CHAPTER 12

Parallel and Distributed Algorithms

In this chapter we discuss the solution of problems by a number of processors working in concert. In specifying an algorithm for such a setting, we must specify not only the sequence of actions of individual processors, but also the actions they take in response to the actions of other processors. The organization and use of multiple processors has come to be divided into two categories: *parallel processing* and *distributed processing*. In the former, a number of processors are coupled together fairly tightly: they are similar processors running at roughly the same speeds and they frequently exchange information with relatively small delays in the propagation of such information. For such a system, we wish to assert that at the end of a certain time period, all the processors will have terminated and will collectively hold the solution to the problem. In distributed processing, on the other hand, less is assumed about the speeds of the processors or the delays in propagating information between them. Thus, the focus is on establishing that algorithms terminate at all, on guaranteeing the correctness of the results, and on counting the number of messages that are sent between processors in solving a problem. We begin by studying a model for parallel computation. We then describe several parallel algorithms in this model: sorting, finding maximal independent sets in graphs, and finding maximum matchings in graphs. We also describe the randomized solution of two problems in distributed computation: the choice coordination problem and the Byzantine agreement problem.

12.1. The PRAM Model

Our model for parallel computation will be the *synchronous parallel random access machine*, which we will abbreviate by PRAM. The parallel computer will consist of P processors, each of which can be viewed as supporting the RAM model of computation (see Section 1.5.1). There is a *global memory* consisting of M locations; each processor has a (small) constant number of local registers to

which it alone has access. Each of the P processors may read from and write into any of the M global memory locations; these global memory locations serve as the only mechanism for communication between the processors. Computation proceeds in a series of synchronous *parallel steps*. In a parallel step, each processor first chooses a global memory location whose contents it reads; next it executes an instruction on the operand fetched, together with any operands in its registers (the allowable instructions are any of those we allow for a conventional single-processor RAM). Finally, the step ends with the processor writing into a memory location of its choice. By our assumption of synchrony, every processor finishes executing step i before any processor begins executing step $i+1$. An instruction for the PRAM is a specification, for each processor, of the actions it is to perform in each of the three phases of a step. A parallel program is a sequence of such instructions.

We now address the important issue of *conflict resolution* in a PRAM: our definition of an instruction permits a number of processors to attempt to read from or write to the same global memory location in a step. Logically, there appears to be no problem in allowing several processors to read the contents of the same global memory location; however, physical limitations make this action difficult to implement in actual hardware. Of greater concern are the difficulties that arise when several processors attempt to write into the same global memory location; which of the (possibly differing) values is actually written into the memory location? A number of solutions have been proposed for this problem of *concurrent writing*. We will adopt the simplest of these: we insist that the parallel program ensure that no execution will ever result in a concurrent write. Thus we deal only with *exclusive write* PRAMs.

As mentioned above, the issue of whether or not to allow concurrent reads is a matter of attention to hardware implementation. These various read/write models for PRAMs are abbreviated as EREW, CREW, and CRCW, where the first two letters denote whether reading is exclusive or concurrent and the last two denote what is permissible for writing. In this chapter, we will only consider EREW and CREW PRAMs.

Of particular theoretical interest is the solution of problems by PRAM algorithms in which the number of processors P is a polynomial function of the input size n, and the number of PRAM steps is bounded by a polylogarithmic function of n. We define the classes ***NC*** and ***RNC*** to capture these notions.

► **Definition 12.1:** The class ***NC*** consists of languages L that have a PRAM algorithm A such that for any $x \in \Sigma^*$

- $x \in L \Rightarrow A(x)$ accepts;
- $x \notin L \Rightarrow A(x)$ rejects;
- the number of processors used by A on x is polynomial in $|x|$;
- the number of steps used by A on x is polylogarithmic in $|x|$.

For randomized PRAM algorithms, we similarly define the class ***RNC***:

► **Definition 12.2:** The class ***RNC*** consists of languages L that have a PRAM algorithm A such that for any $x \in \Sigma^*$

- $x \in L \Rightarrow \mathbf{Pr}[A(x) \text{ accepts}] \geq 1/2$;
- $x \notin L \Rightarrow \mathbf{Pr}[A(x) \text{ accepts}] = 0$;
- the number of processors used by A on x is polynomial in $|x|$;
- the number of steps used by A on x is polylogarithmic in $|x|$.

As in the case of ***RP***, although the definition is in terms of decision or language problems, there is an obvious generalization to function computations. Notice that an ***RNC*** algorithm is Monte Carlo with one-sided error. We can define the two-sided error version analogous to ***BPP***. The Las Vegas version of this class (zero-error and polylogarithmic expected time) is called ***ZNC***, and is defined similar to ***ZPP***.

Exercise 12.1: In the above definitions, we did not distinguish between the various models of concurrent reading and writing. Show that if a problem has a CRCW PRAM algorithm using a number of processors that is polynomial in the input size, and a number of steps that is polylogarithmic, then the problem has an EREW PRAM algorithm using a number of processors that is polynomial in the input size, and a number of steps that is polylogarithmic.

12.2. Sorting on a PRAM

In this section we study algorithms for sorting n numbers on a PRAM with n processors. For convenience, we will assume that the input numbers to be sorted all have distinct values. Our eventual goal will be a randomized (***ZNC***) algorithm that terminates in $O(\log n)$ steps with high probability. Such an algorithm would thus result in a total of $O(n \log n)$ operations among all processors, with high probability.

Consider the implementation of the following variant of randomized quicksort on a CREW PRAM. Initially, each of the n processors contains a distinct input element. We first describe the structure of the algorithm. Following this high-level description, we will break down each stage of this description into a sequence of PRAM steps. Let P_i denote the ith processor.

0. If $n = 1$ stop.

1. We pick a *splitter* uniformly at random from the n input elements.

2. Each processor determines whether its element is bigger or smaller than the splitter.

3. Let j denote the rank of the splitter. If $j \notin [n/4, 3n/4]$, we declare the step a failure and repeat starting at (1) above. If $j \in [n/4, 3n/4]$, the step is a success.

We then move the splitter to P_j. Each element that is smaller than the splitter is moved to a distinct processor P_i for $i < j$. Each element that is larger than the splitter is moved to a distinct processor P_k for $k > j$.

4. We sort the elements recursively in processors P_1 through P_{j-1}, and the elements in processors P_{j+1} through P_n. These recursive sorts are independent of each other.

Let us study the number of CREW PRAM steps taken by each of the above stages. Before we proceed with a detailed analysis, we make a prognosis of what we need in order for the above algorithm to terminate in $O(\log n)$ steps. The best we can hope for is success whenever we split. If we were fortunate enough that this were to happen, every sequence of recursive splits would terminate within $O(\log n)$ stages. Even so, in order for the algorithm to terminate in $O(\log n)$ steps, we would require each split to be implemented in a constant number of steps. Unfortunately we know of no way of doing this.

The second stage in our scheme is trivial and can be implemented in a single step of a CREW PRAM. Let us turn to Stage 3 of the above description. Our goal is to ensure that processor P_i, for $i < j$, contains a distinct input element whose rank is smaller than j, and similarly processor P_k for $k > j$, contains a distinct input element whose rank is larger than j. How many PRAM steps are taken up by this process?

Processor P_i sets a bit b_i in one of its registers to 0 if its element is greater than the splitter, and to 1 otherwise. For all i, let $S_i = \sum_{t \le i} b_t$.

Exercise 12.2: Devise a PRAM algorithm by which, given the b_i, the S_i can be computed (with the result contained in P_i) in $O(\log n)$ steps. Using this, show how Stage 3 of the algorithm can be implemented in $O(\log n)$ steps.

Thus, we see that a single splitting stage can be implemented in $O(\log n)$ steps of a CREW PRAM. In Problem 12.1 we will see that from this, we can infer that the above algorithm terminates in $O\left(\log^2 n\right)$ steps with high probability.

The shortcoming of the above scheme is that the splitting work in Stage 3, consuming $O(\log n)$ steps, yielded a relatively small benefit – it cuts the problem size down from n to a constant fraction of n. To improve on this, we consider a more efficient algorithm in which we invest the same amount of work in splitting, but in the process break up the problem into pieces of size $n^{1-\epsilon}$ for a fixed constant ϵ. If we could do this, we could hope for an overall parallel running time of $O(\log n)$ steps: at the next level of recursion, the splitting time would be logarithmic in $n^{1-\epsilon}$, which is a constant fraction of the splitting time at the first level. Thus, the times for proceeding from one level of recursion to the next would form a geometric series summing to $O(\log n)$. The following two exercises pave the way for a concrete scheme for implementing this idea. Exercise 12.3 demonstrates that we can indeed sort in $O(\log n)$ steps if our PRAM were endowed with many more processors than elements to be sorted.

Exercise 12.3: Consider a CREW PRAM having n^2 processors. Suppose that each of the processors P_1 through P_n has an input element to be sorted. Give a deterministic algorithm by which this PRAM can sort these n elements in $O(\log n)$ steps. (**Hint:** We have enough processors to compare all pairs of elements.)

Next, suppose that we have n processors and n elements. Suppose that processors P_1 through P_r contain r of the elements in sorted order, and that processors P_{r+1} through P_n contain the remaining $n-r$ elements. Call the sorted elements in the first r processors the *splitters*. For $1 \leq j \leq r$, let s_j denote the jth largest splitter. Our goal is to "insert" the $n-r$ unsorted elements among the splitters, in the following sense.

1. Each processor should end up with a distinct input element.
2. Let $i(s_j)$ denote the index of the processor containing s_j following the insertion operation. Then, for all $k < i(s_j)$, processor P_k contains an element that is smaller than s_j; similarly, for all $k > i(s_j)$, processor P_k contains an element that is larger than s_j.

In other words, the splitters are contained in processors in increasing order, and the remaining elements are in processors between their "adjacent" splitters.

Exercise 12.4: For n processors, and n elements of which $\sqrt{n}$ are splitters, give a deterministic scheme that completes the above insertion process in $O(\log n)$ steps.

Here are the stages of our parallel sorting algorithm, which we call **BoxSort**. Note that it is a Las Vegas algorithm: it always produces the correct output. Further, it always uses a fixed number of processors; only the number of parallel steps is a random variable. This will be typical of all the parallel algorithms we present. The function **LogSort** is described following Exercise 12.5.

Algorithm BoxSort:

Input: A set of numbers S.

Output: The elements of S sorted in increasing order.

1. Select $\sqrt{n}$ elements at random from the n input elements. Using all n processors, sort them in $O(\log n)$ steps (using the ideas in Exercise 12.3). If two splitters are adjacent in this sorted order, we call them *adjacent splitters*.
2. Using the sorted elements from Stage 1 as splitters, insert the remaining elements among them in $O(\log n)$ steps (using the ideas in Exercise 12.4).
3. Treating the elements that are inserted between adjacent splitters as sub-problems, recur on each sub-problem whose size exceeds $\log n$. For sub-problems of size $\log n$ or less, invoke **LogSort**.

Note that in Step 3 we have available as many processors as elements for each sub-problem on which we recur. The sub-problems that result from the $\sqrt{n}$ splitters have size roughly $\sqrt{n}$, with good probability. This fits with our paradigm for progressing from a problem of size n to one of size $n^{1-\epsilon}$ in $O(\log n)$ steps. As we will see below, with high probability every sub-problem resulting from a splitting operation is small, provided the set being split is itself not too small. We deal with this issue using the following idea. When we have $\log n$ elements to be sorted using $\log n$ processors, we abandon the recursive approach and use brute force:

Exercise 12.5: Show that a CREW PRAM with m processors can sort m elements deterministically in $O(m)$ steps.

Thus, when a sub-problem size is down to $\log n$, we can sort it with the $\log n$ available processors in $O(\log n)$ steps; we call this operation **LogSort**.

We now analyze the use of random sampling for choosing the splitters. Let us call the set of elements that fall between adjacent splitters a *box*. The analysis is similar to the one we used in the analysis of randomized selection in Section 3.3. By invoking the Chernoff bound instead of the Chebyshev bound, the following is an easy consequence:

Exercise 12.6: Consider m splitters chosen uniformly at random from m^2 given distinct elements. Show that the probability that a box has size exceeding bm is at most ma^b, for a constant $a < 1$.

To complete the analysis of the algorithm, we represent an execution of the algorithm by a tree. Each node of the tree is a box that arises during the execution. For this purpose, we will also regard the n input elements as forming a box (of size n), and this is the root of our tree. The children of a node are the boxes that arise when it is partitioned by random splitters. Each leaf is a box of size at most $\log n$.

We are interested in root-leaf paths in this tree. In bounding the running time of algorithm, the quantity of interest is not the length of such root-leaf paths, but rather the number of PRAM steps that elapse as we go down such a path. This is because the time to proceed from a box to one of its children is logarithmic in the size of the box. We will argue that with high probability, the sum of the logarithms of box sizes on any root-leaf path is $O(\log n)$, and this will yield an overall running time of $O(\log n)$.

The idea is to partition the interval $[1, n]$ into sub-intervals $I_0, I_1, \ldots$, and bound the probability that a box whose size is in I_k has a child whose size is also in I_k. To this end, let γ and d be fixed constants such that $1/2 < \gamma < 1$ and $1 < d < 1/\gamma$. For a positive integer k, define $\tau_k = d^k$, $\rho_k = n^{\gamma^k}$, and the interval $I_k = [\rho_{k+1}, \rho_k]$.

Exercise 12.7: Show that $\rho_k < \log n$ for a value of $k \leq c \log\log n$, for a constant c that depends only on γ.

Thus we confine our attention to $O(\log\log n)$ intervals I_k. For a box B in the tree, we say that $\alpha(B) = k$ if $|B| \in I_k$. In terms of this notation, the time to split B is $O(\log \rho_{\alpha(B)})$. For a root-leaf path $\zeta = (B_1, \ldots, B_t)$, we will study $\sum_{j=1}^{t} \log \rho_{\alpha(B_j)}$, since the overall running time of the algorithm is

$$O\left(\log n + \max_{\zeta} \sum_{j=1}^{t} \log \rho_{\alpha(B_j)}\right).$$

For a path $\zeta = (B_1, \ldots, B_t)$, we say that event $\mathcal{E}_\zeta$ holds if the sequence $\alpha(B_1), \ldots, \alpha(B_t)$ does not contain the value k more than τ_k times, for $1 \leq k \leq c \log\log n$. If $\mathcal{E}_\zeta$ holds, the number of PRAM steps spent on path ζ is at most

$$O\left(\log n + \sum_{k=1}^{\infty} \tau_k \gamma^k \log n\right).$$

Since $\tau_k = d^k$, and $\gamma d < 1$, this sums to $O(\log n)$. Thus it suffices to argue that $\mathcal{E}_\zeta$ holds with high probability for any ζ. This is an easy calculation following the bound from Exercise 12.6.

Lemma 12.1: *There is a constant $\beta > 1$ such that $\mathcal{E}_\zeta$ holds with probability at least $1 - \exp(-\log^\beta n)$.*

The following sequence of three probability calculations establishes Lemma 12.1. These calculations are straightforward, and the reader is asked to perform them in Problem 12.2.

1. Bound the probability that $\alpha(B_{j+1}) = \alpha(B_j)$ using the result of Exercise 12.6.
2. Bound the probability that for any particular k, the value k is contained more than τ_k times in the sequence $\alpha(B_1), \ldots, \alpha(B_t)$.
3. Bound the probability that for $1 \leq k \leq c \log\log n$, the value k is contained more than τ_k times in the sequence $\alpha(B_1), \ldots, \alpha(B_t)$.

Since the number of paths ζ in an execution is at most n, we have:

Theorem 12.2: *There is a constant $b > 0$ such that with probability at least $1 - \exp(-\log^b n)$ the algorithm* **BoxSort** *terminates in $O(\log n)$ steps.*

12.3. Maximal Independent Sets

Let $G(V, E)$ be an undirected graph with n vertices and $m = \Omega(n)$ edges. A subset of vertices $I \subseteq V$ is said to be *independent* in G if no edge in E has both

its end-points in I. Equivalently, I is independent if for all $v \in I$, $\Gamma(v) \cap I = \emptyset$. Recall that $\Gamma(v)$ is the set of vertices in V that are adjacent to v and that the degree of v is $d(v) = |\Gamma(v)|$.

An independent set I is *maximal* if it is not properly contained in any other independent set in G. Recall that the problem of finding a *maximum* independent set is ***NP***-hard. In contrast, finding a maximal independent set (MIS) is trivial in the sequential setting. The following greedy algorithm constructs an MIS in $O(m)$ time.

Algorithm Greedy MIS:

Input: Graph $G(V, E)$ with $V = \{1, 2, \ldots, n\}$.

Output: A maximal independent set $I \subseteq V$.

1. $I \leftarrow \emptyset$.

2. **for** $v = 1$ **to** n **do**
    if $I \cap \Gamma(v) = \emptyset$ **then** $I \leftarrow I \cup \{v\}$.

Exercise 12.8: Prove that the **Greedy MIS** algorithm terminates in $O(m)$ time with a maximal independent set, if the input is given in the form of an adjacency list.

A greedy algorithm such as this is inherently sequential. The output of this algorithm is called the *lexicographically first* MIS (LFMIS). It is known that the existence of an ***NC*** (or ***RNC***) algorithm for finding the LFMIS would imply that ***P*** = ***NC*** (respectively, ***P*** = ***RNC***), a consequence that appears almost as unlikely as ***P*** = ***NP***. Thus, we have the somewhat paradoxical situation that the most trivial algorithm finds the LFMIS sequentially, whereas it appears impossible to solve it fast in parallel. However, it turns out that there are simple parallel algorithms for finding an MIS (not necessarily the lexicographically first MIS). We start by describing an ***RNC*** algorithm and later indicate how it can be derandomized to obtain an ***NC*** algorithm. The problem of verifying an MIS is relatively easy to solve in parallel.

Exercise 12.9: Devise a deterministic EREW PRAM algorithm for *verifying* that a set I is an MIS, using $O(m/\log m)$ processors and $O(\log m)$ time.

Consider the variant of the **Greedy MIS** algorithm, which starts with $I = \emptyset$ and repeatedly performs the following step: pick any vertex v, add v to I, and delete v and $\Gamma(v)$ from the graph. The algorithm terminates when all vertices have either been deleted or added to I. Choosing v to be the lowest numbered vertex present in the graph leads to exactly the same outcome as in **Greedy MIS**.

The key idea behind the parallel algorithm is to generalize the basic iterative step in the new algorithm: find an independent set S, add S to I, and delete $S \cup \Gamma(S)$ from the graph. The trick is to ensure that each iteration can be implemented fast in parallel, while also guaranteeing that the total number of iterations is small. One way of ensuring that the number of iterations is small is to choose an independent set S such that $S \cup \Gamma(S)$ is large. This is difficult, but we achieve the same effect by ensuring that the number of edges incident on $S \cup \Gamma(S)$ is a large fraction of the total number of remaining edges; clearly, this will result in an empty graph in a small of number of iterations.

To find such an independent set S, we pick a large random set of vertices $R \subseteq V$. While it is quite unlikely that R will be independent, biasing the sampling in favor of low degree vertices will ensure that there are very few edges with both end-points in R. To obtain the independent set from R we consider each edge of this type and drop the end-point of lower degree. This results in an independent set, and the choice of the end-point retained for S ensures that $\Gamma(S)$ is likely to be large.

This idea is implemented in Algorithm **Parallel MIS**, where the marking of a vertex corresponds to selecting it for the set R. We assume that each vertex (and edge) of G is assigned a dedicated processor that performs the parallel tasks associated with that vertex (or edge). This uses a total of $O(n+m)$ processors.

Algorithm Parallel MIS:

Input: Graph $G(V, E)$.

Output: A maximal independent set $I \subseteq V$.

1. $I \leftarrow \emptyset$.

2. repeat

2.1. for all $v \in V$ **do** (in parallel)
if $d(v) = 0$ **then** add v to I and delete v from V
else mark v with probability $1/2d(v)$.

2.2. for all $(u, v) \in E$ **do** (in parallel)
if both u and v are marked
then unmark the lower degree vertex.

2.3. for all $v \in V$ **do** (in parallel)
if v is marked **then** add v to S.

2.4. $I \leftarrow I \cup S$.

2.5. delete $S \cup \Gamma(S)$ from V, and all incident edges from E.

until $V = \emptyset$

Ties are broken arbitrarily in Step 2.2. It is clear that the set S in Step 2.3 is an independent set. The reader should verify that this algorithm is guaranteed to terminate with a maximal independent set in a linear number of iterations. Our

goal is to prove that the random choices in Step 2.1 will ensure that the expected number of iterations is in fact $O(\log n)$. We leave the details of implementing each iteration in *NC* as an exercise.

Exercise 12.10: Show that each iteration of the **Parallel MIS** algorithm can be implemented in $O(\log n)$ time using an EREW PRAM with $O(n+m)$ processors.

The analysis is based on showing that the expected fraction of edges removed from E during each iteration is bounded from below by a constant. In fact, we will focus only on a specific class of *good* edges, defined as follows.

► **Definition 12.3:** A vertex $v \in V$ is *good* if it has at least $d(v)/3$ neighbors of degree no more than $d(v)$; otherwise, the vertex is *bad*. An edge is good if at least one of its end-points is a good vertex, and it is bad if both end-points are bad vertices.

In the following discussion, we will analyze only a single iteration of the **Parallel MIS** algorithm. The notion of goodness is with respect to the vertices and edges surviving at the start of that specific iteration. It should be clear that the argument can be applied repeatedly to the successive iterations; together with Theorem 1.3, this implies the result.

We start with an intuitive sketch of the analysis, which is then fleshed out in a sequence of lemmas. A good vertex is quite likely to have one of its lower degree neighbors in S and, thereby be deleted from V. We will show that the number of good edges is large, and since good vertices are likely to be deleted, a large number of edges will be deleted during each iteration.

Lemma 12.3: *Let $v \in V$ be a good vertex with degree $d(v) > 0$. Then, the probability that some vertex $w \in \Gamma(v)$ gets marked is at least $1 - \exp(-1/6)$.*

PROOF: Each vertex $w \in \Gamma(v)$ is marked independently with probability $1/2d(w)$. Since v is good, there exist at least $d(v)/3$ vertices in $\Gamma(v)$ with degree at most $d(v)$. Each of these neighbors gets marked with probability at least $1/2d(v)$. Thus, the probability that none of these neighbors of v gets marked is at most

$$\left(1 - \frac{1}{2d(v)}\right)^{d(v)/3} \leq e^{-1/6}.$$

The remaining neighbors of v can only help in increasing the probability under consideration. □

Lemma 12.4: *During any iteration, if a vertex w is marked then it is selected to be in S with probability at least $1/2$.*

PROOF: The only reason a marked vertex w becomes unmarked, and hence not selected for S, is that one of its neighbors of degree at least $d(w)$ is also marked. Each such neighbor is marked with probability at most $1/2d(w)$, and the number of such neighbors certainly cannot exceed $d(w)$. Thus, the probability that a marked vertex is selected to be in S is at least

$$\begin{aligned} 1 \quad - \quad & \mathbf{Pr}[\exists x \in \Gamma(w) \text{ such that } d(x) \geq d(w) \text{ and } x \text{ is marked}] \\ \geq \quad & 1 - |\{x \in \Gamma(w) \mid d(x) \geq d(w)\}| \times \frac{1}{2d(w)} \\ \geq \quad & 1 - \sum_{x \in \Gamma(w)} \frac{1}{2d(w)} \\ = \quad & 1 - d(w) \times \frac{1}{2d(w)} \\ = \quad & \frac{1}{2}. \end{aligned}$$

□

Let v be a good vertex with $d(v) > 0$, and consider the event $\mathcal{E}$ that some vertex in $\Gamma(v)$ does indeed get marked. Let w be the lowest-numbered marked vertex in $\Gamma(v)$. By Lemma 12.4, the probability that w is selected to be in S is at least $1/2$. Clearly, if $w \in S$, then v must belong to $S \cup \Gamma(S)$. Using the bound on the probability of the event $\mathcal{E}$ from Lemma 12.3, we obtain the following.

Lemma 12.5: *The probability that a good vertex belongs to $S \cup \Gamma(S)$ is at least $(1 - \exp(-1/6))/2$.*

The final step is to bound the number of good edges.

Lemma 12.6: *In a graph $G(V, E)$, the number of good edges is at least $|E|/2$.*

PROOF: Direct the edges in E from the lower degree end-point to the higher degree end-point, breaking ties arbitrarily. Define $d_i(v)$ and $d_o(v)$ as the in-degree and out-degree, respectively, of the vertex v in the resulting digraph. It follows from the definition of goodness that for each bad vertex v,

$$d_o(v) - d_i(v) \geq \frac{d(v)}{3} = \frac{d_o(v) + d_i(v)}{3}.$$

For all S, $T \subseteq V$, define the subset of the (oriented) edges $E(S, T)$ as those edges that are directed from vertices in S to vertices in T; further, define $e(S, T)$ to be $|E(S, T)|$. Let V_G and V_B be the set of good and bad vertices, respectively. The total degree of the bad vertices is given by

$$\begin{aligned} 2e(V_B, V_B) \quad + \quad & e(V_B, V_G) + e(V_G, V_B) \\ = \quad & \sum_{v \in V_B} (d_o(v) + d_i(v)) \\ \leq \quad & 3 \sum_{v \in V_B} (d_o(v) - d_i(v)) \end{aligned}$$

$$
\begin{aligned}
&= 3\sum_{v \in V_G} (d_i(v) - d_o(v)) \\
&= 3[(e(V_B, V_G) + e(V_G, V_G)) - (e(V_G, V_B) + e(V_G, V_G))] \\
&= 3[e(V_B, V_G) - e(V_G, V_B)] \\
&\leq 3[e(V_B, V_G) + e(V_G, V_B)]
\end{aligned}
$$

The first and last expressions in this sequence of inequalities imply that $e(V_B, V_B) \leq e(V_B, V_G) + e(V_G, V_B)$. Since every bad edge contributes to the left side and only good edges contribute to the right side, the desired result follows. □

Since a constant fraction of the edges are incident on good vertices, and good vertices get eliminated with a constant probability, it follows that the expected number of edges eliminated during an iteration is a constant fraction of the current set of edges. By Theorem 1.3, this implies that the expected number of iterations of the **Parallel MIS** algorithm is $O(\log n)$.

Theorem 12.7: *The* **Parallel MIS** *algorithm has an EREW PRAM implementation running in expected time* $O(\log^2 n)$ *using* $O(n+m)$ *processors.*

It is straightforward to obtain a high-probability version of this result.

We briefly describe the construction of an ***NC*** algorithm for MIS obtained by a derandomization of the ***RNC*** algorithm described above. The first step is to show that the preceding analysis works even when the marking of the vertices is not completely independent, but instead is only pairwise independent. Note that the only part of the analysis that uses complete independence is Lemma 12.3. In Problem 12.9 the reader is asked to prove that a marginally weaker version of Lemma 12.5 holds even with pairwise independent marking of vertices. The key advantage of pairwise independence is that only $O(\log n)$ random bits are required to generate the sample points in the corresponding probability space (see the discussion in Section 3.4). In the current application, it is necessary to generate pairwise independent Bernoulli random variables that are not uniform. In Problem 12.10, the reader is asked to modify the earlier construction of pairwise independent probability space to apply to Bernoulli variables that take on the value 1 with non-uniform probabilities, i.e., the marking probabilities of $1/2d(v)$.

The final and most crucial idea is to observe that the total number of choices of the $O(\log n)$ random bits needed for generating pairwise independent marking is polynomially bounded. All such choices can be tried in parallel to see if they yield a good marking, i.e., a marking of vertices that leads to an appropriately large reduction in the number of edges. Note that in each iteration, we are guaranteed that most choices of the random bits will give a good marking; in particular, there exists at least one setting of the $O(\log n)$ random bits that will provide a good marking. Trying all possibilities will (deterministically) identify a good marking. Thus, each iteration can be derandomized and the entire algorithm can be implemented in ***NC***.

12.4. Perfect Matchings

We now turn to the problem of finding an independent set of *edges* (or a *matching*) in a graph. Let $G(V,E)$ be a graph with the vertex set $V = \{1,\ldots,n\}$; without loss of generality, we may assume that n is even. Recall (Chapter 7) that a *matching* in G is a collection of edges $M \subset E$ no two of which are incident on the same vertex. A *maximal matching* is a matching that is not properly contained in any other matching. A *maximum matching* is a matching of maximum cardinality, and a *perfect matching* is one containing an edge incident on every vertex of G.

The matchings in a graph $G(V,E)$ correspond to independent sets in the *line graph* H obtained by creating a vertex for each edge in E, with two such vertices being adjacent if the corresponding edges in E are incident on the same vertex. This implies that the problem of finding matchings is a special case of the independent set problem. A maximal matching can be found sequentially via a greedy algorithm, and on a PRAM, as suggested in Problem 12.6, using the algorithms discussed in Section 12.3. Unlike the case of maximum independent sets, the problem of finding a maximum matching has a polynomial time solution. This raises the possibility of constructing an ***NC*** algorithm for maximum matchings. However, randomization appears to be an essential component of all known fast parallel algorithms for maximum matching, and we devote this section to describing one such ***RNC*** algorithm.

For now we focus on the problem of finding a perfect matching in a graph that is guaranteed to have one, deferring the issue of finding a maximum matching till later. First we show that the decision problem of determining the *existence* of a perfect matching is in ***RNC***. This is based on the algebraic techniques developed in Chapter 7; the reader is advised to review Sections 7.2 and 7.3 from that chapter. We make use of Tutte's Theorem described in Problem 7.8; this is a generalization of Theorem 7.3, which dealt with the case of bipartite matchings.

Theorem 12.8 (Tutte's Theorem): *Let A be the $n \times n$ (skew-symmetric) Tutte matrix of indeterminates obtained from $G(V,E)$ as follows: a distinct indeterminate x_{ij} is associated with the edge (v_i,v_j), where $i < j$, and the corresponding matrix entries are given by $A_{ij} = x_{ij}$ and $A_{ji} = -x_{ij}$, that is,*

$$A_{ij} = \begin{cases} x_{ij} & (v_i,v_j) \in E \text{ and } i < j \\ -x_{ji} & (v_i,v_j) \in E \text{ and } i > j \\ 0 & (u_i,v_j) \notin E \end{cases}$$

Then G has a perfect matching if and only if $\det(A)$ *is not identically zero.*

The ***RNC*** algorithm for deciding the existence of a perfect matching in G first constructs the matrix A with each indeterminate replaced by independently and uniformly chosen random values from a suitably large set of integers, as described in Section 7.2. Then, it evaluates the determinant of the resulting

integer matrix. If G has a perfect matching, then with suitably large probability, the determinant will be non-zero. On the other hand, if G does not have any perfect matchings, the determinant will always be zero.

The first stage of this algorithm is easily implemented in ***NC***. Finding the determinant of a matrix in ***NC*** is not trivial, but at least one ***NC*** algorithm is known (see the Notes section). Thus the problem of deciding the existence of a perfect matching is in ***RNC***.

We turn to the task of actually finding a perfect matching in a graph. Once again, the idea is to reduce the search problem to some matrix computations. We summarize known results for parallel matrix computations without attempting to describe the algorithms in any detail.

The (i, j) *minor* of a matrix $\boldsymbol{U}$, denoted $\boldsymbol{U}^{ij}$, is the matrix obtained by deleting the ith row and the jth column of $\boldsymbol{U}$. The *adjoint* $\text{adj}(\boldsymbol{U})$ of the matrix $\boldsymbol{U}$ is the matrix $\boldsymbol{A}$ whose (j, i) entry has absolute value equal to the determinant of the (i, j) minor of $\boldsymbol{U}$, i.e., $A_{ji} = (-1)^{i+j} \det(\boldsymbol{U}^{ij})$. It is easy to verify the following relation: $\boldsymbol{U}\text{adj}(\boldsymbol{U}) = \det(\boldsymbol{U})$.

Theorem 12.9: *Let $\boldsymbol{U}$ be an $n \times n$ matrix whose entries are k-bit integers. Then the determinant, adjoint, and inverse of $\boldsymbol{U}$ can be computed in **NC**. In particular, let $\text{MM}(n) = \text{O}(n^{2.376})$ denote the number of arithmetic operations required to multiply two $n \times n$ matrices. Then the determinant can be computed in $\text{O}(\log^2 n)$ time using $\text{O}(n^2 \text{MM}(n))$ processors; further, there are **RNC** algorithms for computing the inverse and the adjoint running in time $\text{O}(\log^2 n)$ using $\text{O}(n^{3.5}k)$ processors.*

It is instructive to attempt to search for perfect matchings using the decision algorithm described above. It is not very hard to see that this can be done for the special case where the graph has a *unique* perfect matching.

Exercise 12.11: Suppose that G has a unique perfect matching M. Analyze the effect of removing an edge on the determinant of the Tutte matrix, considering both the case where the edge belongs to M and where it does not belong to M. Using this analysis, devise an ***RNC*** algorithm for finding the matching M.

As outlined in Problem 12.15, an ***NC*** algorithm is possible for finding a unique perfect matching. In fact, it is known that there is an ***NC*** algorithm for finding perfect matchings in graphs with a polynomial number of perfect matchings. However, these algorithms break down when the number of perfect matchings in the graph is large.

The problem with having a large number of perfect matchings is that it is necessary to coordinate the processors to search for the same perfect matching. This is the major stumbling block in the parallel solution of the matching problem and is perhaps the main reason why no ***NC*** algorithm is known. If the number of matchings is small, then the processors can easily focus on the

same perfect matching. The first ingredient in the ***RNC*** algorithm is to take an arbitrary graph and *isolate* a specific perfect matching. The isolation is achieved by assigning weights to the edges and looking for a minimum weight perfect matching. Of course, there is no reason why there should be a unique minimum weight perfect matching but, as we show in the next section, if the weights are chosen at random there is a good chance that isolation occurs.

12.4.1. The Isolating Lemma

Our goal now is to define a positive integer weight function over the edges of G, say $w : E \rightarrow \mathbb{Z}^+$, such that there is a unique minimum weight perfect matching. Observing that the set of all possible perfect matchings can be viewed as a family of subsets of E, we consider a more general setting involving an arbitrary set family.

► **Definition 12.4:** A *set system* $(X, \mathcal{F})$ consists of a finite universe $X = \{x_1, \ldots, x_m\}$ and a family of subsets $\mathcal{F} = \{S_1, \ldots, S_k\}$, where $S_i \subseteq X$ for $1 \leq i \leq k$. The *dimension* of the set system is (the size of the universe) m.

Given a positive integer weight function $w : X \rightarrow \mathbb{Z}^+$, we define the weight of a set $S \subseteq X$ as $w(S) = \sum_{x_j \in S} w(x_j)$. The following lemma shows that a random weight function is quite likely to lead to a unique set of $\mathcal{F}$ being of minimum weight.

Lemma 12.10 (Isolating Lemma): *Suppose $(X, \mathcal{F})$ is a set system of dimension m. Let $w : X \rightarrow \{1, \ldots, 2m\}$ be a positive integer weight function defined by assigning to each element of X a random weight chosen uniformly and independently from $\{1, \ldots, 2m\}$. Then,*

$$\mathbf{Pr}[\textit{there is a unique minimum weight set in } \mathcal{F}] \geq \frac{1}{2}.$$

Remark: This lemma is truly counterintuitive. First of all, the size of $\mathcal{F}$ is completely irrelevant to the claim. This allows the family $\mathcal{F}$ to be of size as large as 2^m. Since the weights of the sets must lie in the range $\{1, \ldots, 2m^2\}$, one would expect that there could be as many as $2^m/(2m^2)$ sets of any given weight. However, the weights of the sets follow the lattice structure of the family of all subsets of X, thereby ensuring that the weights of the sets are not independent or uniformly distributed.

PROOF: We assume, without loss of generality, that each element of X occurs in at least one of the sets in $\mathcal{F}$. Suppose that we have chosen the (random) weights of all elements of X except one, say x_i. Let W_i be the weight of a minimum weight set containing x_i, computed by ignoring the (undetermined) weight of x_i. Further, let $\overline{W}_i$ be the weight of a minimum weight set not containing the element x_i. Define $\alpha_i = \overline{W}_i - W_i$ and note that α_i could be negative.

Suppose that initially x_i is assigned the weight $-\infty$ (actually, $-2m^2$ will suffice). It is clear that now every set of minimum weight must contain x_i. Consider the effect of increasing the weight of x_i until it reaches $+\infty$ (here, $2m^2$ will suffice). At this point it is clear that no set of minimum weight contains x_i.

We claim that for $w(x_i) < \alpha_i$, every minimum weight set must contain x_i, because there exists a set containing x_i of weight $W_i + w(x_i) < \overline{W}_i$, and all sets not containing x_i must have weight at least $\overline{W}_i$. Similarly, we claim that for $w(x_i) > \alpha_i$, no minimum weight set contains x_i, because any set containing x_i has weight at least $W_i + w(x_i) > \overline{W}_i$, and there exists a set not containing x_i of weight $\overline{W}_i$.

Thus, so long as $w(x_i) \neq \alpha_i$, either every minimum weight set contains x_i or none of them contains x_i. We say that x_i is ambiguous when $w(x_i) = \alpha_i$, since then it cannot be said for certain whether x_i will belong to a minimum weight set chosen arbitrarily. The crucial observation is that since α_i depends only on the weights of the elements other than x_i, and the weights are chosen independently, the random variable α_i is independent of $w(x_i)$. It follows that the probability that x_i is ambiguous is no more than $1/2m$. Note that it is possible that $\alpha_i \notin \{1, \ldots, 2m\}$, in which case the probability is actually zero.

While the ambiguities of the different elements are correlated, it is safe to say that the probability that there exists an ambiguous element in X is at most

$$m \times \frac{1}{2m} = \frac{1}{2}.$$

It follows that with probability at least a half, none of the elements is ambiguous. But if there exist two distinct minimum weight sets, say S_i and S_j, there must be an element that belongs to one of these sets but not the other, i.e., there must be an ambiguous element. Thus, with probability at least a half there is a unique minimum weight set. □

Exercise 12.12: Determine the probability that there is a unique minimum weight set when the weights are chosen from the set $\{1, \ldots, t\}$.

Exercise 12.13: Does a similar result hold for the maximum weight set?

The application of this lemma to the perfect matching problem is obvious. Let X be the set of edges in the graph, and $\mathcal{F}$ the set of perfect matchings. It follows that assigning random weights between 1 and $2m$ to the edges leads to a unique minimum weight perfect matching with probability at least $1/2$. We now turn to the task of identifying this perfect matching.

12.4.2. The Parallel Matching Algorithm

Suppose we have chosen the random weight function w for the edges of G as described above, and let w_{ij} be the weight assigned to the edge (i, j). We will

assume that there is a unique minimum weight perfect matching, and that its weight is W. If there is more than one minimum weight perfect matching, the following algorithm will fail (the mode of failure will be evident from the description below). This happens with probability at most $1/2$, and the algorithm can be repeated to reduce the error probability.

Consider the Tutte matrix $\boldsymbol{A}$ corresponding to the graph G. Let $\boldsymbol{B}$ be the matrix obtained from $\boldsymbol{A}$ by setting each indeterminate x_{ij} to the (random) integer value $2^{w_{ij}}$.

Lemma 12.11: *Suppose that there is a unique minimum weight perfect matching and that its weight is W. Then,* $\det(\boldsymbol{B}) \neq 0$ *and, moreover, the highest power of* 2 *that divides* $\det(\boldsymbol{B})$ *is* 2^{2W}.

PROOF: The proof is a generalization of the proof of Tutte's theorem. For each permutation $\sigma \in \mathbb{S}_n$ defined over $V = \{1, \ldots, n\}$, define its *value* with respect to $\boldsymbol{B}$ as $\text{val}(\sigma) = \prod_{i=1}^{n} B_{i\sigma(i)}$. Observe that $\text{val}(\sigma)$ is non-zero if and only if for each $i \in V$, the edge $(i, \sigma(i))$ is present in G. Recall from Section 7.2 that the determinant of the matrix $\boldsymbol{B}$ is given by

$$\det(\boldsymbol{B}) = \sum_{\sigma \in \mathbb{S}_n} \text{sgn}(\sigma) \times \text{val}(\sigma),$$

where $\text{sgn}(\sigma)$ is the sign of a permutation σ. Permutations with sign $+1$ are called even, and those with sign -1 are called odd. The reader should not confuse the sign of a permutation with the sign of its value.

We focus only on the permutations with non-zero value, since the others do not contribute to the determinant. Let us first explicate the structure of the non-zero permutations. The *trail* of a permutation σ of non-zero value is the subgraph of G containing exactly the edges $(i, \sigma(i))$, for $1 \leq i \leq n$. It is convenient to view the edges $(i, \sigma(i))$ as being directed from i to $\sigma(i)$. The n edges corresponding to σ form a multiset where each edge has multiplicity 1 or 2, and the edges of multiplicity 2 occur with both orientations. Each vertex has two edges from the trail incident on it, one incoming and the other outgoing, and these may correspond to the two orientations of the same undirected edge from G. Thus, the trail consists of disjoint cycles and edges, where the isolated edges are those of multiplicity 2. The orientations on the edges are such that the cycles are oriented, and the isolated edges may be viewed as oriented cycles of length 2. Define an odd-cycle permutation as one whose trail contains at least one odd-length cycle, while even-cycle permutations have only even length cycles.

In each odd-cycle permutation σ, fix a canonical odd cycle C as follows; for each cycle, sort the list of vertex indices and use the sorted sequence of indices as label for that cycle; pick the odd cycle whose label is the lexicographically smallest. We can pair off the odd-cycle permutations by associating with such σ the unique odd-cycle permutation $-\sigma$ obtained by reversing the orientation of the edges in the canonical odd cycle C. Given these definitions, both σ and

$-\sigma$ have the same canonical odd cycle and $-(-\sigma) = \sigma$. The skew-symmetric nature of the matrix $\boldsymbol{B}$ implies that $\text{val}(\sigma) = -\text{val}(-\sigma)$, while the identical cycle structure of the two permutations implies that $\text{sgn}(\sigma) = \text{sgn}(-\sigma)$. It follows that their net contribution to $\det(\boldsymbol{B})$ is 0. Thus, the set of odd-cycle permutations has a net contribution of zero toward the value of $\det(\boldsymbol{B})$. This value of the determinant is completely determined by the value of the even-cycle permutations.

Notice that a permutation σ that corresponds to a perfect matching M has a trail consisting exactly of the set of edges in M, and each of these edges has multiplicity 2. Also, for any perfect matching M, the value of the permutation corresponding to it is exactly $(-1)^{n/2}2^{2w(M)}$, where $w(M)$ is the weight of the matching M. If these were the only even-cycle permutations to consider, the result would follow immediately. However, there are even-cycle permutations that do not correspond to any particular perfect matching, although as discussed below they can all be viewed as representing a union of two perfect matchings.

An even-cycle permutation σ consists of a collection of even cycles, and its trail can be partitioned into two perfect matchings, say M_1 and M_2, by considering alternating edges from each cycle.

Exercise 12.14: Verify that $|\text{val}(\sigma)| = 2^{w(M_1)+w(M_2)}$.

When the trail of σ has a cycle of length greater than 2, the two perfect matchings M_1 and M_2 are distinct and, since at most one of these two perfect matchings can be the unique perfect matching of minimum weight, it follows that $|\text{val}(\sigma)| > 2^{2W}$. On the other hand, when the trail has only cycles of length 2, i.e., the permutation corresponds to a perfect matching, we have $M_1 = M_2$ and $|\text{val}(\sigma)| = 2^{2w(M_1)} \geq 2^{2W}$. But note that equality with 2^{2W} is achieved only when $M_1 = M_2$ is the unique minimum weight perfect matching.

Thus, the *absolute* contribution to $\det(\boldsymbol{B})$ from each even-cycle permutation is a power of 2 no smaller than 2^{2W}. Moreover, exactly one of these contributions – the one from the even-cycle permutation corresponding to the unique minimum weight perfect matching – is equal to 2^{2W}. The determinant of $\boldsymbol{B}$ can now be viewed as a sum of powers of 2, possibly negated, such that the exponent of every term but one is strictly greater than $2W$. Since the term of absolute value 2^{2W} cannot cancel out, it follows that $\det(\boldsymbol{B}) \neq 0$ and in fact the largest power of 2 dividing it is 2^{2W}. □

Exercise 12.15: Observe that, after choosing the random weights, both $\boldsymbol{B}$ and $\det(\boldsymbol{B})$ can be computed via *NC* algorithms. Show that the value of W can also be determined in *NC*.

Of course, this only shows how to compute the weight of the minimum weight perfect matching. The following lemma is the basis for actually determining the edges in that matching. Recall that $\boldsymbol{B}^{ij}$ is the minor of $\boldsymbol{B}$ obtained by removing the ith row and the jth column from $\boldsymbol{B}$.

Lemma 12.12: *Let M be the unique minimum weight perfect matching in G, and let its weight be W. An edge (i, j) belongs to M if and only if*

$$\frac{\det(\boldsymbol{B}^{ij})2^{w_{ij}}}{2^{2W}}$$

is odd.

PROOF: Consider the matrix $\boldsymbol{Q}$ obtained from $\boldsymbol{B}$ by zeroing out each entry in the ith row and jth column of $\boldsymbol{B}$, except for B_{ij}. Notice that any permutation of non-zero value with respect to $\boldsymbol{Q}$ must map i to j.

Exercise 12.16: Verify that

$$\det(\boldsymbol{Q}) = (-1)^{i+j}2^{w_{ij}}\det(\boldsymbol{B}^{ij}) = \sum_{\sigma:\sigma(i)=j} \operatorname{sgn}(\sigma) \times \operatorname{val}(\sigma). \tag{12.1}$$

We can now apply the same argument as in Lemma 12.11 to claim that odd-cycle permutations (mapping i to j) will have a zero net contribution to the sum (12.1). One possible problem with doing so is that the canonical odd cycle in a specific permutation σ may contain the oriented edge going from i to j, in which case its partner $-\sigma$ will invert the orientation on that edge and hence not belong to the set of permutations mapping i to j. This will create problems in the canceling argument. However, note that since n is even, any odd-cycle permutation has at least two odd cycles and so we can choose the canonical cycle to be one not containing the edge from i to j.

If the edge (i, j) belongs to M, then (as before) exactly one even-cycle permutation contributes 2^{2W} to the sum and all others contribute a strictly larger power of 2. This implies that 2^{2W} is the largest power of 2 dividing the sum, and the remainder must be an odd integer. On the other hand, if (i, j) does not belong to M, all even-cycle permutations must contribute powers of 2 strictly larger than 2^{2W}, implying that the sum is divisible by 2^{2W+1} and the remainder of its division by 2^{2W} is an even number. □

It is now easy to determine all the edges in the minimum weight perfect matching M, and the algorithm is summarized below.

Algorithm Parallel Matching:

Input: Graph $G(V, E)$ with at least one perfect matching.

Output: A perfect matching $M \subseteq E$.

1. **for all** edges (i, j), in parallel **do**
   choose random weight w_{ij}.
2. **compute** the Tutte matrix $\boldsymbol{B}$ from w.
3. **compute** $\det(\boldsymbol{B})$.
4. **compute** W such that 2^{2W} is the largest power of 2 dividing $\det(\boldsymbol{B})$.
5. **compute** $\text{adj}(\boldsymbol{B}) = \det(\boldsymbol{B}) \times \boldsymbol{B}^{-1}$ whose (j, i) entry has absolute value $\det(\boldsymbol{B}^{ij})$.
6. **for all** edges (i, j) **do** (in parallel)
   compute $r_{ij} = \det(\boldsymbol{B}^{ij}) 2^{w_{ij}} / 2^{2W}$.
7. **for all** edges (i, j) **do** (in parallel)
   if r_{ij} is odd **then** add (i, j) to M

Exercise 12.17: Verify that each step of this algorithm can be implemented in ***RNC***, implying that it is an ***RNC*** algorithm for finding perfect matchings.

The most expensive computations in this algorithm are those of finding the determinant, inverse, and adjoint of an $n \times n$ matrix whose entries are $O(m)$-bit integers (since the matrix entries have magnitudes that are exponential in the edge weights).

Theorem 12.13: *Given a graph G with at least one perfect matching, the* **Parallel Matching** *algorithm finds a perfect matching with probability at least* $1/2$. *For a graph G with n vertices it requires* $O\left(\log^2 n\right)$ *time and* $O\left(n^{3.5} m\right)$ *processors.*

This is a Monte Carlo algorithm with (one-sided) error probability of $1/2$, and this probability can be reduced by repetitions. The only possible error arises when, even though the graph does have a perfect matching, the algorithm determines a set of edges that do not form a perfect matching because the random choice of weights did not yield a unique perfect matching. It is a simple matter to check for this error and convert this into a Las Vegas algorithm. Although we assumed throughout that the number of vertices n is even, it is possible to apply this algorithm to the case of odd n.

Exercise 12.18: In a graph $G(V, E)$ with n vertices, when n is odd we define a perfect matching to be a matching of cardinality $\lfloor n/2 \rfloor$. Explain how the **Parallel Matching** algorithm may be adapted to this case.

Finally, the **Parallel Matching** algorithm can be adapted to obtain a Las Vegas algorithm for finding a maximum matching, as outlined in Problems 12.16–12.18.

12.5. The Choice Coordination Problem

We now move on to distributed computation, in this section and in Section 12.6; we thus no longer use the PRAM model. A problem often arising in parallel and distributed computing is that of destroying the symmetry between a set of possibilities. This may be achieved by the use of randomization as in the case of the *Choice Coordination Problem* (CCP) discussed below. That this is a very "natural" problem is demonstrated by the following situation, which has been studied in biology. A particular class of mites (genus *Myrmoyssus*) reside as parasites on the ear membrane of the moths of family *Phaenidae*. The moths are prey to bats and the only defense they have is that they can hear the sonar used by an approaching bat. Unfortunately, if both ears of the moth are infected by the mites, then their ability to detect the sonar is considerably diminished, thereby severely decreasing the survival chances of both the moth and its colony of mites. The mites would like to ensure the continued survival of their host, and they can do so by infecting only one ear at a time. The mites are therefore faced with a "choice coordination problem": how does any collection of mites infecting a particular ear ensure that every other mite chooses the same ear? The protocol used by these mites involves leaving chemical trails around the ears of the moth.

Our interest in this abstract problem has a more computational motivation. Consider a collection of n identical processors that operate in total asynchrony. They have no global clock and no assumptions can be made about ther relative speeds. The processors have to reach a consensus on a unique choice out of a collection of m identical options. We use the following simple model of communication between the processors. There is a collection of m read-write registers accessible to all n processors. Several processors may simultaneously attempt to access or modify a register. To deal with such conflicts, we assume that the processors use a locking mechanism whereby a unique processor obtains sole access to a register when several processors attempt to access it; moreover, all the remaining processors then wait until the lock is released, and then contend once again for access to the register. The processors are required to run a protocol for picking a unique option out of the m choices. This is achieved by ensuring that at the end of the protocol exactly one of the m registers contains a special symbol $\surd$. The complexity of a choice coordination protocol is measured in terms of the total number of read and write operations performed by the n processors. (Clearly, running time has little meaning in an asynchronous situation.)

It is known that any deterministic protocol for solving this problem will have a complexity of $\Omega(n^{1/3})$ operations. We now illustrate the power of randomization

in this context by showing that there is a randomized protocol which, for any $c > 0$, will solve the problem using c operations with a probability of success at least $1 - 2^{-\Omega(c)}$. For simplicity we will consider only the case where $n = m = 2$, although the protocol and the analysis generalize in a rather straightforward manner.

We start by restricting ourselves to the rather simple case where the two processors are synchronous and operate in lock-step according to some global clock. The following protocol is executed by each of the two processors. We index the processors P_i and the possible choices by C_i for $i \in \{0,1\}$. The processor P_i initially scans the register C_i. Thereafter, the processors exchange registers after every iteration of the protocol. This implies that at no time will the two processors scan the same register. Each processor also maintains a local variable whose value is denoted by B_i.

Algorithm SYNCH-CCP:

Input: Registers C_0 and C_1 initialized to 0.

Output: Exactly one of the two registers has the value $\surd$.

0. P_i is initially scanning the register C_i and has its local variable B_i initialized to 0.

1. Read the current register and obtain a bit R_i.

2. Select one of three cases.

case: 2.1 [$R_i = \surd$]
halt;

case: 2.2 [$R_i = 0$, $B_i = 1$]
Write $\surd$ into the current register and **halt;**

case: 2.3 [otherwise]
Assign an unbiased random bit to B_i and write B_i into the current register;

3. P_i exchanges its current register with P_{1-i} and returns to Step 1.

To verify the correctness of this protocol it suffices to see that at most one register can ever have $\surd$ written into it. Suppose that both registers get the value $\surd$. We claim that both registers must have had $\surd$ written into them during the same iteration; otherwise, Case 2.1 will ensure that the protocol halts before this error takes place. Let us assume that the error takes place during the tth iteration. Denote by $B_i(t)$ and $R_i(t)$ the values used by processor P_i just after Step 1 of the tth iteration of the protocol. By Case 2.3, we know that $R_0(t) = B_1(t)$ and $R_1(t) = B_0(t)$. The only case in which P_i writes $\surd$ during the tth iteration is when $R_i = 0$ and $B_i = 1$; then, $R_{1-i} = 1$ and $B_{1-i} = 0$, and P_{1-i} cannot write $\surd$ during that iteration.

We have shown that the protocol terminates correctly by making a unique choice. But this assumes that the protocol terminates in a finite number

of iterations. Why should this happen? Notice that during each iteration, the probability that both the random bits B_0 and B_1 are the same is $1/2$. Moreover, if at any stage these two bits take on distinct values, then the protocol terminates within the next two stages. Thus, the probability that the number of stages exceeds t is $O(1/2^t)$. The computational cost of each iteration is bounded, so that this protocol does $O(t)$ work with probability $1-O(1/2^t)$.

We now generalize this protocol to the asynchronous case where the two processors may be operating at varying speeds and cannot "exchange" the registers after each iteration. In fact, we no longer assume that the two processors begin by scanning different registers – choosing a unique starting register C_0 or C_1 is in itself an instance of the choice coordination problem. Instead, we assume that each processor chooses its starting register at random. Thus, the two processors could be in a conflict at the very first step and must use the lock mechanism to resolve this conflict. The basic idea is to put time-stamps t_i on the register C_i, and T_i on the local variable B_i. We assume that a read operation on C_i will yield a pair $\langle t_i, R_i \rangle$, where t_i is the time-stamp and R_i is the value of that register. If the processors were to operate synchronously, these time-stamps would be exactly the same as the iteration number t of the previous protocol.

Algorithm ASYNCH-CCP:

Input: Registers C_0 and C_1 initialized to $\langle 0, 0 \rangle$.

Output: Exactly one of the two registers has the value $\surd$.

0. P_i is initially scanning a randomly chosen register. Thereafter, it changes its current register at the end of each iteration. The local variables T_i and B_i are initialized to 0.

1. P_i obtains a lock on the current register and reads $\langle t_i, R_i \rangle$.

2. P_i selects one of five cases.

Case 2.1: $[R_i = \surd]$
halt;

Case 2.2: $[T_i < t_i]$
$T_i \leftarrow t_i$ and $B_i \leftarrow R_i$.

Case 2.3: $[T_i > t_i]$
Write $\surd$ into the current register and **halt**;

Case 2.4: $[T_i = t_i,\ R_i = 0,\ B_i = 1]$
Write $\surd$ into the current register and **halt**;

Case 2.5: [otherwise]
$T_i \leftarrow T_i + 1$, $t_i \leftarrow t_i + 1$, assign a random (unbiased) bit to B_i and write $\langle t_i, B_i \rangle$ into the current register.

3. P_i releases the lock on its current register, moves to the other register, and returns to Step 1.

Theorem 12.14: *For any $c > 0$, Algorithm* **ASYNCH-CCP** *has total cost exceeding c with probability at most $2^{-\Omega(c)}$.*

PROOF: The only real difference from the previous protocol is in Cases 2.2 and 2.3. A processor in Case 2.2 is playing catch-up with the other processor, and the processor in Case 2.3 realizes that it is ahead of the other processor and is thus free to make the choice. To prove the correctness of this protocol, we consider the two cases where a processor can write $\surd$ into its current cell – these are Cases 2.3 and 2.4. Whenever a processor finishes an iteration, its personal time-stamp T_i equals that of the current register t_i. Further, $\surd$ cannot be written during the very first iteration of either processor.

Suppose P_i has just entered Case 2.3 with time-stamp T_i^* and its current cell is C_i with time-stamp t_i^*, where $t_i^* < T_i^*$. The only possible problem is that P_{1-i} may write (or already have written) $\surd$ into the register C_{1-i}. Suppose this error does indeed occur, and let t_{1-i}^* and T_{1-i}^* be the time-stamps during the iteration of P_{1-i} when it writes $\surd$ into C_{1-i}.

Now P_i comes to C_i with a time-stamp of T_i^*, and so it must have left C_{1-i} with a time-stamp of the same value *before* P_{1-i} could write $\surd$ into it. Since time-stamps cannot decrease, $t_{1-i}^* \geq T_i^*$. Moreover, P_{1-i} cannot have its time-stamp T_{1-i}^* exceeding t_i^*, since it must go to C_{1-i} from C_i and the time-stamp of that register never exceeds t_i. We have established that $T_{1-i}^* \leq t_i^* < T_i^* \leq t_{1-i}^*$. But P_{1-i} must enter Case 2.2 for $T_{1-i}^* < t_{1-i}^*$, contradicting the assumption that it writes $\surd$ into C_{1-i} for these values of the time-stamps.

Case 2.4 can be analyzed similarly, except that we finally obtain that $T_{1-i}^* \leq t_i^* = T_i^* \leq t_{1-i}^*$. This may cause a problem since it allows $T_{1-i}^* = t_{1-i}^*$, and so Case 2.4 can cause P_{1-i} to write $\surd$; however, we can now invoke the analysis of the synchronous case and rule out the possibility of error.

The complexity of this protocol is easy to analyze. The cost is proportional to the largest time-stamp obtained during the execution of this protocol. The time-stamp of a register can go up only in Case 2.5, and this happens only when Case 2.4 fails to apply. Moreover, the processor P_i that raises the time-stamp must have its current B_i value chosen during a visit to the other register. Thus, the analysis of the synchronous case applies. □

12.6. Byzantine Agreement

The subject of this section is the classic *Byzantine agreement problem* in distributed computation. As in Section 12.5, we study a process by which n processors reach an agreement. However, in the scenario we consider here, t of the n processors are faulty processors. We further assume that the faulty processors may collude in order to try and subvert the agreement process. A protocol designed to withstand such strong adversaries should certainly work in

the face of weaker faulty behavior arising in practice. The goal is a protocol that achieves agreement while tolerating as large a value of t as possible.

The Byzantine agreement problem is the following. Each of the n processors initially has a *value* that is 0 or 1; let b_i denote the value initially held by the ith processor. There are t faulty processors, and we refer to the remaining $n-t$ identical processors as *good* processors. Following communication according to the rules below, the ith processor ends the protocol with a *decision* $d_i \in \{0,1\}$, which must satisfy the following properties.

1. All the good processors should finish with the same decision.
2. If all the good processors begin with the same value v, then they all finish with their (common) decision equaling v.

The set of faulty processors is determined before the execution of the protocol begins (though of course the good processors do not know the identities of the faulty processors). The agreement protocol proceeds in a sequence of *rounds*. During each round, each processor may send one message to each other processor. Each processor receives a message from each of the remaining processors, before the following round begins. A processor need not send the same message to all the other processors. In the protocol described below, every message will be a single bit. All good processors follow the protocol exactly. A faulty processor may send messages that are totally inconsistent with the protocol, and may send different messages to different processors. In fact, we assume that the t faulty processors work in collusion: at the start of each round, they decide among themselves what messages each of them will send to each good processor, with the goal of inflicting the maximum damage. Agreement is achieved when every good processor has computed its decision consistent with the two properties above. We study the number of rounds it takes to achieve agreement.

It is known (see the Notes section) that any deterministic protocol for agreement in this model requires $t+1$ rounds. We now exhibit a simple randomized algorithm that terminates in a number of steps whose expectation is a constant. The number of rounds is a random variable, but the protocol is always correct in that it results in agreement as defined above; thus we have a Las Vegas protocol. We assume that at each step there is a *global coin toss* that a trusted party performs. The coin toss equiprobably results in a HEADS or a TAILS, and this result (denoted *coin*) is correctly conveyed to all the processors. This assumption can be dispensed with in more complicated protocols, but we do not discuss these here (see the Notes section).

For the remainder of the discussion, the reader may find it convenient to think of $t < n/8$; however, this is not a fundamental barrier, and the protocol in fact works for somewhat larger values of t. (This is the subject of Problem 12.27.) During each round of the protocol, each processor transmits a single bit, called its *vote*, to each other processor. A good processor sends the same vote to all other processors. Faulty processors may send arbitrary, inconsistent votes to good processors. Assume that n is a multiple of 8 for simplicity of exposition;

let L denote $(5n/8)+1$, H denote $(3n/4)+1$, and G denote $7n/8$. (In fact, the protocol only requires that $L \geq (n/2)+t+1$ and $H \geq L+t$ in order to work.) The ith processor executes the following routine, for $1 \leq i \leq n$.

Algorithm ByzGen:

Input: A value b_i.

Output: A decision d_i.

1. $vote = b_i$;

2. For each round, **do**

3. Broadcast *vote*;

4. Receive votes from all other processors;

5. *maj* ← majority value (0 or 1) among votes received including own vote;

6. *tally* ← number of occurrences of *maj* among votes received;

7. **if** *coin* = HEADS **then** *threshold* ← L;
else *threshold* ← H;

8. **if** *tally* ≥ *threshold* **then** *vote* ← *maj*;
else *vote* ← 0;

9. **if** *tally* ≥ G **then** set d_i to *maj* permanently;

We begin the analysis with an easy exercise:

Exercise 12.19: Show that if all the good processors begin a round with the same initial value, they all set their decisions to this value in a constant number of rounds.

The more interesting case for analysis is when the good processors do not all start with the same initial value. In the absence of faulty processors, a solution would be for all processors to broadcast their values, and then set their decisions to the majority of these values. The algorithm **ByzGen** implements this idea in the face of malicious faults.

If two good processors compute different values for *maj* in Step 5, *tally* does not exceed *threshold* regardless of whether L or H was chosen as *threshold*. Then, all good processors set *vote* = 0 in Step 8.2. As a result, all good processors set their decisions to 0 in the following round. It thus remains to consider the case when all good processors compute the same value for *maj* in Step 5.

We say that the faulty processors *foil* a threshold $x \in \{L, H\}$ in a round if, by sending different messages to the good processors, they cause *tally* to exceed x for at least one good processor, and to be no more than x for at least one good processor. Since the difference between the two possible thresholds L and H is

at least t, the faulty processors can foil at most one threshold in a round. Since the threshold is chosen equiprobably from $\{L, H\}$, it is foiled with probability at most $1/2$. Thus, the expected number of rounds before we have an unfoiled threshold is at most 2. If the threshold is not foiled, then all good processors compute the same value v for *vote* in Step 8. In the following round, every good processor receives at least $G > H > L$ votes for v, and sets *maj* to v in Step 5. Then, in Step 9, *tally* exceeds whichever threshold is chosen. When a good processor sets d_i the other good processors must have *tally* $\geq$ *threshold*, since $G \geq H + t$. Therefore they will all vote the same as d_i henceforth.

Theorem 12.15: *The expected number of rounds for* **ByzGen** *to reach agreement is a constant.*

The protocol **ByzGen** above does not include a termination criterion.

Exercise 12.20: Suggest a modification to the protocol **ByzGen** in which all good processors halt upon agreement.

Exercise 12.21: In the protocol **ByzGen**, is it always true that all good processors determine their decisions in the same round?

Notes

Karp and Ramachandran [241] give a comprehensive survey of PRAM algorithms. Some good references for parallel algorithms are the books by JáJá [208] and Leighton [271] and the volume edited by Reif [354]. The **BoxSort** algorithm of Section 12.2 is due to Reischuk [356]. Following Reischuk's work, a number of deterministic sorting algorithms running in $O(\log n)$ steps using n processors have been devised, most notably by Ajtai, Komlós, and Szemerédi [8] with later simplifications and improvements by Paterson [328]; Cole [110] gave a different deterministic parallel algorithm using n processors and $O(\log n)$ steps.

The intractability of the parallel solution of the LFMIS problem was established by Cook [111]. The first ***RNC*** algorithm for MIS is due to Karp and Wigderson [251]; they also provided a derandomized version of their algorithm. This was a complex algorithm requiring a large running time and a high processor count. The **Parallel MIS** algorithm and its derandomization is due to Luby [282]; this paper pioneered the idea of using random variables of limited independence to lead to a deterministic algorithm for a concrete problem (see also the Notes section of Chapter 3). Alon, Babai, and Itai [19] independently gave an ***RNC*** algorithm for the MIS problem and also derandomized it to obtain an ***NC*** algorithm. A more efficient ***NC*** algorithm was later provided by Goldberg and Spencer [173]. The paradigm of derandomizing parallel algorithms using limited independence has found a variety of applications. Luby [284] has combined it with the method of conditional probabilities (Section 5.6) to achieve processor efficiency for the maximal independent set problem. Berger and Rompel [55] and Motwani, Naor, and Naor [313] have used a combination of $\log n$-wise independence and the method of conditional probabilities to derive ***NC*** algorithms for a variety of problems. Karger and

Motwani [233] have used the combination of pairwise independence with the random walk technique for recycling random bits described in Chapter 6 to construct an *NC* algorithm for the min-cut problem. The min-cut problem is closely related to the matching problem – an *NC* algorithm for min-cut in directed graphs would result in an *NC* algorithm for maximum matching in bipartite graphs.

The reader may refer to the survey article by von zur Gathen [412] for a survey of parallel matrix algorithms. The first *NC* algorithm for matrix determinants is due to Csanky [115], but it applies only to fields of characteristic zero. Borodin, von zur Gathen, and Hopcroft [79] gave an *NC* algorithm for the general case (see Berkowitz [56] for a more elegant version). The algorithm due to Chistov [95] is currently the best known solution, and it requires only $O(\log^2 n)$ time. The computation of adjoints and inverses of a matrix can be reduced to the determinant computation at the cost of an increase in time and processor count. The randomized algorithm cited in Theorem 12.9 is due to Pan [323].

The book by Lovász and Plummer [281] is an excellent source for combinatorial and algorithmic results related to matchings, and Vazirani [405] surveys parallel matching algorithms. Section 7.8.3 gives a history of results establishing the connection between matchings and matrix determinants. Israeli and Shiloach [207] give an *NC* algorithm for finding maximal matchings. The *NC* algorithm in the case of a unique perfect matching is due to Rabin and Vazirani [348, 349], and in the case of polynomially small number of perfect matchings is due to Grigoriev and Karpinski [184]. The first *RNC* algorithm for matchings was given by Karp, Upfal, and Wigderson [242], and this was subsequently improved by Galil and Pan [162]. This work raised several interesting questions with respect to the parallel complexity of search versus decision problems, and this theme is explored by Karp, Upfal, and Wigderson [250]. The Isolating Lemma and the **Parallel Matching** algorithm are due to Mulmuley, Vazirani, and Vazirani [317]. These Monte Carlo algorithms were converted into Las Vegas algorithms by Karloff [237]. The best known deterministic algorithm using a polynomial number of processors, due to Goldberg, Plotkin, and Vaidya [172], requires $\Omega(n^{2/3})$ time. An interesting special case for which *NC* algorithms are known is that of finding perfect matchings in regular bipartite graphs. Lev, Pippenger, and Valiant [274] derived this result by providing an algorithm for edge coloring (which is a partition into matchings) a bipartite graph of maximum degree Δ with Δ colors. In the non-bipartite case, Karloff and Shmoys gave an *RNC* algorithm for approximate edge coloring, and this was derandomized by Berger and Rompel [55] and Motwani, Naor, and Naor [313]. Some interesting open problems are:

► **Research Problem 12.1:** Devise an *NC* algorithm for finding a maximum matching in a given graph.

► **Research Problem 12.2:** Devise an *NC* or an *RNC* algorithm for edge coloring a graph of maximum degree Δ using at most $\Delta+1$ colors (see Vizing's Theorem [71]).

► **Research Problem 12.3:** Aggarwal and Anderson [4] have shown that the problem of finding a depth-first search tree in a graph can be solved in *RNC* using *RNC* algorithms for finding maximum matchings; once again, the issue of an *NC* algorithm is unresolved.

The algorithm for the choice coordination problem in Section 12.5 is due to Rabin [344], and the biological analog is described in a paper by Treat [397]. The Byzantine agreement problem was introduced by Pease, Shostak and Lamport [330]. Fischer and Lynch [148] showed that in our model, any deterministic protocol requires $t+1$ rounds to reach agreement, in the worst case. This lower bound matches an upper bound given in [330]. The **ByzGen** protocol of Section 12.6 is due to Rabin [347]. Our presentation follows Chor and Dwork [96], who give a comprehensive account of the history of the problem, the various models under which it has been studied, and the many variants and improvements of Rabin's scheme. They point out that if the processors do not operate in synchrony, it is impossible to achieve agreement using a deterministic protocol; this result is due to Fischer, Lynch, and Paterson [149]. On the other hand, **ByzGen** and other randomized protocols can be shown to achieve agreement even in an asynchronous setting.

Problems

12.1 Show that the parallel variant of randomized quicksort described in Section 12.2 sorts n elements with n processors on a CREW PRAM, with high probability in $O(\log^2 n)$ steps.

12.2 Prove Lemma 12.1. The following outline is suggested (refer to Section 12.2 for the notation).

1. Bound the probability that $\alpha(B_{j+1}) = \alpha(B_j)$ using the result of Exercise 12.6.
2. Bound the probability that for any particular k, the value k is contained more than τ_k times in the sequence $\alpha(B_1), \ldots, \alpha(B_t)$.
3. Bound the probability that for $1 \leq k \leq c \log\log n$, the value k is contained more than τ_k times in the sequence $\alpha(B_1), \ldots, \alpha(B_t)$.

12.3 Suppose that the random samples in Stage 1 of **BoxSort** are chosen using pairwise independent, rather than completely independent random variables (the choices made by the various boxes are independent of each other, though). Derive the best upper bound you can on the number of parallel steps taken by **BoxSort**.

12.4 Using the ideas of Section 12.2, devise a CREW PRAM algorithm that selects the kth largest of n input numbers in $O(\log n)$ steps using $n/\log n$ processors. Assume that the n input numbers are initially located in global memory locations 1 through n.

12.5 Devise a *ZNC* algorithm for generating a random (uniformly distributed) permutation of a set S containing n elements. (**Hint:** Consider assigning random weights to the elements of S. If the weights are drawn from a sufficiently large set, each element will have a distinct weight.)

12.6 A *maximal* matching in a graph is a matching that is not properly contained in any other matching. Use the parallel algorithm for the MIS problem to devise an *RNC* algorithm for finding a maximal matching in a graph.

12.7 Consider a graph $G(V, E)$ with maximum degree Δ. Show that a sequential greedy algorithm will color the vertices of the graph using at most $\Delta+1$ colors such that no two adjacent vertices are assigned the same color. Employing the parallel algorithm for MIS, devise an ***RNC*** algorithm for finding a $\Delta+1$ coloring of a given graph.

12.8 (Due to M. Luby [282].) The *vertex partition* problem is defined as follows: given a graph $G(V, E)$ with edge weights, partition the vertices into sets V_1 and V_2 such that the net weight of the edges crossing the cut (V_1, V_2) is at least a half of the total weight of the edges in the graph. Describe an ***RNC*** algorithm for this problem, and explain how you will convert this into an ***NC*** algorithm using the idea of pairwise independence.

12.9 (Due to M. Luby [282].) In the **Parallel MIS** algorithm, suppose that the random marking of the vertices is only pairwise independent. Show that the probability that a good vertex belongs to $S \cup \Gamma(S)$ is at least $1/24$.

12.10 (Due to M. Luby [282].) Suppose that you are provided with a collection of n pairwise independent random numbers uniformly distributed over the set $\{0, 1, \ldots, p-1\}$, where $p \geq 2n$. It is desired to construct a collection of n pairwise independent Bernoulli random variables where the ith random variable should take on the value 1 with probability $1/t_i$, for $1 \leq t_i \leq n/8$. Show how you can achieve this goal approximately by constructing a collection of pairwise independent Bernoulli random variables such that the ith variable takes on the value 1 with probability $1/T_i$ where for a constant $c > 1$, T_i satisfies

$$T_i \leq t_i \leq cT_i.$$

12.11 (Due to M. Luby [282].) Combining the results of Problems 12.9 and 12.10, show that the **Parallel MIS** algorithm can be derandomized to yield an ***NC*** algorithm for the MIS problem. Note that the approach in Problem 12.10 will not work for marking vertices with degree exceeding $n/16$, and these will have to be dealt with separately.

12.12 (Due to M. Luby [282].) In this problem we consider a variant of the **Parallel MIS** algorithm. For each vertex $v \in V$, independently and uniformly choose a random weight $w(v)$ from the set $\{1, \ldots, n^4\}$. Repeatedly strip off an independent set S and its neighbors $\Gamma(S)$ from the graph G, where at each iteration the set S is the set of marked vertices generated by the following process: mark all vertices in V, and then in parallel for each edge in E unmark the end-point of larger weight. Show that this yields an ***RNC*** algorithm for MIS. Can this algorithm be derandomized using pairwise independence?

12.13 (Due to D.R. Karger [231].) Recall the randomized algorithm for min-cuts discussed in Section 1.1 (see also Section 10.2). Describe an ***RNC*** implementation of this algorithm. (**Hint:** While contracting the edges appears to be sequential process, it can be implemented in parallel using the following observation. Consider generating a random permutation on the edges, as described in Problem 12.5 and using this to determine the order in which the edges are contracted. The contraction algorithm will terminate at that point in the permutation where the preceding edges constitute a graph with

exactly two connected components. Assume that there is an NC algorithm for determining connected components.)

12.14 (Due to M. Luby, J. Naor, and M. Naor [285].) Using the idea of pairwise independence, construct an RNC algorithm for the min-cut problem that uses only a polylogarithmic number of random bits (see also Problem 12.13). What implications does this have for placing the min-cut problem in NC? (**Hint:** Select a set of edges by choosing each edge pairwise independently with probability $1/c$, where c is the size of the min-cut; see Problem 12.10. In parallel, contract all edges in this set. Repeat this process until the graph is reduced to two vertices.)

12.15 (Due to M.O. Rabin and V.V. Vazirani [349].) Let $G(V,E)$ be a graph with a unique perfect matching. Devise an NC algorithm for finding the perfect matching in G. (**Hint:** Consider substituting 1 for each indeterminate in the Tutte matrix. What is the significance of the entries in the adjoint of the Tutte matrix?)

12.16 (Due to K. Mulmuley, U.V. Vazirani, and V.V. Vazirani [317].) Consider the problem of finding a *minimum-weight* perfect matching in a graph $G(V,E)$, given edge-weights $w(e)$ for each edge $e \in E$ in *unary*. Note that it is not possible to apply the Isolating Lemma directly to this case since the random weights chosen there would conflict with the input weights. Explain how you would devise an RNC algorithm for this problem. The parallel complexity of the case where the edge-weights are given in *binary* is as yet unresolved – do you see why the RNC algorithm does not apply to the case of binary weights? (**Hint:** Start by scaling up the input edge weights by a polynomially large factor. Apply random perturbations to the scaled edge weights and prove a variant of the Isolating Lemma for this situation.)

12.17 (Due to K. Mulmuley, U.V. Vazirani, and V.V. Vazirani [317].) Devise an RNC algorithm for the problem of finding a *maximum* matching in a graph. Observe that the **Parallel Matching** algorithm does not work (as stated) when the maximum matching is not a perfect matching.

12.18 (Due to H.J. Karloff [237].) Suppose you are given a Monte Carlo RNC algorithm for finding a maximum matching in a bipartite graph. Explain how you would convert this into a Las Vegas algorithm. Can the solution be generalized to the case of non-bipartite graphs? (**Hint:** While this conversion is trivial for perfect matching algorithms, for maximum matching algorithms you will need to devise a parallel algorithm for determining an upper bound on the *size* of a maximum matching in a graph. This requires a non-trivial use of structure theorems for matchings in graphs.)

12.19 This problem explores a different method for converting the Monte Carlo maximum matching into a Las Vegas one. Recall from Problem 7.7 that the rank of the matrix of indeterminates constructed for a bipartite graph is exactly equal to the size of the maximum matching (a similar result holds for the general case). Consider the following approach for determining the size of the maximum matching: replace the indeterminates by random values and compute the rank of the resulting matrix. The rank of an integer matrix

can be computed in $\mathcal{NC}$, and one would hope that the random substitution method would preserve the rank with high probability. We would like to use this to verify that the matching algorithm is indeed producing the maximum matching, and thereby obtain a Las Vegas algorithm. Does this method work?

12.20 (Due to R.M. Karp, E. Upfal, and A. Wigderson [242].) In a bipartite graph $G(U, V, E)$, for any set $F \subseteq E$ define the *rank* $r(F)$ as the maximum size of intersection of F with a perfect matching, i.e., $r(F)$ is the largest number of edges in F that appear together in some perfect matching. Devise an $\mathcal{RNC}$ algorithm for computing the rank for any given set F. Can this be generalized to non-bipartite graphs?

12.21 (Due to R.M. Karp, E. Upfal, and A. Wigderson [242].) Assume you are given the algorithm from Problem 12.20. Using this, we will outline the construction of an alternative $\mathcal{RNC}$ algorithm for perfect matchings.

- Assuming that the input graph is sparse in that it has a total of n vertices and fewer than $3n/4$ edges, devise an $\mathcal{NC}$ algorithm for finding a large set S of edges that are guaranteed to belong to every perfect matching in G.
- Suppose now that the input graph has more than $3n/4$ edges. Using the rank algorithm, devise an $\mathcal{RNC}$ algorithm for finding a large set T of edges such that there exists a perfect matching in G none of whose edges belong to T.

Using the above tools, describe an alternative $\mathcal{RNC}$ algorithm for perfect matchings.

12.22 (Due to V.V. Vazirani [405].) Prove that the Isolating Lemma holds even when the weight of a set is defined to be the *product* (instead of *sum*) of the weights of its elements. Can you identify any general family of mappings from the weights of elements to the weights of sets for which the Isolating Lemma is guaranteed to be valid?

12.23 (Due to K. Mulmuley, U.V. Vazirani, and V.V. Vazirani [317].) An intriguing application of the Isolating Lemma is to the class of "uniqueness" problems, i.e., determining whether some problem in $\mathcal{NP}$ has a unique solution. Consider the following two problems, which take as input a graph $G(V, E)$ and a positive integer k:

CLIQUE: Determine whether the graph has a clique of size k.
UNIQUE CLIQUE: Determine whether there is *exactly one* clique of size k.

The complexity of unique solutions has been studied with respect to randomized reductions, which are the natural generalization of polynomial time reductions to allowing randomized polynomial time reductions. Devise a randomized polynomial time reduction from the CLIQUE problem to the UNIQUE CLIQUE.

12.24 (Due to J. Naor.) Let $G(V, E)$ be an unweighted, undirected graph with n vertices and m edges. Under any weight function $w : E \to \{0, \ldots, W\}$, the

length of a path in G is the sum of the weights of the edges in that path. A weight function is said to be *good* if the following two conditions hold for each vertex $x \in V$.

1. For all vertices $y \in V$, the shortest path from x to y is unique.
2. For any pair of vertices $y, z \in V$, the net weight of the shortest path from x to y is different from the net weight of the shortest path from x to z.

What is the smallest value of W (as a function of n and m) for which you can guarantee the *existence* of a good weight assignment?

12.25 (Due to K. Mulmuley, U.V. Vazirani, and V.V. Vazirani [317].) An even more intriguing application of the Isolating Lemma is to the *Exact Matching* problem – given a graph $G(V,E)$ with a subset of edges $R \subseteq E$ colored red, and a positive integer k, determine whether there is a perfect matching using exactly k red edges. This problem is not known to be in $\boldsymbol{P}$, but can be shown to be in $\boldsymbol{RNC}$ via a (non-trivial) application of the Isolating Lemma. Devise $\boldsymbol{RNC}$ algorithms for the decision and search versions of this problem.

12.26 (Due to M.O. Rabin [344].) Show that Algorithm **ASYNCH-CCP** works equally well in the case where the numbers of processors and choices are both greater than 2. How does the complexity depend on the number of processors and choices?

12.27 How large a value of t can the **ByzGen** algorithm tolerate? (Modify the parameters L, H, and G if necessary.)

12.28 Consider what happens if the outcome of the coin toss generated by the trusted party in the **ByzGen** algorithm is corrupted before it reaches some good processors.

(a) Can disagreement occur if different good processors see different outcomes? What happens if, instead of a global coin toss, each processor chooses a random coin independently of other processors, at every round?

(b) Suppose that we were guaranteed that at least H good processors receive the correct outcome of each coin toss. Give a modification for the protocol **ByzGen** that achieves agreement in an expected constant number of rounds, under this assumption.

30.2-2
Give an EREW algorithm for FIND-ROOTS that runs in $O(\lg n)$ time on a forest of n nodes.

30.2-3
Give an n-processor CRCW algorithm that can compute the OR of n boolean values in $O(1)$ time.

30.2-4
Describe an efficient CRCW algorithm to multiply two $n \times n$ boolean matrices using n^3 processors.

30.2-5
Describe an $O(\lg n)$-time EREW algorithm to multiply two $n \times n$ matrices of real numbers using n^3 processors. Is there a faster common-CRCW algorithm? Is there a faster algorithm in one of the stronger CRCW models?

30.2-6 $\star$
Prove that for any constant $\epsilon > 0$, there is an $O(1)$-time CRCW algorithm using $O(n^{1+\epsilon})$ processors to find the maximum element of an n-element array.

30.2-7 $\star$
Show how to merge two sorted arrays, each with n numbers, in $O(1)$ time using a priority-CRCW algorithm. Describe how to use this algorithm to sort in $O(\lg n)$ time. Is your sorting algorithm work-efficient?

30.2-8
Complete the proof of Theorem 30.1 by describing how a concurrent read on a p-processor CRCW PRAM is implemented in $O(\lg p)$ time on a p-processor EREW PRAM.

30.2-9
Show how a p-processor EREW PRAM can implement a p-processor combining-CRCW PRAM with only $O(\lg p)$ performance loss. (*Hint:* Use a parallel prefix computation.)

30.3 Brent's theorem and work efficiency

Brent's theorem shows how we can efficiently simulate a combinational circuit by a PRAM. Using this theorem, we can adapt many of the results for sorting networks from Chapter 28 and many of the results for arithmetic circuits from Chapter 29 to the PRAM model. Readers unfamiliar with combinational circuits may wish to review Section 29.1.

A ***combinational circuit*** is an acyclic network of ***combinational elements***. Each combinational element has one or more inputs, and in this section, we shall assume that each element has exactly one output. (Combinational elements with $k > 1$ outputs can be considered to be k separate elements.) The number of inputs is the ***fan-in*** of the element, and the number of places to which its output feeds is its ***fan-out***. We generally assume in this section that every combinational element in the circuit has bounded ($O(1)$) fan-in. It may, however, have unbounded fan-out.

The ***size*** of a combinational circuit is the number of combinational elements that it contains. The number of combinational elements on a longest path from an input of the circuit to an output of a combinational element is the element's ***depth***. The ***depth*** of the entire circuit is the maximum depth of any of its elements.

Theorem 30.2 (Brent's theorem)
Any depth-d, size-n combinational circuit with bounded fan-in can be simulated by a p-processor CREW algorithm in $O(n/p + d)$ time.

Proof We store the inputs to the combinational circuit in the PRAM's global memory, and we reserve a location for each combinational element in the circuit to store its output value when it is computed. A given combinational element can then be simulated by a single PRAM processor in $O(1)$ time as follows. The processor simply reads the input values for the element from the values in memory corresponding to circuit inputs or element outputs that feed it, thereby simulating the wires in the circuit. It then computes the function of the combinational element and writes the result in the appropriate position in memory. Since the fan-in of each circuit element is bounded, each function can be computed in $O(1)$ time.

Our job, therefore, is to find a schedule of the p processors of the PRAM such that all combinational elements are simulated in $O(n/p+d)$ time. The main constraint is that an element cannot be simulated until the outputs from any elements that feed it have been computed. Concurrent reads are employed whenever several combinational elements being simulated in parallel require the same value.

Since all elements at depth 1 depend only on circuit inputs, they are the only ones that can be simulated initially. Once they have been simulated, all elements at depth 2 can be simulated, and so forth, until we finish with all elements at depth d. The key idea is that if all elements from depths 1 to i have been simulated, we can simulate any subset of elements at depth $i+1$ in parallel, since their computations are independent of one another.

Our scheduling strategy, therefore, is quite naive. We simulate all the elements at depth i before proceeding to simulate those at depth $i + 1$. Within a given depth i, we simulate the elements p at a time. Figure 30.8 illustrates such a strategy for $p = 2$.

Let us analyze this simulation strategy. For $i = 1, 2, \ldots, d$, let n_i be the number of elements at depth i in the circuit. Thus,

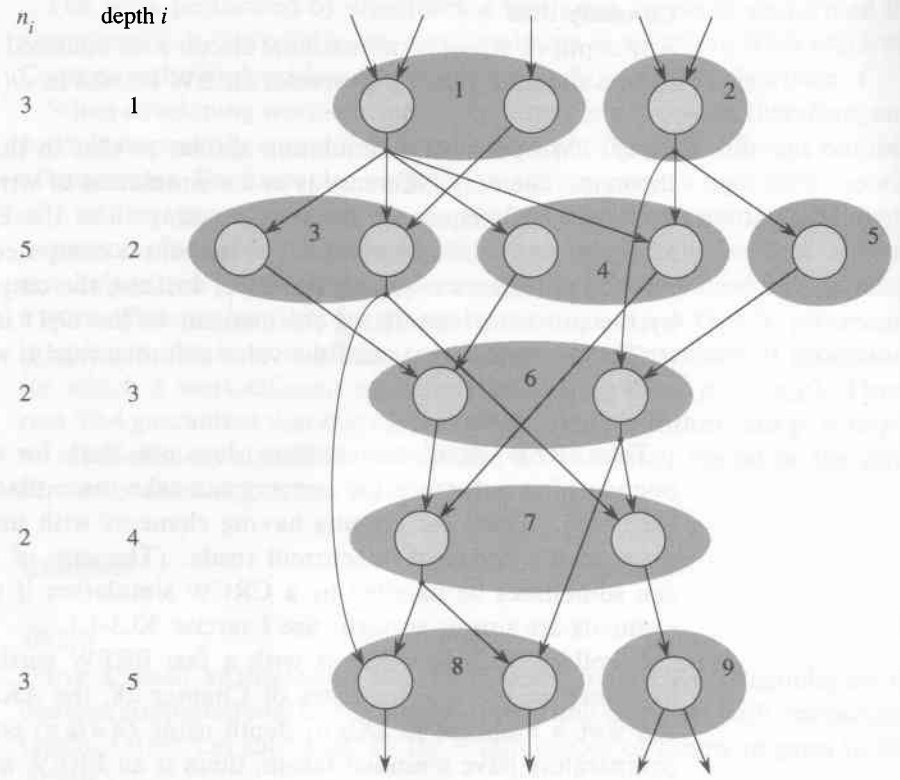

Figure 30.8 Brent's theorem. The combinational circuit of size 15 and depth 5 is simulated by a 2-processor CREW PRAM in $9 \leq 15/2 + 5$ steps. The simulation proceeds from top to bottom through the circuit. The shaded groups of circuit elements indicate which elements are simulated at the same time, and each group is labeled with a number corresponding to the time step when its elements are simulated.

$$\sum_{i=1}^{d} n_i = n \ .$$

Consider the n_i combinational elements at depth i. By grouping them into $\lceil n_i/p \rceil$ groups, where the first $\lfloor n_i/p \rfloor$ groups have p elements each and the leftover elements, if any, are in the last group, the PRAM can simulate the computations performed by these combinational elements in $O(\lceil n_i/p \rceil)$ time. The total simulation time is therefore on the order of

$$\begin{aligned} \sum_{i=1}^{d} \left\lceil \frac{n_i}{p} \right\rceil &\leq \sum_{i=1}^{d} \left(\frac{n_i}{p} + 1 \right) \\ &= \frac{n}{p} + d \ . \end{aligned}$$

Brent's theorem can be extended to EREW simulations when a combinational circuit has $O(1)$ fan-out for each combinational element.

Corollary 30.3
Any depth-d, size-n combinational circuit with bounded fan-in and fan-out can be simulated on a p-processor EREW PRAM in $O(n/p + d)$ time.

Proof We perform a simulation similar to that in the proof of Brent's theorem. The only difference is in the simulation of wires, which is where Theorem 30.2 requires concurrent reading. For the EREW simulation, after the output of a combinational element is computed, it is not directly read by processors requiring its value. Instead, the output value is copied by the processor simulating the element to the $O(1)$ inputs that require it. The processors that need the value can then read it without interfering with each other. ■

This EREW simulation strategy does not work for elements with unbounded fan-out, since the copying can take more than constant time at each step. Thus, for circuits having elements with unbounded fan-out, we need the power of concurrent reads. (The case of unbounded fan-in can sometimes be handled by a CRCW simulation if the combinational elements are simple enough. See Exercise 30.3-1.)

Corollary 30.3 provides us with a fast EREW sorting algorithm. As explained in the chapter notes of Chapter 28, the AKS sorting network can sort n numbers in $O(\lg n)$ depth using $O(n \lg n)$ comparators. Since comparators have bounded fan-in, there is an EREW algorithm to sort n numbers in $O(\lg n)$ time using n processors. (We used this result in Theorem 30.1 to show that an EREW PRAM can simulate a CRCW PRAM with at most logarithmic slowdown.) Unfortunately, the constants hidden by the O-notation are so large that this sorting algorithm has solely theoretical interest. More practical EREW sorting algorithms have been discovered, however, notably the parallel merge-sorting algorithm due to Cole [46].

Now suppose that we have a PRAM algorithm that uses at most p processors, but we have a PRAM with only $p' < p$ processors. We would like to be able to run the p-processor algorithm on the smaller p'-processor PRAM in a work-efficient fashion. By using the idea in the proof of Brent's theorem, we can give a condition for when this is possible.

Theorem 30.4
If a p-processor PRAM algorithm A runs in time t, then for any $p' < p$, there is an p'-processor PRAM algorithm A' for the same problem that runs in time $O(pt/p')$.

Proof Let the time steps of algorithm A be numbered $1, 2, \ldots, t$. Algorithm A' simulates the execution of each time step $i = 1, 2, \ldots, t$ in time $O(\lceil p/p' \rceil)$. There are t steps, and so the entire simulation takes time $O(\lceil p/p' \rceil t) = O(pt/p')$, since $p' < p$. ■

The work performed by algorithm A is pt, and the work performed by algorithm A' is $(pt/p')p' = pt$; the simulation is therefore work-efficient. Consequently, if algorithm A is itself work-efficient, so is algorithm A'.

When developing work-efficient algorithms for a problem, therefore, one needn't necessarily create a different algorithm for each different number of processors. For example, suppose that we can prove a tight lower bound of t on the running time of any parallel algorithm, no matter how many processors, for solving a given problem, and suppose further that the best serial algorithm for the problem does work w. Then, we need only develop a work-efficient algorithm for the problem that uses $p = \Theta(w/t)$ processors in order to obtain work-efficient algorithms for all numbers of processors for which a work-efficient algorithm is possible. For $p' = o(p)$, Theorem 30.4 guarantees that there is a work-efficient algorithm. For $p' = \omega(p)$, no work-efficient algorithms exist, since if t is a lower bound on the time for any parallel algorithm, $p't = \omega(pt) = \omega(w)$.

Exercises

30.3-1

Prove a result analogous to Brent's theorem for a CRCW simulation of boolean combinational circuits having AND and OR gates with unbounded fan-in. (*Hint:* Let the "size" be the total number of inputs to gates in the circuit.)

30.3-2

Show that a parallel prefix computation on n values stored in an array of memory can be implemented in $O(\lg n)$ time on an EREW PRAM using $O(n/\lg n)$ processors. Why does this result not extend immediately to a list of n values?

30.3-3

Show how to multiply an $n \times n$ matrix A by an n-vector b in $O(\lg n)$ time with a work-efficient EREW algorithm. (*Hint:* Construct a combinational circuit for the problem.)

30.3-4

Give a CRCW algorithm using n^2 processors to multiply two $n \times n$ matrices. The algorithm should be work-efficient with respect to the normal $\Theta(n^3)$-time serial algorithm for multiplying matrices. Can you make the algorithm EREW?

30.3-5

Some parallel models allow processors to become inactive, so that the number of processors executing at any step varies. Define the work in this model as the total number of steps executed during an algorithm by active processors. Show that any CRCW algorithm that performs w