. Note that the degree of the node v is the number of ways in which we can orient the p edges of G_v.

Since the input numbers are from a linear order, any orientation of the edges of G_v that introduces a directed cycle is impossible. The question then is how many such *acyclic orientations* are possible? This number will be a better estimate of d. U. Manber and M. Tompa have proved the following.

Lemma 13.7 [Manber and Tompa] A graph with n vertices and m edges has at most $\left(1 + \frac{2m}{n}\right)^n$ acyclic orientations. $\square$

Combining Lemmas 13.6 and 13.7, we get the following theorem.

Theorem 13.28 Any PCT with p processors needs an average case time of $\Omega\left(\frac{\log n}{\log(1+p/n)}\right)$ to sort n numbers.

Proof: Using Lemma 13.7, a better estimate for d is $\left(1 + \frac{2p}{n}\right)^n$. Then, according to Lemma 13.6, the average case time for sorting is

$$\Omega\left(\frac{\log n!}{\log(1 + 2p/n)^n}\right) = \Omega\left(\frac{n \log n}{n \log(1 + 2p/n)}\right) = \Omega\left(\frac{\log n}{\log\left(1 + \frac{p}{n}\right)}\right) \quad \square$$

13.9.2 Finding the maximum

Now we prove a lower bound on the problem of identifying the maximum of n given numbers. This problem can also be solved in $O(1)$ time if we have $p \geq \binom{n}{2}$.

Theorem 13.29 [Valiant] Given n unordered elements and $p = n$ PCT processors, if MAX(n) is a lower bound on the worst-case time needed to determine the maximum value in parallel time, then MAX(n) $\geq \log\log n - c$, where c is a constant.

Proof: Consider the information determined from the set of comparisons that can be made by time t for some parallel maximum finding algorithm. Some of the elements have been shown to be smaller than other elements, and so they have been eliminated. The others form a set S which contains the correct answer. If at time t two elements not in S are compared, then no progress is made in decreasing set S. If an element in set S and one not in S are compared and the larger element is in S, then again no improvement has been made. Assume that the worst case holds; this means that the only way to decrease the set S is to make comparisons between pairs of its elements.

Imagine a graph in which the nodes represent the values in the input and a directed edge from a to b implies that b is greater than a. A subset of the nodes is said to be *stable* if no pair from it is connected by an edge. (In Figure 13.13, the nodes e, b, g, and f form a stable set.) Then the size of S at time t can be expressed as

$$\begin{aligned}|S \text{ at time } t| \;\geq\; & \min\ \{\max\ (h|G \text{ contains a stable set of size } h)| \\ & G \text{ is a graph with } |S| \text{ nodes and } n \text{ edges}\}\end{aligned}$$

It has been shown by Turan in *On the Theory of Graphs* (Colloq. Math., 1954) that the size of S at time t is $\geq$ the size of S at time $t-1$, squared and divided by $2p$ plus the size of S. We can solve this recurrence relation using the fact that initially the size of S equals n; this shows that the size of S will be greater than one so long as $t < \log\log n - c$. □

EXERCISES

1. [Valiant] Devise a parallel algorithm that produces the maximum of n unordered elements in $\log\log n + c$ parallel time, where c is a constant.

2. [Valiant] Devise a parallel sorting algorithm that takes a time of at most $2\log n \log\log n + O(\log n)$.

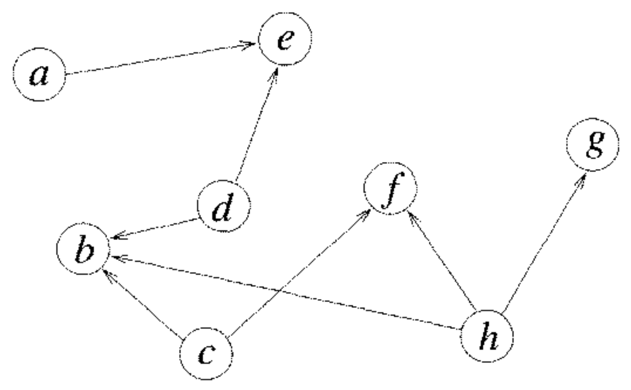


Figure 13.13 The set $\{e, f, b, g\}$ is stable.

3. **Theorem** Given n bits, any algorithm for computing the parity of these bits will need $\Omega(\frac{\log n}{\log \log n})$ time in the worst case if there are only a polynomial number of CRCW PRAM processors.

   Using this theorem prove that any algorithm for sorting n given numbers will need $\Omega(\frac{\log n}{\log \log n})$ time in the worst case, if the number of processors used is $n^{O(1)}$.

13.10 REFERENCES AND READINGS

Three excellent texts on parallel computing are:

Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes, by Tom Leighton, Morgan-Kaufmann, 1992.

Parallel Algorithms: Design and Analysis, by J. Já Já, Addison-Wesley, 1992.

Synthesis of Parallel Algorithms, edited by J. H. Reif, Morgan-Kaufmann, 1992.

For a definition of the PRAM and the metrics see:

"Parallel algorithmic techniques for combinatorial computation," by D. Eppstein and Z. Galil, *Annual Reviews in Computer Science* 3 (1988): 233–283.

"Performance metrics: keeping the focus on runtime," by S. Sahni and V. Thanvantri, *IEEE Parallel and Distributed Technology*, Spring 1996: 1–14.

Material on prefix computation and list ranking can be found in the three texts mentioned above.

The work-optimal randomized selection algorithm is based on:

"Expected time bounds for selection," by R. W. Floyd and R. L. Rivest, *Communications of the ACM* 18, no. 3 (1975): 165–172.

"Derivation of randomized algorithms for sorting and selection," by S. Rajasekaran and J. H. Reif. in *Parallel Algorithm Derivation And Program Transformation*, edited by R. Paige, J. H. Reif, and R. Wachter, Kluwer, 1993, pp. 187–205.

For a survey of parallel sorting and selection algorithms see

"Sorting and selection on interconnection networks," by S. Rajasekaran, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 21, 1995: 275–296.

Preparata's algorithm first appeared in "New parallel sorting schemes," by F. P. Preparata, *IEEE Transactions on Computers* C27, no. 7 (1978): 669–673. The original Reischuk's algorithm was recursive and appeared in "Probabilistic parallel algorithms for sorting and selection," by R. Reischuk, *SIAM Journal of Computing* 14, no. 2 (1985): 396–409. The version presented in this chapter is based on "Random sampling techniques and parallel algorithms design," by S. Rajasekaran and S. Sen, in *Synthesis of Parallel Algorithms*, J. H. Reif, ed., Morgan-Kaufmann, 1993, pp. 411–451. A deterministic algorithm for sorting with a run time of $O(\log n)$ using n EREW PRAM processors can be found in "Parallel merge sort," by R. Cole, *SIAM Journal on Computing* 17, no. 4 (1988): 770–785.

For a proof of Theorem 13.19 see "Optimal and sub-logarithmic time randomized parallel sorting algorithms," by S. Rajasekaran and J. H. Reif, *SIAM Journal on Computing* 18, no.3 (1989): 594–607. Solutions to Exercises 2, 3, and 10 of Section 13.6 can be found in this paper.

The general paradigm for the solution of many graph problems is given in "Parallel computation and conflicts in memory access," by L. Kucera, *Information Processing Letters*, (1982): 93–96.

For more material on convex hull and related problems see the text by J. Já Já. For the lower bound proof for finding the maximum see "Parallelism in comparison problems," by L. Valiant, *SIAM Journal on Computing* 4, no. 3 (1975): 348–355.

Theorem 13.28 was first proved in "The average complexity of deterministic and randomized parallel comparison-sorting algorithms," by N. Alon and Y. Azar, *SIAM Journal on Computing* (1988): 1178–1192. The proof was greatly simplified in "The average-case parallel complexity of sorting," by R. B. Boppana, *Information Processing Letters* 33 (1989): 145–146. A proof of Lemma 13.7 can be found in "The effect of number of Hamiltonian paths on the complexity of a vertex coloring problem," by U. Manber and M. Tompa, *SIAM Journal on Computing* 13, (1984): 109–115. Lemma 13.6 was proved in "A mathematical theory of communication," by Shannon, *Bell System Technical Journal* 27 (1948): 379–423 and 623–56.

13.11 ADDITIONAL EXERCISES

1. Suppose you have a sorted list of n keys in common memory. Give an $O(\log n/(\log p))$ time algorithm that takes a key x as input and searches the list for x using p CREW PRAM processors.

2. A sequence of n keys $k_1, k_2, \ldots, k_n$ is input. The problem is to find the right neighbor of each key in sorted order. For instance if the input is $5.2, 7, 2, 11, 15, 13$, the output is $7, 11, 5.2, 13, \infty, 15$.

   (a) How will you solve this problem in $O(1)$ time using n^3 CRCW PRAM processors?

   (b) How will you solve the same problem using a Las Vegas algorithm in $\widetilde{O}(1)$ time employing n^2 CRCW PRAM processors?

3. The input is a sequence S of n arbitrary numbers with many duplications, such that the number of distinct numbers is $O(1)$. Present an $O(\log n)$ time algorithm to sort S using $\frac{n}{\log n}$ priority-CRCW PRAM processors.

4. A, B, and C are three sets of n numbers each, and ℓ is another number. Show how to check whether there are three elements, picked one each from the three sets, whose sum is equal to ℓ. Your algorithm should run in $\widetilde{O}(\log n)$ time using at most n^2 CRCW PRAM processors.

5. An array A of size n is input. The array can only be of one of the following three types:

   Type I: A has all zeros.
   Type II: A has all ones.
   Type III: A has $\frac{n}{4}$ ones and $\frac{3}{4}n$ zeros.

   How will you identify the type of A in $O(1)$ time using a Monte Carlo algorithm? You can use $\log n$ CRCW PRAM processors. Show that the probability of a correct answer will be $\geq 1 - n^{-\alpha}$ for any fixed $\alpha \geq 1$.

6. Input is an array A of n numbers. Any number in A occurs either only once or more than $n^{3/4}$ times. Elements that occur more than $n^{3/4}$ times each are called *significant* elements. Present a Monte Carlo

algorithm with a run time of $O(n^{3/4} \log n)$ to identify all the significant elements of A. Prove that the output will be correct with high probability.

7. Let A be an array of size n such that each element is marked with a bucket number in the range $[1, 2, \ldots, m]$, where m divides n. The number of elements belonging to each bucket is exactly $\frac{n}{m}$. Develop a randomized parallel algorithm to rearrange the elements of A so that the elements in the first bucket appear first, followed by the elements of the second bucket, and so on. Your Las Vegas algorithm should run in $\widetilde{O}(\log n)$ time using $\frac{n}{\log n}$ CRCW PRAM processors. Prove the correctness and time bound of your algorithm.

8. Input are a directed graph $G(V, E)$ and two nodes $v, w \in V$. The problem is to determine whether there exists a directed path from v to w of length ≤ 3. How will you solve this problem in $O(1)$ time using $|V|^2$ CRCW PRAM processors? Assume that G is available in common memory in the form of an adjacency matrix.

9. Given is an undirected graph $G(V, E)$ in adjacency matrix form. We need to decide if G has a *triangle*, that is, three mutually adjacent vertices. Present an $O(\log n)$ time, $(n^3/\log n)$-processor CRCW PRAM algorithm to solve this problem.