CUCS-326-88

Parallel Algorithmic Techniques for Combinatorial Computation

David Eppstein [1,3] **and Zvi Galil** [1,2,3]

[1] Computer Science Department, Columbia University, New York NY 10027
[2] Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel
[3] Work supported in part by NSF grants DCR-85-11713 and CCR-86-05353

February 15, 1988

INTRODUCTION

Parallel computation offers the promise of great improvements in the solution of problems that, if we were restricted to sequential computation, would take so much time that solution would be impractical.

There is a drawback to the use of parallel computers, however, and that is that they seem to be harder to program. For this reason, parallel algorithms in practice are often restricted to simple problems such as matrix multiplication. Certainly this is useful, and in fact we shall see later some non-obvious uses of matrix manipulation, but many of the large problems requiring solution are of a more complex nature. In particular, an instance of a problem may be structured as an arbitrary graph or tree, rather than in the regular order of a matrix.

In this paper we describe a number of algorithmic techniques that have been developed for solving such combinatorial problems. The intent of the paper is to show how the algorithmic tools we present can be used as building blocks for higher level algorithms, and to present pointers to the literature for the reader to look up the specifics of these algorithms. We make no claim to completeness; a number of techniques have been omitted for brevity or because their chief application is not combinatorial in nature. In particular we give very little attention to parallel sorting, although sorting is used as a subroutine in a number of the algorithms we describe. We also only describe algorithms, and not lower bounds, for solving problems in parallel.

The model of parallelism used in this paper is that of the PRAM, or shared memory machine. This model is commonly used by theoretical computer scientists but less often by builders of actual parallel machines; nevertheless it is hoped that the techniques described here will be useful not only on the shared memory machines that have been built, but also on other types of parallel computers, either through simulations of the shared memory model on those computers [Mehlhorn and Vishkin 1984; Vishkin 1984a; Karlin and Upfal 1986; Ranade 1987], or through analogues of these techniques for different models of parallel computation (e.g. see Hillis and Steele [1986]).

This paper is organized as follows. In the next section we give a brief description of the PRAM model and the motivation for its use. Then we describe prefix computation and ranking, two tools for performing calculations on data arranged in some linear order. In the third section we describe methods for finding such an order when the data are arranged in a graph or tree. Finally we discuss ways of solving problems using parallel matrix computations.

All logarithms in this paper should be assumed to be base 2 unless explicitly stated otherwise.

1. THE MODEL OF PARALLELISM

The model of parallel computation we will for the most part be using in this paper is the shared memory parallel random access machine, known for short as the PRAM. In this model, one is given an arbitrarily large collection of identical processors and a separate collection of memory cells; any processor can access any memory cell in unit time. The processors are assumed to know the size of their input, but the program controlling the machine is the same for all sizes of input.

There are two motivations for this model, practical and theoretical.

The theoretical motivation is based on another model, that of the circuit. An algorithm in this model consists of a family of Boolean circuits, one for each size of a problem instance. To correspond with the PRAM requirement of having one program for all input sizes, the circuit family is usually required to satisfy a *uniformity condition*. Such a condition states that there must be a program meeting certain conditions (for instance taking polynomial time, or running in logarithmic space) which, given as input a number, produces as output the circuit having that number as its input size.

The output of the algorithm for a given instance of a problem consists of the result of the circuit of the appropriate size, when the circuit's input lines are set to some binary representation of the instance. The two measures of an algorithm's complexity in this model are the circuit's size and depth, both as a function of input size.

Circuit models are good for producing theoretical lower bounds on the complexity of a problem, because a gate in a Boolean circuit is much simpler than an entire processor in a parallel computer, and because most other models of parallelism can be described in terms of circuits. But it is difficult to construct algorithms in the circuit model, again because of the simplicity of the circuits.

It turns out that any uniform circuit family can be described as a PRAM algorithm, by simulating each gate of the circuits with a processor of the PRAM. The number of processors used is proportional to the size of the circuit, and the parallel time is proportional to the circuit's depth. Similarly, a PRAM algorithm may be simulated by a circuit family of size proportional to the total number of PRAM operations, and of depth proportional to the parallel time, with perhaps an extra logarithmic factor in the size and depth due to the fact that PRAM operations are performed on numbers rather than single bits.

Therefore if a problem can be solved by a PRAM algorithm with a number of processors polynomial in the size of the input, and a time polynomial in the logarithm of the input size, it can be solved by a polynomial sized circuit with polylogarithmic depth, and vice versa. The class of problems solved in such bounds is called NC; the theoretical importance of this class is due to the fact that it does not depend on whether one is using the PRAM or the circuit as a model, or on details such as whether the circuit is allowed unbounded fan-in or whether the PRAM is allowed simultaneous access to the same memory cell by multiple processors.

Practically, one is given a machine which is capable neither of configuring itself into arbitrary circuits nor of accessing memory in unit time, and one would like an algorithm that takes a total number of operations that is close to the best possible sequential time, but with a non-constant factor speedup over that sequential time. It is in many cases possible to simulate a PRAM algorithm on a real parallel machine, with only a logarithmic loss in time for the simulation. In particular such simulations are known for the butterfly network [Karlin and Upfal 1986; Ranade 1987], and for a more specialized combination of sorting and merging networks [Mehlhorn and Vishkin 1984; Vishkin 1984a]. In both cases the interconnection network has bounded degree.

Many combinatorial problems can be solved on the PRAM using algorithms that fulfill both the theoretical requirement of taking polylogarithmic time, and the practical requirement of taking a number of operations within a polylogarithmic factor of the best known sequential algorithm. For a number of other problems, the best known NC algorithm takes a number of processors that is a small polynomial of the input size, and so for small problem instances the parallel solution may still be practical. In this paper we restrict our attention to NC algorithms, but we attempt to keep the number of processors used to a small enough number that the algorithms remain useful in practice.

1.1 Details of the PRAM Model

We assume that each processor of the PRAM is arbitrarily assigned a unique identifier, which can fit in a single memory cell; we also assume that memory addresses can fit in memory cells. Each processor can access any memory cell in unit time. In some cases we add the further restriction that a memory cell can contain only a number bits of information proportional to the logarithm of the size of the input (note that there must be at least that many bits to meet our earlier restriction that addresses fit in a cell). Processors are allowed to perform in constant time any other operation that a typical sequential processor could do; this usually means processors are restricted to arithmetic and bit vector operations.

The input to a PRAM algorithm is given in a predetermined fashion in the memory cells of the processor. We assume that the size of a given input instance (which may be represented by more than one parameter; e.g. the numbers of vertices and edges in a graph) is known. The number of processors used for a given computation may vary depending on the size of the instance, but may not change during the computation, nor may it in any other way depend on the actual contents of the problem instance.

The efficiency of a PRAM algorithm is measured by two criteria, taken as functions of the instance size: the number of processors used to perform the computation, and the amount of time taken by that number of processors for the computation. The latter criterion is usually taken to be the worst case over all instances of a given size. Another criterion, the total number of operations, is then the product of the number of processors with the total time taken; that is, we count a processor as performing an operation even when it is idle. We later describe a way of performing a parallel computation such that at most a fixed fraction of the time is spent idle; this means that we could instead count non-idle operations only, and take the number of processors to be the number of operations divided by the time.

Since a parallel algorithm can easily be simulated on a sequential computer in time proportional to the number of parallel operations, it is clear that conversely the number of operations of a parallel algorithm must be at least proportional to the best known sequential algorithm. A parallel algorithm that meets this bound is called optimal. Optimality depends on the specifics of the model we are using, because as we shall see simulation of an algorithm designed for one model on a processor of a different model may lose a polylogarithmic factor in the number of operations. Many of the algorithms we will consider are in fact optimal for some model.

There is an important question we have not yet addressed, which is what happens when more than one processor tries to access the same memory cell at a given time. There are various possible answers that have been chosen; it will be convenient to pick different answers for different algorithms. This does not mean that the algorithms will only work for a specific choice of concurrent memory access behavior, because, as we shall see, there are a number of simulation results that let algorithms written for one choice work with machines that use a different choice.

There are three main choices for processor action on concurrent memory access. The first choice is simply to disallow such access altogether, and leave undefined the behavior of a program that attempts to perform such access. Such a machine is called exclusive read exclusive write, or EREW for short. This is the weakest of the various types of PRAM, and so an algorithm that works on this model with a given time and number of processors is preferable to one that achieves the same time and processors on a stronger model.

The second type of PRAM, known as a concurrent read exclusive write machine, or CREW for short, allows several processors to read the same cell at once, but disallows multiple concurrent writes to a cell. Again, the behavior of a program that violates these constraints is undefined. For a long time this seems to have been the only type of PRAM used [Kučera 1982]; it appeared in the literature as early as 1976 [Csanky 1976; Hirschberg 1976].

In third and strongest version of the PRAM, the concurrent read concurrent write machine or CRCW, both concurrent reads and concurrent writes are allowed. We still need to define what happens when several processors write to a single cell with different data values. A number of different possibilities have appeared in the literature under various names; these are summarized as follows.

(1) Weak CRCW. In this model concurrent writes are allowed only if all processors performing a concurrent write are writing the value zero. Sometimes the further restriction is added that they be writing to one of a set of special cells that can only contain zero or one; but this restricted weak CRCW is equivalent to the version described here, as we shall show in the simulations below. This model was introduced by Kučera [1982], who showed that certain problems could be solved more quickly on it than on the CREW which had typically been used before then, and that (as we shall see below) problems that can be solved in a given time on any version of the CRCW can be solved within a constant factor of that time on this version (but perhaps with many more processors).

(2) Common-mode CRCW. In this model there are no restrictions on the values written in a concurrent write; however all processors writing at a given time to the same cell must write the same value. Therefore there is no question of which value is actually written to the cell.

(3) Arbitrary-winner CRCW. Processors may write different values to the same cell, and one of the values written is the value the cell ends up containing. But which processor wrote the winning value may be chosen arbitrarily; in particular the result may be different if the same step is executed another time. This version of the CRCW is the one most commonly used by workers in the area of parallel algorithms.

(4) Priority CRCW. The result written in a concurrent write is the value from the writer with the largest processor identifier. Recall that these identifiers may be chosen arbitrarily; thus

like the arbitrary-winner CRCW it is not clear a priori which processor will be the one to get its value chosen as the result; however unlike the arbitrary-winner CRCW a repetition of the same concurrent write will give the same result.

(5) Strong CRCW. This model is less often seen than the previous ones, but it has appeared in the literature [Galil 1986a; Cole and Vishkin 1986c]. In it, the value written in any concurrent write is the largest (or equivalently smallest) of any of the values the processors are attempting to write.

Note that the strong CRCW model can simulate each step of a priority CRCW in constant time by, in a concurrent write, first writing the identifiers of the processors attempting the write, and then the winner writing the original value it wanted written. Thus it is at least as strong as the priority CRCW. Similarly we can see that each of the models we have given is at least as strong as the previous ones, so the various models form a hierarchy. Grolmusz and Ragde [1987] have recently described some other CRCW models which are intermediate in power among the ones we have listed, but which do not fall into a hierarchy with them.

Vishkin [1984b] has described an even stronger version of the CRCW PRAM. In this model, which has been partially implemented as the NYU Ultracomputer [Schwartz 1980], there is a further kind of concurrent write available. If one processor performs this write to a cell, the value it was writing is added to the value at the cell, and the result is stored back into the cell. The old cell value, before the addition, is returned to the writing processor. If several processors write to the same cell, the effect is as if they had performed the above operation one at a time, in some arbitrary order. One could further generalize this to any (associative) binary operation, rather than just addition. Vishkin showed that in certain simulations of one parallel model by another, this very strong write operation can also be simulated with no further loss of time or efficiency.

A final distinction to be made among PRAM models is whether they are randomized or deterministic. If we allow randomness in parallel computation, it is performed as follows. Each processor is given its own random number generator, with which it can generate in constant time a number drawn from a uniform distribution over the numbers fitting in a memory cell. These numbers are private; if a processor wants to share its random number with another processor it must write it to a memory cell. The distribution of numbers drawn at any one processor is usually taken to be independant of that drawn at all other processors, although some algorithms have weaker independance requirements.

When a parallel algorithm uses random numbers, we still require that the number of processors be a fixed function of the input size; that is, only the run time, and not the number of processors, may vary randomly. The algorithm may always terminate with the correct answer (such an algorithm is commonly known as a Las Vegas algorithm) or it may get the correct answer with high probability (a Monte Carlo algorithm). The algorithm must be able to detect when it has terminated. The time we measure for a randomized parallel algorithm is then the worst case expected time of termination over all instances of a given size.

Random numbers are used in a number of parallel algorithms for which there are efficient deterministic sequential solutions. One important use of them is for what is known as symmetry breaking; for instance, one may have a graph in which many nodes have similar local neighborhoods, but in which after the algorithm terminates different nodes must have different data values; one such problem is that of coloring the graph. If we used a deterministic algorithm, the nodes must look far away from themselves to see enough to differentiate them from their neighbors, but in a parallel algorithm they might merely flip coins and expect with high probability to get a different result. For certain problems these coin flips can be replaced by a use of the processor identifiers known as deterministic coin tossing, introduced by Cole and Vishkin [1986a, 1986b]; this technique will be discussed later.

There are several ways in which randomness can improve the parallel time and processor bounds of an algorithm. First, a randomized parallel algorithm may solve in polylogarithm expected time a problem for which there is no known NC algorithm. We call the class of problems solved by such algorithms RNC. Such is the case for matching [Karp et al. 1985a; Galil and Pan 1985a; Mulmuley et al. 1987] and for the construction of maximal paths [Anderson 1985] and of depth first search trees [Aggarwal and Anderson 1987]. Karp et al. [1985b] describe a general model of parallel computation in the presence of oracles in which they give a problem which they prove that no deterministic PRAM with polynomial processors can solve in polylogarithmic parallel time, but which can be solved quickly and efficiently in parallel using a probabilistic algorithm.

Second, a randomized parallel algorithm may perform a task that can be performed deterministically in parallel, but the parallel algorithm may be more efficient than the best known deterministic algorithm, may take less time than it, or may run on a weaker model of parallelism. Even if the randomized algorithm takes the same asymptotic bounds, it may be simpler and therefore more practical than its deterministic counterpart. Examples of improved efficiency in randomized parallel computation are the various algorithms known for finding maximal independent sets; the randomized algorithm of Luby [1985] uses less operations than the deterministic algorithm of the same paper, and runs more quickly (with the same number of processors) than the newer deterministic algorithm of Goldberg and Spencer [1987].

Other examples of randomized parallel algorithms that are more efficient than their deterministic counterparts include those for finding connected components [Gazit 1986], and sorting integers [Reif 1985]. Another example in which the randomized parallel time is faster than the best known deterministic time for a given number of processors (in fact the time is better than the deterministic lower bound) is the constant time selection algorithm given by Reischuk [1981]; note however that both Reischuk's algorithm and the lower bounds given are for the comparison tree model of parallelism, and therefore do not translate directly to the PRAM.

Finally, a randomized algorithm may use the same asymptotic time and number of processors as a deterministic algorithm, but may be much simpler. Such an algorithm would typically be more useful in a practical implementation, because of the ease of writing the program for it and also because these simpler programs tend to run faster (by a constant factor, but sometimes a very large constant factor) than their deterministic counterparts. Also the randomized algorithms are often discovered some time earlier than an equally efficient deterministic solution. One such instance is the randomized EREW list ranking algorithm of Vishkin [1984c]; this algorithm was later made deterministic [Cole and Vishkin 1986a, 1986b] and still later made faster with the same asymptotic number of operations [Cole and Vishkin 1986c].

1.2 Simulations Among PRAM Models

We have looked at a number of different ways that the PRAM model of parallel computation may have its details of concurrent memory access, and of randomness or determinism, filled out. All of the different types of deterministic PRAM we have seen have been used in the development of parallel algorithms to solve various problems, and the same is true for many probabilistic PRAM models. An algorithm designed for a weak model can be used in a stronger model within the same processor and time bounds as they would take in the weaker model. But we would like to use any PRAM algorithms on any model, and so it is important be able to simulate efficiently stronger models by weaker ones.

We will give a number of these simulation results between different pairs of models, but before we do, let us first see a result in which the model being simulated and the model doing the simulation are the same. That is, if we have a parallel algorithm running on a certain PRAM model with a given number of processors and within a given amount of time, we would like to be able to run it with fewer processors without more than a proportional loss of time.

Theorem 1 [Brent 1974]. If a parallel computation can be performed in time t, using q operations on any number of PRAM processors (of a given fixed type), then it can be performed in time $t + (q - t)/p$ using p processors (of the same type).

Proof: Suppose that at each time step i, taking i from 1 to t, the original computation performs q_i operations, so that $q = \sum_{i=1}^{t} q_i$. Then with p processors, we can simulate step i in time $\lceil q_i/p \rceil = \lfloor (q_i + p - 1)/p \rfloor \leq (q_i + p - 1)/p$, by breaking the operations into blocks of size p and performing one block at a time, with one processor per operation within each block. Summing gives total time $\sum_{i=1}^{t} \lceil q_i/p \rceil \leq \sum_{i=1}^{t} (q_i + p - 1)/p = t + (q - t)/p$. •

Corollary. A parallel computation that can be performed in time t, using p processors, can be performed in any amount of time greater than t and less than pt using $O(pt)$ operations.

Note that the proof of the theorem assumes we know how to split the operations into blocks of a given size, and within each block assign one processor to each operation of the block. In most uses we will make of this theorem this scheduling problem will be easy, although this is by no means true in general. In particular the corollary follows immediately without any assumptions, because the operations we are counting there are simply the actions of the original processors.

Now let us look at simulations between different types of PRAM. With all of these simulation results, we have that if the original algorithm took polylog time with polynomial processors, the same will be true of the simulation of the algorithm.

The next theorem is the most important: it simulates the strongest of our models, the strong CRCW, on the weakest, the EREW. Therefore it can also be used to simulate an algorithm designed for any model by a parallel computer of any weaker model.

Theorem 2 (folklore). A parallel computation that can be performed in time t, using p strong CRCW processors, can also be performed in time $t \log p$, using p EREW processors.

Proof: Each step of the strong CRCW PRAM is simulated by $\log p$ steps of the EREW. We assume that processor identifiers on the EREW run from 1 to p. If the processors are reading from memory, each processor first writes the pair consisting of its own identifier and the address it wants to read from. If the processors are writing, they instead write triples of processor identifiers, addresses, and the values to be written. In either case the pairs or triples are written in constant time to an auxiliary array, with the position in the array being the identifier of the processor doing the writing. Because of uniqueness of identifiers, this causes no write conflict.

Next the pairs or triples are sorted by the addresses in them. In a write cycle, ties are broken by sorting according to the value being written. This sorting stage can be performed in time $O(\log p)$ with $O(p)$ processors [Ajtai et al. 1983; Cole 1986]. We now assign one processor for each element of the sorted list.

If the processors were writing, then each processor reads first the triple it is assigned to, and then the next position in the list. If a processor is assigned the final triple in the list, or if the address in the following triple differs from the one in the processor's own triple, the processor writes the value from its triple to the address from its triple. The sorted order of the list guarantees that only the processor with the highest value for a given address will attempt to write to that address.

If the processors were reading from memory, the values they want can be distributed as follows. Construct a list, where the values at each point of the list are the addresses and processor identifiers in sorted order as above, together with a value for each address. For each point in the list, the new value is computed as follows. If the point is the start of the list, or if the address there differs from that at the previous point, the processor for that point reads the value from memory. Otherwise it is taken to be the least value that will fit in a memory cell.

Once the list has been constructed, the processors compute for each prefix of the list the maximum pair of address and value that can be found in that prefix; pairs are ordered lexicographically, address first and then the value. This prefix maximum operation can be performed using the prefix computation algorithm described in the next section. The maximum up to a given element in the list will have the same address as that element, because the addresses were already sorted, but the value part of the maximum will be the actual data value at that address. Now the processor for each triple of value, address, and processor identifier, writes the value of the triple to an auxiliary array indexed by the processor identifier. Finally, the processors read the value they wanted from this array using their own identifier as index.

It is not difficult to verify that the above actions simulate exactly those performed by the original strong CRCW. The initial writing of addresses and reading of values, and the final writing of values, can be performed in constant time with $O(p)$ processors; by the corollary to theorem 1 they could also be performed in time $O(\log p)$ using $O(p/\log p)$ processors. The two prefix computations each take time $O(\log p)$ using $O(p/\log p)$ processors, as we shall see later. The initial sorting can be performed in the same time but with $O(p)$ processors. This gives the total bounds of the statement of the theorem. •

In fact with a somewhat more complicated prefix computation, we need not sort the addresses, but may instead choose any order for the list of addresses and values in which all members of the list having the same address are adjacent in the list. If such an ordering can be found more efficiently than by sorting, or if we could sort addresses more efficiently, we could perform the total simulation using only $O(p/\log p)$ processors, which is the best we could hope for. In general it is not known how to do this, but for many specific algorithms the structure of concurrent reads and writes is simple enough that this more efficient simulation can be performed. Also, for probabilistic computation we have the following:

Corollary. A parallel computation that can be performed in time t, with p probabilistic strong CRCW processors, and using the assumption that addresses can be written in at most $O(\log p)$ bits, can also be performed in expected time $O(t \log p)$, using $O(p/\log p)$ probabilistic arbitrary-winner CRCW processors.

Proof: Reif [1985] has shown that n numbers, each represented by at most $O(\log n)$ bits, can be sorted in expected time $O(\log n)$ using $O(n/\log n)$ probabilistic arbitrary-winner CRCW processors. Since it is easy to test that the numbers are sorted, the algorithm is Las Vegas (always correct). The result follows by using this result in place of the deterministic sort in the proof of the theorem. •

The corollary above gives one example where a weak model of concurrent write can perform a better simulation of a strong CRCW than the best known EREW simulation. As another example, we give the following theorem, which applies when we want to know the minimum time for performing a parallel computation with less concern for the total number of processors used.

Theorem 3 [Kučera 1982]. A parallel computation that can be performed in time t, using p strong CRCW processors, can also be performed in the same asymptotic time using $O(p^2)$ weak CRCW processors.

Proof: For each pair of original processors p_i and p_j we add an auxiliary processor q_{ij}. When p_i wants to write a value v_i to address a_i, it first writes a_i and v_i to locations in auxiliary memory where all processors q_{ij} read them, and sets a flag f_i in auxiliary memory to one. Each pair of processors q_{ij} and q_{ji} then compare their addresses and data. If $a_i \neq a_j$, they do nothing. Otherwise if $v_i < v_j$, or if $v_i = v_j$ and $i < j$, processor q_{ij} writes a zero to f_i. Finally all original processors for which the flag is still one perform the write. It can be seen that the only concurrent writes are those zeroing flags, that of all processors wanting to write to a given address exactly one will be left with a nonzero flag, and that that one will have the largest data value of those processors wanting to write there. •

As we discussed earlier, the result of theorem 2 can be improved in the case that we know enough of the pattern of concurrent memory access to be able to group accesses to the same cell into blocks without the use of a general-purpose comparison sort. A similar improvement can be made in theorem 3, again in the case that the pattern of concurrency is known. We will only describe algorithms for finding the maximum among n values; many uses of strong concurrent write in parallel algorithms reduce to this case.

Theorem 4 [Fich et al. 1983; Shiloach and Vishkin 1981]. The maximum of n values can be found in the weak CRCW model (a) in time $t = O(\log \log_k n)$ using $O(kn/t)$ processors; (b) in constant time using $O(n^{1+\epsilon})$ processors; and (c) in constant time using $O(n)$ processors, this last case under the assumption that the values can be represented using at most $O(\log n)$ bits.

Proof: Cases (a) and (b) were proved by Shiloach and Vishkin [1981]; in the proof here we use a technique known as accelerated cascades, described by Cole and Vishkin [1986a], in which an instance of a problem is reduced in a sequence of stages to smaller instances of the same type of problem. The stages start out making small reductions, but as the problem instances become smaller there are proportionally more processors available and the reductions can become larger and larger.

The reductions use a subroutine which we will call *reduce*(r). It reduces the problem from one of taking the maximum of n values to one of n/r values, using rn operations, in the following manner. The values are split into n/r blocks of size r. Then for each block, using r^2 processors per block, we find the maximum in constant time as in theorem 3. The maximum of all n values can then be found as the maximum of the n/r block maxima.

First we prove case (a). For notational convenience, let $m = n/\log\log_k n$. The algorithm for case (a) first performs $\log\log\log_k n$ iterations of *reduce*(2), reducing the total number of values left to m in time $O(\log\log\log_k n)$. Because each iteration uses half as many operations as the previous one, and using theorem 1, the total number of operations is $O(n)$. We now proceed in $\log\log_k m$ stages, each taking constant time. Prior to stage i there will be $m/k^{2^{i-1}-1}$ values left to take the maximum of; at stage i we call *reduce*$(k^{2^{i-1}})$, leaving m/k^{2^i-1} values. The number of processors used by each call to *reduce* is $O(k^{2^{i-1}} \cdot m/k^{2i-1-1}) = O(km)$, so the total number of processors used by the algorithm is again $O(km)$.

Case (b) is a special case of case (a), with $k = n^\epsilon$.

The proof of case (c) was given by Galil [1986a], who attributed it to Fich et al. [1983]. We split each value into blocks of $\log n/2$ bits. Since each value can be represented using at most $O(\log n)$ bits per value, there is a constant number of blocks per value. We proceed one block at a time, starting with the high order bits first. Within a block we use weak concurrent write to find which of the $\sqrt{n}$ settings of the block's bits are used by the values we have left. Then we take the maximum of these $\sqrt{n}$ values in constant time with $O(n)$ processors, using theorem 3, throw out all values whose bits in that block do not match the maximum, and repeat until we reach the last block. At this point we know the bits in each block of the maximum value, and therefore we know the maximum value itself. •

Valiant [1975] has given lower bounds showing that cases (a) and (b) are the best possible for comparison algorithms. Case (c) improves these bounds but only by splitting the values into bits, which causes it not to be a comparison algorithm. Valiant also gave matching upper bounds using a decision tree model of parallelism; the algorithms of Shiloach and Vishkin [1981] given in the proofs of cases (a) and (b) are essentially those of Valiant translated to the PRAM model.

2. PREFIX COMPUTATION AND RANKING

2.1 The Prefix Computation Algorithm

A prefix calculation takes as input a sequence of data values $v_1, v_2, \ldots v_n$ for some associative binary operation $\circ$ taking pairs of data values to single values drawn from the same range. The binary operation need not be commutative. The result of the computation is a sequence of prefix sums $p_1 = v_1, p_2 = v_1 \circ v_2, \ldots p_n = v_1 \circ v_2 \circ \cdots \circ v_n$. There is a simple algorithm for performing prefix computation on an EREW PRAM, discovered by Ladner and Fischer [1980], which runs as follows.

(1) For each input value v_i, if i is even, calculate the values in a new array x of temporary values as $x_{i/2} = v_{i-1} \circ v_i$.

(2) Recursively calculate the prefix sums of the $\lfloor n/2 \rfloor$ data values from x_i into another new temporary array y_i.

(3) For each value v_i, if $i = 1$ then $p_i = v_i$; if i is even then $p_i = y_{i/2}$; or finally if i is odd and greater than 1 then $p_i = y_{(i-1)/2} \circ v_i$.

Theorem 5 [Ladner and Fischer 1980]. A prefix computation with n inputs can be performed in $O(\log n)$ time using $O(n/\log n)$ EREW processors, each capable of performing the binary operation of the prefix computation in constant time.

Proof: Steps (1) and (3) take constant time with n processors. The recursion in step (2) involves a problem instance half the size of the original one; therefore the recursion bottoms out $O(\log n)$ levels deep, and the whole computation takes time $O(\log n)$. At first it would seem that we need $O(n)$ processors to achieve this time, but in fact we only need $O(n/\log n)$. This can be seen in the fact that the number of processors needed decreases by a factor of two at each level of the recursion, and therefore (using theorem 1) there is a total of $O(n)$ operations performed. •

We should mention here that Reif [1983] has proved that certain prefix computations can be performed in $O(\log\log n)$ expected time, again with $O(n)$ total operations. This result is deterministic, but for the average case in which the inputs are drawn from a random distribution. The result uses essentially the same algorithm as above, together with the observation that it is likely to be necessary to look at only $O(\log n)$ values prior to a value v_i, rather than the entire sequence of values up to that point, in order to determine the correct value of p_i. Obviously this observation does not hold for arithmetic operations such as addition or maximum. But it does hold for others, and in particular it holds for the operation used by Ladner and Fischer [1980] to construct circuits for bitwise binary addition.

We should also note that prefix maximum may be performed in $O(\log\log n)$ (worst case) time, using linear weak CRCW operations [Schieber 1987]. The algorithm is similar to that described in theorem 4.

The prefix computation algorithm above only works when the input values are listed in an array, because we need to be able to tell for each value whether it is in an even or odd position in the list. We shall see later in this section how to deal with input sequences represented as a linked list; but there is another possible representation of a list for which it is possible to perform prefix computations. This is a representation as a binary tree in which each path from the root to a leaf has length at most $O(\log n)$, with the elements of the sequence stored at the leaves of the tree. Then in step 1 we compute sums of pairs of leaves into their parent in the tree, in step 2 we recursively compute the sums in the tree formed by removing the leaves from the original tree, and in step 3 we bring the results back down to the leaves. In step 3 we need some further information that was more easily available in the array version, and that is the address of the next leaf after each leaf in the tree; but that is easily calculated recursively as part of step 2. The condition that the tree be balanced is necessary for this version of prefix computation to take the same time as the array version, $O(\log n)$. Note also that we must be able to perform operations on one level of the tree at a time without having to pick out the nodes at that level from all the other nodes in the tree.

2.2 Applications of Prefix Computation

As we have seen in section 1, prefix computation can be used in the simulation of one model of parallel computation by another. Another simple application of prefix computation is in list manipulation. Let us assume that we have a list of data values, represented as an array with the contents of each array cell being the value at the corresponding position in the list. As mentioned above, we will see later how to deal with a linked list representation. Suppose that certain members of the list are marked, and others are unmarked; we want to extract a sublist consisting only of the marked members.

This problem can be easily solved using prefix computation. With each member of the list we associate an integer, initially one for the marked members and zero for the unmarked ones. Then we perform a prefix sum, using the usual arithmetic addition operation. The result we have calculated is, for each marked member of the list, the position that member has in the new sublist; then the marked members can simply copy themselves across to an array representing the sublist.

If we wanted to represent the resulting sublist as a linked list (i.e. each member of the sublist knows the name of the next member of the sublist), we could again do it with prefix computation. In this case the data values would be indices in the original list together with the mark or lack of a mark. If the first index is marked and the second is not, we take as the result of the binary operation the first index; otherwise we take the second index. The prefix computation for this operation copies the index of a marked list member all the way across a sequence of unmarked members until the member just before the next marked member, which can then read it from there in a single step.

As a less obvious example of prefix computation we will now describe the parallel simulation of a finite state automaton. A typical use for such simulation would be the lexical analysis step in a compiler [Aho and Ullman 1977]; the state of the automaton after each character of the program text determines where the edges of each token are and which kinds of token those are the edges of.

To calculate the state of an automaton after it has processed each prefix of a string of characters, first replace each input character by the transition function from states to states of the automaton corresponding to that character. Since the automaton is finite, all such transition functions can be represented in a constant amount of space. Now perform the prefix sum over these transition functions, using the binary operation of function composition. This will compute for each character position the function corresponding to the action of the whole prefix substring from the start of the string to that position. Finally for each resulting composite transition function, compute the result of that function given as input the initial state of the automaton. This will be the correct state of the automaton after it has processed characters up through that position.

This use of prefix computation to simulate a finite automaton was given in the original prefix computation paper of Ladner and Fischer [1980] and also later on, in a description of an implementation for the Connection Machine, by Hillis and Steele [1986]. The latter paper uses a simpler version of prefix computation that uses $O(n)$ processors instead of $O(n/\log n)$. The original paper uses its automaton simulation to define Boolean circuits of size $O(n)$ that can find the sum of two binary numbers, each n bits long, in time $O(\log n)$. The more general prefix computation result seems to have been new with that paper, but addition circuits with those bounds had been known for some time [Krapchenko 1970]. The construction follows.

The output of an addition circuit at a given bit position is the sum taken modulo 2 of the two input bits at that position with the carry bit from the previous position. The carry is defined to be the majority of those three bits. Thus one can define a four state automaton, where the states correspond to each combination of carry and output bits. The input to the automaton at each bit position, starting from the low order bits, is the pair of input bits at that position. Then the new state can be calculated from those two bits together with the carry portion of the old state.

One can construct a circuit to perform prefix computation analogously to the EREW algorithm described above. Composition of the transition functions of the binary addition automaton can easily be computed by circuits of constant size, so this composition operation can be incorporated in the prefix computation circuit. Finally, one needs a stage to translate pairs of input bits to transition functions, and another to translate states to the corresponding output bits. Putting all of this together we have a linear size circuit to perform binary addition in logarithmic time.

2.3 Ranking

We have seen that prefix computation is useful in a great variety of circumstances. But often we might want to perform a prefix computation on a sequence of values represented as a linked list, and the techniques described above are not adequate for the task. Here we shall consider the problem of ranking, which as we shall see is equivalent to that of prefix computation.

The rank of a member of a linked list is simply the number of its position in the list, i.e. 1 for the first member, 2 for the next, and so on. The ranking problem is, given a linked list, to calculate for each member of the list its rank in the list. If we could perform prefix computation we could perform ranking, by giving each member a number value initially 1, and performing a prefix sum on those numbers. Conversely, if we could rank we could perform prefix computation, by using the ranks to move the data into an array and performing the array prefix computation algorithm described above. Thus, as we have said, ranking is equivalent to prefix computation.

One algorithm for ranking, or equivalently prefix computation, due to Wyllie [1979], is as follows.

(1) Give each list member of the list a data value of 1.

(2) For each list member other than the first, add the value of the previous member in the list into the value at this list member.

(3) Recursively call steps (2) and (3) on a new linked list formed by linking each list member with the list members two steps away in the original linked list.

This is just like the array prefix computation algorithm, except that we don't distinguish between even and odd members of the linked list. The recursive call in step (3) splits the list into two separate lists of half the length, and this can only continue for $O(\log n)$ stages before all lists have only one member; therefore the algorithm takes total time $O(\log n)$. But because we don't distinguish between even and odd, the size of the recursive subinstances of the problem adds to the same size as the original instance, so we need $O(n)$ processors at each stage. Thus we have lost a factor of $O(\log n)$ in efficiency over the array version of prefix computation.

2.4 Randomized Ranking and Deterministic Coin Tossing

As we have noted, if our list were represented as a balanced binary tree we could perform prefix computation, or equivalently ranking, as efficiently as if it were an array. Therefore we might start looking for an algorithm to turn the list into a tree. This can be done as follows:

(1) For each element of the list, flip a random coin. Form set S as that of all elements whose coin was 1 and whose predecessor's coin was 0. S is an independent set, that is, no member of S is adjacent in the list to another member of S.

(2) For each member of S, make a new tree node with both it and its predecessor (which is not in S) as children.

(3) For each non-member of S that is not followed in the list by a member of S, add a new tree node with that element as its only child.

(4) Make a linked list in a separate array of the new tree nodes.

(5) Recursively find a tree connecting the new list of tree nodes.

With high probability, the new list is 3/4 the size of the old one; therefore the tree has logarithmic expected height. Further, this geometric reduction between levels of the recursion ensures that the total number of operations is linear in the size of the original input list. The steps requiring the most time are (3) and (4), which can each be solved with a prefix sum taking $O(\log n)$ time. There are $O(\log n)$ levels of recursion; therefore the algorithm as a whole takes time $O(\log^2 n)$, which is more than we would like.

We can make one improvement fairly simply, and that is to only construct tree levels until the size of the remaining linked list (which can be calculated as part of the prefix sum for step (4)) is $\Theta(n/\log n)$. Each level still takes time $O(\log n)$, but in this version there are only $O(\log\log n)$ levels of recursion. This constructs a forest of $O(n/\log n)$ trees; we can perform prefix computation by using the tree algorithm within each tree, and using Wyllie's linked list algorithm on the roots of the trees. The total expected time is now $O(\log n \log\log n)$, again using a linear number of operations. The algorithm described above was first given by Vishkin [1984c], who also gave a more complicated probabilistic algorithm taking time $O(\log n \log^* n)$ using the same asymptotic number of operations.

It turns out that, if we are willing to make the assumption that memory addresses are represented using at most $O(\log n)$ bits, we can achieve the same bounds as above deterministically. We describe the $O(\log n \log\log n)$ version, but similar techniques apply to the $O(\log n \log^* n)$ algorithm mentioned above. Instead of flipping coins we find a maximal independent set in the linked list; this is an independent set such that each list element is adjacent to some member of the set. Such a set must have at least $n/3$ elements, so if we use it as S in the algorithm above we will again get the correct geometric reduction in list length at each level. A maximal independant set can be found in time $O(\log n)$ using $O(n)$ operations (matching the bounds used in step (4) above, and thus retaining the same total bounds for ranking) using the following algorithm.

(1) Initialize the set S to the empty set.

(2) Give each element i a different color s_i, such that adjacent members of the list have different colors. Initially we pick color s'_i to be i itself; this gives us n colors ranging from 1 to n. Then in constant time we can reduce the number of color classes to $O(\log n)$, by choosing a new color s_i to be the position of the first bit at which s'_i differs from s'_{i-1}, together with the value of that bit in s'_i.

(3) Bucket sort the array containing the linked list, ordering the list members by their color classes. This is performed using the following steps.

  (a) Divide the list elements into groups of size $\log n$. Allocate one processor for each of the $n/\log n$ groups; sequentially within each group, calculate for each color class s the number $c_{g,s}$ of elements in the group with color equal to s. Also count for each list element i the number p_i of vertices before it in the same group and having the same color.

  (b) Perform for each color class s a prefix sum algorithm on the $n/\log n$ numbers $c_{g,s}$, into $d_{g,s}$, the number of elements up through that group having that color. Let d_s be the total number of elements in all groups having color s.

  (c) With a single processor, perform a prefix sum on the numbers d_s to obtain t_s, the total number of list elements having colors less than or equal to s. Using binary trees, make $n/\log n$ copies of each t_s, one for each group.

  (d) Again allocate one processor per group. For each element i in group g, with elements processed sequentially within each group, calculate the position of i in the sorted list as $t_{s_i-1} + d_{g-1,s_i} + p_i$. This is the number of elements in the list having a lower color, or having the same color but appearing earlier in the unsorted array.

(4) Divide the sorted list into blocks, with each block consisting of all elements with a given color. We perform $O(\log n)$ stages, in each stage processing a single color block. Within each block check in parallel for each list element i whether either of its neighbors is in S already. If not, add i to S.

Because adjacent elements have different colors, and because step (4) only processes one color class with each parallel step, it never adds two adjacent elements to S, and thus the result is an independant set. The set is maximal because step (4) checks each element once, and only discards it if some adjacent element is already in S. It can be seen that all steps can be implemented in time $O(\log n)$ using $O(n/\log n)$ EREW processors; the only point at which these bounds are not entirely clear is step (4), for which we need to apply theorem 1. Thus we can use this routine in place of the coin tossing in our ranking algorithm, and rank a linked list in $O(\log n \log\log n)$ time with linear deterministic EREW operations.

We should also note that, by repeatedly reducing the number of colors as we did in step (2), we can find the independent set S in time $O(\log^* n)$, using a linear number of processors. For our purposes the algorithm above is better, since it uses less total operations without slowing down the main algorithm of which it is a subroutine.

The maximal independent set algorithm above, along with its application to ranking, was discovered by Cole and Vishkin [1986a, 1986b], who called it deterministic coin tossing. A bucket sorting algorithm similar to the one in step (3) was described by Reif [1985], who used it as part of his probabilistic algorithm for bucket sorting numbers with larger keys.

2.5 Optimal Logarithmic Time Ranking

We described above an algorithm for parallel list ranking that runs in time $O(\log n \log\log n)$ using $O(n/\log n \log\log n)$ EREW processors, for a linear total number of operations; this algorithm is likely to be sufficient for all practical purposes. But the extra factor of $O(\log\log n)$ in the time is irritating to theoreticians, and so two of them, Cole and Vishkin [1986c], have come up with an algorithm that runs on logarithmic time and linear EREW operations. As we have said, the previous algorithm is sufficient in practice; further, the newer logarithmic time algorithm is not practical, because the constants involved in the bound on its runtime are quite large. Nevertheless, we shall spend the rest of this section describing the new algorithm.

The general outline of the algorithm is as follows. We start by performing a sequence of operations, each one removing an element from the list, that reduces the problem to one of size $O(n/\log n)$; then we perform Wyllie's list ranking algorithm on the reduced linked list; finally we undo the operations we started out with to obtain the final ranking.

We assume that the list we are processing is doubly linked, so each element can find its successor and predecessor in constant time. Each element i of the list will start out with a number $r[i] = 1$, and for most of the algorithm the prefix sum of the $r[i]$ for the elements remaining in the list will give the ranks of those elements. At the last stage of the algorithm we will use Wyllie's algorithm to perform this prefix sum, after which point $r[i]$ will be the actual rank of element i.

The elements removed from the list will further have a pointer $p[i]$ to the predecessor of i at the time i was removed, and a timestamp $t[i]$. A removal of element i is performed by setting $p[i]$ to the current predecessor of i, adding $r[i]$ into $r[j]$ where j is the successor of i in the remaining list, removing i from the doubly linked list, and setting $t[i]$ to be the time at which the removal was performed. For the algorithm to be correct, no two adjacent list elements may be removed at the same time.

When Wyllie's algorithm on the shortened list has finished, we still need to find the ranks of the removed list elements. This is done in the reverse order from that in which they were removed. Therefore we need to sort elements by their removal times; these times are $O(\log n)$ so we can use the deterministic bucket sort of Reif [1985] described earlier. The reverse order guarantees that, by the time we calculate the rank of element i, the rank of $p[i]$ will already have been calculated into $r[p[i]]$. We add this rank into $r[i]$, giving the correct total rank of element i.

It remains to show how we can perform a sequence of removals, with no two adjacent elements removed at the same time, taking total time $O(\log n)$ with $O(n/\log n)$ processors, and leaving at most $O(n/\log n)$ elements unremoved.

The algorithm is controlled by a scheduler, about which we will have more to say later. Choices made by the scheduler alternate with stages of the algorithm; each stage takes constant time using $O(n/\log n)$ processors, and the scheduler will make sure that there are $O(\log n)$ stages.

We introduce two more values kept by each list element i: A count $c[i]$ of the number of removals performed by that element on other elements of the list, and a state $s[i]$. The state can be any one of the following five values: waiting, active, marked, full, done. If $s[i]$ is waiting, no processor has yet been assigned to i, but the scheduler may at any time change the state of i to active. If $s[i]$ is active, the scheduler will assign a processor to i in the next stage of the algorithm. If $s[i]$ is marked, i will no longer be assigned a processor, but it has not yet been removed from the list; some other active element will remove it later. We mark elements rather than removing them immediately because otherwise we might try to remove two adjacent elements at the same time. If $s[i]$ is full, it is still in the list, and will remain in the final reduced list. If $s[i]$ is done, i has been removed from the list. Not all removals are counted in $c[i]$; in particular if i is removed by its own processor there is no need to increment $c[i]$.

Each element is initially waiting; when the algorithm finishes, all elements must either be full or done, and there can be at most $O(n/\log n)$ full elements. Therefore we can see that after the completion of the algorithm, the reduced list will have length at most $O(n/\log n)$ as desired.

At each stage of the algorithm, after the scheduler has perhaps made some waiting elements active, each active element i is assigned a processor p_i which performs the following operations.

(1) If $c[i] \geq \log n$ and the successor of i is not marked, p_i sets $s[i]$ to full and stops processing element i.

(2) If i is still active and its successor is full, p_i checks the state of the predecessor of i. If the predecessor is also active, p_i sets $s[i]$ to marked; otherwise it removes i from the list and sets $s[i]$ to done.

(3) If i is still active, then the previous steps did nothing, so in particular the successor of i is not full. Nor is it done, because done elements have been removed from the list. Therefore i belongs to a chain of active elements followed by an element that is either marked or waiting. The processors corresponding to the elements of this chain find an independent subset S of the chain, such that each member of S is at a distance of at most $O(\log n)$ from some other member; this independent set computation will be described below

(4) If i is active, and the first member of its chain, but is not in S, then p_i tests whether the successor of i is in S. If so, p_i removes i from the list; otherwise, it adds i to S. After this step, the first element in the remaining chain will be in S.

(5) If i is active, but not in S, p_i sets $s[i]$ to marked.

(6) If i is still active, and therefore in S, p_i removes the (marked or waiting) successor to i (setting its state to done) and increments $c[i]$.

Elements are only marked when their predecessors are either marked or active, and when it is known that the chain of marked elements is headed by some active element; an active element at the head of such a chain will not become inactive until all marked elements have become removed. Therefore when the algorithm terminates there can be no marked elements. All waiting elements will eventually become active, so when the algorithm terminates there can be no waiting elements. And if there is an active element the algorithm has not terminated, so when it does terminate there can be no active elements. Therefore we can deduce that all elements either become full or are eventually removed from the list. If an element i becomes full, this can only be after p_i has performed $\log n$ removals of other elements in step 6, and so there can be at most $O(n/\log n)$ full elements. Thus our algorithm correctly reduces the length of the list to $O(n/\log n)$.

Before we show that it can be performed in the claimed bounds, let us first tie up the remaining loose end, the calculation of S. S is found in constant time as follows, using a slightly different version of the deterministic coin tossing technique described earlier. S will be an independent set, but it will not necessarily be maximal. As in the previous application of deterministic coin tossing, calculate for each element a color drawn from a set of $O(\log n)$ colors. The members of S are taken to be those elements i for which the color of i is less than those of both its successor and its predecessor.

Colors of adjacent list elements are different; therefore it is impossible for two adjacent elements to both have colors that are local minima, so S is an independent set. The colors of the elements following a particular local minimum can grow larger for at most $O(\log n)$ elements before they must drop again, and after this drop they can continue to drop for only $O(\log n)$ elements before rising again; therefore the members of S are at most $O(\log n)$ entries apart.

It remains to show that all of the computations described above can be performed in time $O(\log n)$ using linear operations. First let us see that each element i is active for at most $O(\log n)$ time units. Call a stage in which i remains active, but the successor to i was active and becomes marked, bad; call a stage in which i becomes inactive itself, indifferent; and call a stage in which i removes a marked successor from the list and increases $c[i]$ good. It can be seen that all stages are bad, good, or indifferent. Further, there is at most one indifferent stage, and each bad stage is followed by a good stage, so we need only count the number of good stages.

At any stage, $c[i]$ is exactly the number of prior good stages. Let stage s be the last bad stage executed for element i. Then prior to s there can only have been $O(\log n)$ good stages, because otherwise i would have declared itself full and s would be an indifferent stage. Stage s will be followed by a number of good stages equal to the number of marked elements following i in stage s, because an element following i can only become marked in a bad stage. But the number of marked elements is bounded by the distance between members of S, which as we have seen is $O(\log n)$. Therefore there are $O(\log n)$ good stages both before and after the last bad stage, so the total number of stages in which i is active is also $O(\log n)$.

Now we can see that, assuming a smart enough scheduler, the algorithm described above can be performed in time $O(\log n)$ using $O(n/\log n)$ EREW processors as claimed. Each processor is assigned a pool of waiting elements; whenever the element it was assigned to becomes inactive, it chooses a new element from its pool to make active. The pools are periodically rebalanced, so that until the number of waiting elements has been reduced to $O(n/\log n)$, a constant fraction of the processors will have waiting elements in their pools. The rebalancing itself is beyond the scope of this paper; for details see Cole and Vishkin [1986c].

At each stage, each active processor will either make its own element inactive or remove another element from the list; each of these actions can happen at most linear times. Prior to the stage s at which the number of waiting elements is reduced to $O(n/\log n)$, there will be $\Omega(n/\log n)$ active processors, so together with the linear bound on the number of actions this gives a bound of $O(\log n)$ on the time used. After that stage, we can distribute the remaining waiting elements evenly among the processors using a prefix computation; each processor will have a constant number of elements in its pool, and as we have shown each element can remain active for at most $O(\log n)$ stages, so the time used will again be bounded by $O(\log n)$.

3. EULER TOURS, EAR DECOMPOSITION, AND RELATED TECHNIQUES

In the previous section we discussed prefix computation techniques, which have wide application to problems in which the data is arranged in some linear order. In this section we discuss a number of algorithmic techniques for handling the case that the data to be processed are not already in a linear order, and in particular when the data are structured as a graph or as a tree. These techniques find an ordering of the edges or vertices of the graph or tree, so that prefix computation can then be used on the resulting sequence of edges or vertices. In general graphs it is less clear what such an order might be; we also describe algorithms for partitioning the graph edges into a sequence of subgraphs, each of which has a simple and easily ordered structure.

When the problem already has some structure as a tree, the parallel algorithms can be quite similar to their sequential counterparts. Sequentially, one often solves tree problems by using some standard tree traversal order, such as pre-order, post-order, or in-order. For parallel algorithms one would like to generate a list of the tree vertices in these orders; this can be done using the Euler tour technique described below. Many tree problems can be solved directly using Euler tours, rather than indirectly on some generated traversal order.

We should mention another important parallel technique which has in many cases been replaced by that of Euler tours. One obvious method for computing tree functions would be to process each leaf separately, and then remove the leaves and repeat until all vertices of the tree have been processed. However, the tree may be very deep, in which case this may not give us the polylogarithmic time bounds we want. An improvement to this method, called centroid decomposition [Winograd 1975; Megiddo 1983; Miller and Reif 1985], does give polylogarithmic bounds. In centroid decomposition, one alternates this removal of leaves with the removal of vertices having only one child (sometimes with the further restriction that the parent of the vertex also only has one child).

Cole and Vishkin [1987] have given an improved centroid decomposition algorithm that for a number of problems can be made to run in logarithmic time and a linear number of EREW operations, matching the bounds for Euler tours. We will not describe centroid decomposition further, except to note that as one of the applications of Cole and Vishkin's algorithm we may compute, given a tree with a value stored at each vertex, the minimum value in each subtree of the tree. This will turn out to be useful in computing ear decompositions.

For sequential algorithms on general graphs, one would typically traverse the edges of the graph using a depth first search [Tarjan 1972]. As we have mentioned, finding a depth first search tree in parallel seems to be difficult; a randomized parallel algorithm is known that constructs such a tree [Aggarwal and Anderson 1987], but this algorithm is not useful in practice for finding efficient parallel algorithms. As replacements for depth first search in general graphs, we describe algorithms for finding Euler tours, pseudo-forest decompositions, and ear decompositions.

Finally, note that two more algorithms for ordering graphs, topological sorting and breadth first search, have wide application in sequential algorithms. The best known parallel algorithms to perform these tasks are much less efficient than their sequential counterparts, and therefore these methods are less useful as building blocks for parallel algorithms than they are sequentially. Finding a breadth first search tree is equivalent to finding the shortest paths to or from a single vertex in a graph, but in parallel it seems we must instead find the shortest paths between all pairs of vertices in the graph. This latter task can be described as a closed semiring system [Aho et al. 1974], and the parallel semiring algorithm of Kučera [1982] described in the next section can be used to solve the problem. The best known parallel algorithm for topological sorting, also due to Kučera [1982], again uses a semiring system, this time the one for finding the transitive closure of a graph, and again it will be described in the next section.

3.1 Euler Tours for Trees

In the Euler tour technique, we order the edges of the tree by finding an Euler tour on the directed graph formed by replacing each tree edge with two directed arcs, one in each direction. Such a tour is a directed cycle such that each arc appears exactly once in the cycle. Of course the vertices of the tree will typically appear many times in the cycle, so it is not a simple cycle. The Euler tour technique for trees was introduced by Tarjan and Vishkin [1984], who used it as part of an algorithm for finding biconnected components of a graph. The examples we give of tree functions computable using Euler tours were also introduced in the same paper.

For each vertex v, let $p(v)$ be the parent of v in the tree, or some special marker if v is the root of the tree; let $s(v)$ be the next sibling of v in some ordered list of the children of $p(v)$, or some special marker if v is the last in the list or the root of the tree; and let $c(v)$ be the first child of v, or some special marker if v is a leaf. Denote the number of vertices in the tree by n.

In the Euler tour algorithm that follows we assume that each vertex v has a copy of each of these three values stored together with its local data. If $p(v)$ is not known but $s(v)$ and $c(v)$ are known, then $p(v)$ can be calculated in time $O(\log n)$ and linear operations using a prefix calculation on the linked lists formed by $s(v)$, and no efficiency will be lost in our algorithms; however the tour itself will be found in logarithmic time rather than constant time. But if only $p(v)$ is known, with $s(v)$ and $c(v)$ unknown, it seems that we must sort the vertices by their values of $p(v)$ in order to calculate the other two values for each; this takes linear processors and logarithmic time, which is a loss of both time and efficiency.

In the Euler tour, each vertex v is responsible for two edges, the one from $p(v)$ to v and the other one in the reverse direction. The edges into and out of the root will be taken care of by the children of the root; in the following description we assume that v is not the root. We need to calculate, for each edge, the next edge in the tour. If v is a leaf, we take the edge following $(p(v), v)$ to be $(v, p(v))$; otherwise we follow $(p(v), v)$ by $(v, c(v))$. Then, if $s(v)$ exists, the edge following $(v, p(v))$ is $(p(v), s(v))$. Otherwise $s(v)$ does not exist. In this case, if $p(v)$ is the root, we follow $(v, p(v))$ by $(p(v), c(p(v)))$ to close the cycle, or by nothing if an open Eulerian path is desired; if $p(v)$ is not the root, we follow $(v, p(v))$ by $(p(v), p(p(v)))$.

It can be seen that each edge follows, and is followed by, exactly one edge in the tour; further all such links may be computed in constant time with $O(n)$ EREW processors. Typically we will perform prefix computations on values placed at each edge of the tour; this takes $O(\log n)$ time with $O(n/\log n)$ processors. By theorem 1 we can compute the tour itself in these same bounds.

Theorem 6 [Tarjan and Vishkin 1984]. Given an Euler tour of a tree, a pre-order or post-order traversal of the vertices can be computed in time $O(\log n)$ using $O(n/\log n)$ EREW processors.

Proof: If we form a new tree by reversing the order of the children at each vertex of the original tree, and find the post-order of this new tree, the resulting ordering will be exactly the reverse of a pre-order of the original tree. Therefore we only describe here the construction of a post-order traversal.

We perform a prefix computation on the edges of the tour; the data at each edge will be that edge's name, together with a bit which is true if the edge goes down from a vertex to one of its children, or false if it goes up from a vertex to its parent. The result of the binary operation used in the prefix computation is as follows. If the bit of the second operand is true, the edge name and bit are both copied from the first identifier. Otherwise the resulting edge name is copied from the second identifier, and the bit of the result is set to false.

If we apply this operation to a sequence of edges, the resulting bit will be true if and only if all the edges in the sequence run down. The resulting edge name will be that of the edge appearing before the maximal subsequence of downward edges containing the last edge in the sequence, or the last edge itself if that edge runs up. This operation can clearly be seen to be associative.

Consider edges of the form $(v, p(v))$, let (u, v) be the edge preceding $(v, p(v))$ in the tour, and let (x, y) be the edge name left at (u, v) by the prefix computation. If v is not a leaf, then (x, y) will be (u, v) itself; x is the last child of v to appear in the tour, and can be taken as the predecessor of v in the post-order traversal. If v is a leaf, y is the first ancestor of v such that some child of y appears before the subtree containing v, and x is the last such child. Therefore x can again be seen to be the correct post-order predecessor of v. In this way we can calculate the predecessor of every vertex except the root; then the vertex v that does not yet have a successor, and is not itself the root, is the predecessor of the root. •

Theorem 7 [Tarjan and Vishkin 1984]. Given an Euler tour of a tree, we can calculate the number of vertices in each subtree of the tree, in the same bounds as the previous theorem.

Proof: Assign each edge of the tree the value one if it goes down from $p(v)$ to v, or zero if it goes up from v to $p(v)$. Find the prefix sum of these numbers. The sum computed at edge $(p(v), v)$ gives the number of different vertices appearing before v first appears in the tour; the sum computed at $(v, p(v))$ gives the number of vertices appearing before the last appearance of v. The difference between the two sums is the number of vertices in the subtree rooted at v. •

In a similar fashion, we may use prefix computation on Euler tours to compute many other tree functions. In particular, Tarjan and Vishkin [1984] used this technique to compute the distance of each vertex from the root of the tree, along with two functions they called $low(v)$ and $high(v)$, which they used in a parallel algorithm for computing the biconnected components of a graph. Tsin [1985], and independently Vishkin [1985], used Euler tours to compute a data structure for finding the least common ancestors of pairs of vertices in trees, which they both then used as part of algorithms for computing strong orientations of undirected graphs. Schieber and Vishkin [1987] gave an improved algorithm for least common ancestors, again using Euler tours in trees.

3.2 Euler Tours for Graphs

As in the case of trees, we can order the edges of an undirected or directed graph in an Euler tour; that is, a cycle such that each edge appears exactly once; if the graph is directed we further require that the cycle be directed. Obvious necessary conditions for an undirected graph to have an Euler tour are that it be connected and that the degree of each vertex is even; Euler proved that these conditions are also sufficient. Similarly, necessary and sufficient conditions for a directed graph to have an Euler tour are that each vertex have an in-degree equalling its out-degree, and that the graph be connected (strong connectivity follows from this and the first condition).

Awerbuch et al. [1984] gave efficient algorithms for finding Euler tours in the directed case, and for finding an orientation for any undirected graph such that the resulting directed graph has an Euler tour. Atallah and Vishkin [1984] independently arrived at the same results; we follow here the presentation from Awerbuch et al.

Define an Euler partition of a directed graph to be an assignment to each edge $e = (v, w)$ in the graph of some succeeding edge $s(e) = (w, x)$ in the graph, with the tail of e equal to the head of $s(e)$, such that no other edge is also assigned (w, x). This can easily be seen to partition the edges of the graph into disjoint cycles; the result is an Euler tour if and only if there is only one such cycle. If at each vertex we are given a list of the edges of that vertex, we can use a simple prefix computation on that list to compute for each edge its rank in the sublist of in-edges or out-edges, and from that by matching ranks an Euler partition, in time $O(\log n)$ using $O(n/\log n)$ EREW processors. As in the computation of Euler tours for trees, if we are simply given the collection of edges, without having lists of the edges at each vertex, we will first need to sort the edges.

Given an Euler partition, we now define what it means to swap two edges $e = (u, w)$ and $e = (v, w)$ having a common tail w. Let $s(e) = g$, and $s(f) = h$. Then the result of swapping e and f is to make $s(e) = h$ and $s(f) = g$; i.e. the two edges will have traded successors.

We define the graph CG of an Euler partition as follows. Vertices of CG will be cycles in the Euler partition. For each edge $e = (u, v)$ in the original graph, let f be the next edge after e in the list of edges incoming to v. Then we add an edge in CG between the cycle containing e and that containing f. Note that CG may contain self loops as well as more than one edge between pairs of vertices; this turns out not to be a problem.

Awerbuch et al. [1984] give the following theorems; the proofs are not difficult, but we refer the reader to the original paper for them.

Theorem 8 [Awerbuch et al. 1984]. CG is connected.

Theorem 9 [Awerbuch et al. 1984]. Given a spanning tree of CG, if we execute the swaps corresponding to the edges of the spanning tree in any order the resulting Euler partition will be a single cycle.

Because of these theorems, we can find an Euler tour of a directed graph using the following steps:

(1) Generate an Euler partition as described above.

(2) Find the circuits of the partition (which are the vertices of CG) and determine which circuit each edge of the original graph belongs to.

(3) Construct the edges of CG, and find a spanning tree for it.

(4) Execute the swaps indicated by the spanning tree.

Step (1) has been described already. Step (2) and step (3) both can use any algorithm for finding connected components and spanning trees of a graph. The best known algorithm for this is due to Cole and Vishkin [1986c]. It uses $O(\log n)$ time, with $O(m\alpha(m, n)/\log n)$ CRCW processors. Here n is the number of edges in the graph, m is the number of vertices, and α is the inverse Ackerman function, which grows very slowly. In both uses we are making of this connected components algorithm, both the number of vertices and the number of edges can be proportional to the number of edges in the original graph.

Finally, step (4) can be executed as follows. Let $e_i, e_{i+1}, \ldots e_j$ be a maximal sublist of the edges coming in to vertex v, such that $(e_i, e_{i+1}), (e_{i+1}, e_{i+2}), \ldots (e_{j-1}, e_j)$ are all swaps indicated by the spanning tree of CG. Then for $i \leq k < j$ we assign the old value of $s(e_{k+1})$ to be the new value of $s(e_k)$, and similarly we assign the old value of $s(e_i)$ be the new value of $s(e_j)$. This can be performed in logarithmic time and a linear number of EREW operations using prefix computation on the lists of edges incoming to each vertex v.

Theorem 10 [Awerbuch et al. 1984]. An Euler tour of a directed graph with n vertices and m edges, in which the number of edges incoming to each vertex v is the same as the number of outgoing edges from v, can be found in the same time and processor bounds as those for computing a spanning tree of an undirected graph with $O(m)$ vertices and edges, assuming those bounds are no better than $O(\log n)$ time with $O((n+m)/\log n)$ EREW processors if the graph is given as a list of edges at each vertex, or $O(n+m)$ processors if the graph is given only by its edges. Using the current best known connectivity algorithm gives bounds of $O(\log n)$ time with $O(m\alpha(m,m)/\log n)$ arbitrary winner CRCW processors.

Proof: as described above. •

Theorem 11 [Awerbuch et al. 1984]. An Euler tour of an undirected graph can be found in the same time and processor bounds as theorem 10.

The above algorithm works only on directed graphs. But note that we can construct an Euler partition as above for an undirected graph. Then if we could orient the edges of the graph so that each cycle of the partition is made directed, we would then be able to find an Euler tour for the resulting directed graph, which would also be an Euler tour for the underlying directed graph. These cycles can be oriented as follows.

First split the graph into a new graph which is a union of disjoint undirected cycles, by creating a new vertex between each edge and its successor in the partition. Now find a spanning forest of this graph. Each cycle of the partition will correspond to one tree of the forest, and each tree will consist of two directed paths leading to the root of the tree. Use prefix computation to reverse the direction of the edges on one of the two paths, and once this is done orient the remaining edge of the cycle by looking at the orientations of the edges to either side. Using this process and then continuing with the directed Euler tour algorithm gives us an Euler tour of the original undirected graph. •

Not every graph has an Euler tour, so we cannot usually use this algorithm directly on an arbitrary graph as part of some other graph algorithm. Further, if we tried the same trick that we used for trees, splitting every edge in two, one possible tour could be constructed from a tour of a spanning tree of the graph, and therefore any resulting tour would be little more useful than a spanning tree. But there is a different way to transform a graph into one having an Euler tour, and that is to add a new dummy vertex, and add an edge between that vertex and each original vertex having an odd degree. Israeli and Shiloach [1986] use this construction as part of an algorithm that efficiently and quickly finds a maximal matching for an arbitrary graph.

3.3 Pseudo-Forest Decomposition

Define a pseudo-forest to be a directed graph in which each vertex has out-degree at most one. Every rooted tree or forest is a pseudo-forest, but a pseudo-forest may contain cycles. In this section we describe extensions of the deterministic coin tossing technique of Cole and Vishkin [1986a, 1986b] to pseudo-forests, found by Goldberg et al. [1987]. These extensions will in turn lead to efficient algorithms for graphs of bounded degree, which operate by partitioning the edges of the graph into a number of pseudo-forests.

By a coloring of a graph or pseudo-forest below, we mean an assignment of colors to the vertices such that no two adjacent vertices have the same color. In what follows, n will refer to the number of vertices of a given pseudo-forest or graph, and m will refer to the number of edges of the graph. The number of edges in a pseudo-forest can of course be no more than n.

Lemma [Goldberg et al. 1987]. Given a pseudo-forest colored with k colors, we can compute a new coloring with $2\lceil \log k \rceil$ colors, in constant time using $O(n)$ CREW processors.

Proof: We assign one processor per vertex of the pseudo-forest. Each processor reads the color of the parent of its own vertex, and sets the new color of its vertex to be the number of the first bit at which the parent's color differs from the color at its vertex, together with the value of that bit at the vertex. •

Define $\log^* x$ to be $\min\{i : log^{(i)} x \leq 1\}$, where $\log^{(i)} x$ means the result of iterating the log function i times, starting with an initial value of x.

Theorem 12 [Goldberg et al. 1987]. Given a pseudo-forest, a coloring of that pseudo-forest using only 3 colors can be found in time $O(\log^* n)$ using $O(n)$ CREW processors.

Proof: We start with the n-color coloring given by using the name of each vertex as its color, and iterate the algorithm of the lemma. After $O(\log^* n)$ steps, the number of colors will have been reduced to 6, beyond which the lemma can not make any improvements. We now reduce the number of colors one step at a time, as follows. Each vertex takes as its color the color of its parent, with the root (if any) of the pseudo-forest taking any color differing from its previous color. After this step the neighbors of each vertex will only be of two different colors. Then each vertex having the color we wish to eliminate chooses as its new color some color other than those two. This can again be performed in constant time with a linear number of CREW processors, and after a constant number of such steps the number of colors will have been reduced to 3. •

Theorem 13 [Goldberg et al. 1987]. If the maximum degree of any vertex in the pseudo-forest is bounded by Δ, and if we are already given lists of the vertices incoming at each vertex, then the algorithm of theorem 12 can be performed on $O(n/\log \Delta)$ EREWs using time $O(\log \Delta \log^* n)$.

Proof: The only use we make of concurrent read is to get the color of each vertex's parent at each step. These colors can be distributed using prefix computations on the lists of incoming vertices. •

Theorem 14 [Goldberg et al. 1987]. Given a graph with maximum vertex degree Δ, represented by lists of edges at each vertex, with $\Delta = O(\log n)$, we can calculate a coloring of the graph with $\Delta + 1$ colors in time $O(\Delta \log \Delta(\Delta + \log^* n)$, using $O(n)$ EREW processors.

Proof: We first assign each edge of the graph an arbitrary direction. Next we partition the edges of the graph into $O(\Delta)$ pseudo-forests, by letting each vertex label its outgoing edges from 1 to Δ. For each value of i from 1 to Δ, each vertex constructs the list of edges incoming to it in pseudo-forest i by using a prefix computation on the list of all its incoming edges. Then we 3-color each pseudo-forest in turn, adding its edges back to the graph and combining the 3-coloring with a $\Delta + 1$-coloring of the already processed portion of the graph to find a new $\Delta + 1$-coloring.

The combination of colorings is done as follows. First each vertex takes as its color the pair of colors from the two colorings we are combining; the resulting coloring has $3(\Delta + 1)$ colors. We will also temporarily color the edges of the graph; initially all edges are uncolored, but as we color each vertex in the new $\Delta + 1$-coloring we will also give the same color to its adjacent edges, so that the vertices at the other ends of those edges will be able to find which color was used without worrying about concurrent read conflicts.

We now process each of the pairs of colors in turn; within a color class we process all vertices in parallel. Each vertex, when it is processed, looks at the colors of the edges adjacent to it and chooses a color that is not already taken; because it has at most Δ neighbors there must be at least one such color. Next it again goes through those edges and recolors them with its new color. Both of these steps can be performed with a prefix computation on the list of neighbors; because of the assumption that $\Delta = O(\log n)$ we can use bit vector operations to choose a color. Because we process only vertices from a single color class at a time, each pair of adjacent vertices will be processed at different times, so they will end up with different colors, and no two processors will try to color the same edge at once.

There are Δ pseudoforests, and each must be 3-colored and then combined with the previous coloring. The combination is done for each of $O(\Delta)$ color classes, and each class takes time $O(\log \Delta)$ using $O(n/\log \Delta)$ processors. The coloring also uses $n/\log \Delta$ processors, but it takes time $O(\log \Delta \log^* n)$. Therefore the total time taken is $O(\Delta(\log \Delta \log^* n + \Delta \log \Delta))$. •

Theorem 15 [Goldberg et al. 1987]. A maximal independent set of a bounded degree graph can be found in the same bounds as those of theorem 14.

Proof: We use the above algorithm, but when we recolor a vertex we always pick the lowest numbered available color. The vertices ending up with the lowest numbered color will form a maximal independent set. •

Goldberg et al. also give similar algorithms, again using a pseudo-forest decomposition, for finding colorings, maximal independent sets, maximal matchings, separators, and depth first search trees, all for planar graphs.

3.4 Ear Decomposition

An ear decomposition is another way of partitioning the edges of a graph, in this case into ears. An ear is simply a path in a graph; if the graph is directed, the ear must also be directed. No vertex in the interior of the path may appear more than once in the ear. If the endpoints of the ear are different, it is called an open ear. An ear decomposition consists of a partition of the edges of the graphs into a sequence of ears, such that the endpoints of each ear appear in previous ears but such that the interior points of each ear appear for the first time in that ear.

The following well-known theorems, quoted by Lovász [1985], indicate the importance of ear decompositions to questions of graph connectivity. When we say an ear decomposition starts from a vertex v, an edge e, or a cycle c, we mean that the first ear of the decomposition is v, e, or c respectively. An open ear decomposition is one in which all ears are open.

Theorem 16 [Whitney 1932]. An undirected graph is bridgeless (2-edge-connected) if and only if it has an ear decomposition starting from a single vertex.

Theorem 17 [Whitney 1932]. An undirected graph is biconnected (2-vertex-connected) if and only if it has an open ear decomposition starting from a single edge.

Theorem 18 (Folklore). A directed graph is strongly connected if and only if it has an ear decomposition starting from a single vertex.

In all of the above theorems, the ear decompositions may equivalently be required to start from a cycle; however the above descriptions give rise to simpler algorithms.

Lovász [1985] first noted that ear decompositions can be found quickly in parallel; he gave an algorithm for finding decompositions of strongly connected directed graphs, and of bridgeless and biconnected undirected graphs. Maon et al. [1986] showed Lovász' biconnected graph ear decomposition algorithm to be faulty, and gave a more efficient and correct algorithm for both undirected cases.

We describe here the algorithm given by Maon et al. for finding an ear decomposition of a bridgeless graph, shown to exist by theorem 16. The other algorithms are similar but more complicated. The algorithm itself, perhaps not surprisingly, turns out to be similar to those given by Tsin [1985] and Vishkin [1985] for strongly directing a bridgeless graph, mentioned as an application of the Euler tour technique for trees. In fact, if one applies the cycle directing algorithm, described previously as part of the Euler tour technique for undirected graphs, to the ear decomposition of a bridgeless graph, the result can be seen by theorem 18 to be a strongly connected directed graph.

To find an ear decomposition of a directed graph, one proceeds as follows.

(1) Find a spanning tree of the graph.

(2) For each edge (u, v) not in the spanning tree, compute the least common ancestor $a = lca(u, v)$ of u and v in the spanning tree, and the distance $level(u, v)$ from a to the root of the spanning tree.

(3) For each edge e in the spanning tree, find the set S of edges f such that e is part of the cycle induced by adding f to the tree. Let $master(e)$ be the member of S with the least *level*, with ties being broken by picking the minimum edge identifier.

(4) For each edge (u, v) not in the spanning tree, form an ear consisting of (u, v) itself together with all tree edges over which (u, v) is *master*. Sort the ears by $level(u, v)$ and within levels by the identifiers of the master edges used to break ties above.

Theorem 19 [Maon et al. 1986]. The above algorithm correctly finds an ear decomposition of a bridgeless graph.

Proof: For each tree vertex w other than the root, there must be some non-tree edge from w or one of its descendants to some vertex that is not a descendant of w, or else the edge from w to its parent would be a bridge. Therefore the edge from w to its parent, and similarly each tree edge, is contained in some ear.

The ear calculated for each non-tree-edge (u, v) is clearly a subset of the cycle induced by adding (u, v) to the tree, which can be divided into (u, v) itself, the path from u to $lca(u, v)$, and the path from v to $lca(u, v)$. If some edge e along the path from u to $lca(u, v)$ is missing from the ear, then it is part of a cycle induced by an edge (w, x) with $level(w, x) \geq level(u, v)$. But this implies that $lca(w, x)$ is either $lca(u, v)$ or an ancestor of $lca(u, v)$, so all edges above e in the path are also part of the cycle induced by (w, x), and so they can not be part of the ear for (u, v). Similarly if an edge in the path from v to $lca(u, v)$ is missing from the ear, all edges above that edge will also be missing. Thus we see that each ear is in fact either a simple path or a simple cycle.

We also see from the above argument that if an endpoint x of the ear corresponding to edge (u, v) is not $lca(u, v)$, the edge from x to its parent in the spanning tree must belong to an ear appearing earlier in the sorted order. If $lca(u, v)$ is an endpoint, then either it is the root and appears in the initial point ear, or it has a parent and the edge from it to its parent appears in an earlier ear.

Finally we must show that no interior vertex of the ear containing (u, v) appears in a previous ear. If w is the root of the tree, it must be the *lca* of any ear containing it, and so it will be an endpoint rather than in the interior. Otherwise, assume w is not the root. Each ear consists of a nontree edge together with all or part of the cycle it generates; therefore if w is an interior vertex of an ear, then the edge from w to its parent in the tree must be contained in that ear, and can be contained in no other ear. Therefore w appears as an interior vertex of at most one ear. But the first ear containing w must contain it as an interior ear, because as we have seen the endpoints appear in previous ears. So w appears once as an interior ear, and before that appearance does not appear in any ear. •

Theorem 20 [Maon et al. 1986]. Given a spanning tree of a bridgeless graph, and assuming that the ears need not be sorted, the remaining steps of the ear decomposition algorithm above can be computed in time $O(\log n)$ time with $O((m + n)/\log n)$ CREW processors.

Proof: The least common ancestors can be computed using an algorithm of Schieber and Vishkin [1987]. This algorithm pre-computes a number of values at each vertex of the tree using the Euler tour method described above; from these values the least common ancestor of any pair of vertices can be computed in constant (sequential) time. In particular we can compute $lca(u, v)$ for each non-tree edge (u, v) in constant time with one processor per edge. Then $level(a)$ for each such common ancestor can again be found by the Euler tour method, although in fact it will have already been computed as part of Schieber and Vishkin's algorithm. Finally, we compute for each vertex the adjacent edge with the least *level*. We use the centroid decomposition of Cole and Vishkin [1987] to propagate these minimum edges up through the spanning tree, so that each vertex knows the minimum from all of its descendants.

For each vertex v, we calculate the *master* of the tree edge from v to its parent as the minimum edge value reaching v in the previously described computation; this is the non-tree edge adjacent to some descendant of v and having the least *level* of all such edges. If the tree was bridgeless, this computation will give us a correct ear decomposition. If the tree has a bridge from some vertex v to its parent, the computed *master* of that edge will also be the *master* of more than one edge from a child of v to v itself, and this erroneous condition can be detected again within the time and processor bounds given. •

Maon et al. [1986] also give an algorithm for finding an open ear decomposition of a biconnected graph. This algorithm is essentially the same as the one described above, except that the identifiers used to break ties between edges with the same level are carefully chosen so that none of the resulting ears are open.

A characterization of biconnected graphs closely related to that of the open ear decomposition is that each such graph has what is known as an st-numbering. This is an ordering of the vertices of the graph such that there is an edge between the first vertex s and the last vertex t, and such that each other vertex has a neighbor in the graph somewhere prior to it in the ordering, and another neighbor after it. It is well known [Even and Tarjan 1976] that an st-numbering can be computed from an open ear decomposition of a graph; Maon et al. [1986] also showed how to perform this efficiently in parallel. Klein and Reif [1986] used this st-numbering algorithm as part of a parallel algorithm for finding embeddings of planar graphs.

Ear decompositions have also been used as part of other graph algorithms, in particular in computing the connectivity of a graph. The ear decomposition algorithms we described above will determine whether a graph is bridgeless or biconnected, although at least in the latter case a simpler algorithm was known [Tarjan and Vishkin 1984]. Miller and Ramachandran [1987] used the open ear decomposition algorithm for biconnected graphs as part of an efficient algorithm for testing graph 3-vertex-connectivity, and Kanevsky and Ramachandran [1987] again used this decomposition to test 4-vertex-connectivity.

Linial et al. [1986] have found a characterization of vertex connectivity using a generalization of st-numbering. This characterization does not, however, involve ear decompositions. They give randomized algorithms for computing the connectivity of a graph using various matrix operations; we will not describe these algorithms, but they can be performed efficiently in parallel using the matrix techniques described in the next section.

4. MATRIX METHODS

One usually thinks of matrix calculations in the context of scientific computation, in which the matrix typically contains floating point numbers, and corresponds to an approximation of some physical system. Certainly this is a common use of matrices, and as the matrix systems used tend to be extremely large this is also an important application for parallel computation.

Matrices and matrix methods can, however, also be used to solve combinatorial problems. In such a problem one might for instance have as input a network (graph with edge weights), and want to compute shortest paths in the network. One would again have a matrix of floating point numbers: the adjacency matrix of the graph, with the value at each cell being the weight of the corresponding edge. As we shall see the values of the matrix cells need not even be numeric; matrix-like computations may still be useful given a matrix of values taken from some arbitrary semiring. For this reason the matrix algorithms we describe will be those that work for all matrices, no matter what algebraic system the cell values are taken from.

More formally, we define a semiring as a triple $(S, +, \times)$ where S is some (possibly infinite) set of values, and $+$ (addition) and $\times$ (multiplication) are associative binary operations on S, with multiplication distributing over addition. Addition in the semiring must be commutative, but in general we will not require the same of multiplication. Sometimes, as in the case of closed semiring systems, we will add further properties holding for the semiring operations. We assume that values of S can be stored in single memory cells, and that both addition and multiplication in the semiring can be performed in constant time by a single processor.

A matrix A, of dimension $m \times n$, can be thought of as a function from pairs of integers (i, j) with $1 \leq i \leq m$ and $1 \leq j \leq n$, to values from the semiring. We write $A[i, j]$ for the value of the matrix at the pair (i, j).

4.1 Basic Matrix Operations

Assume we are given two $m \times n$-dimensional matrices A and B. The matrix sum $C = A + B$ can be computed by, for each pair (i, j), letting $C[i, j] = A[i, j] + B[i, j]$; this can be computed in the obvious way, using constant time with $O(mn)$ EREW processors.

Given a matrix A of dimension $m \times n$ and another matrix B of dimension $n \times p$, the product $C = AB$ is defined as the $m \times p$-dimensional matrix calculated by $C[i, k] = \sum_{j=1}^{n} A[i, j] \times B[j, k]$. If we use a balanced binary tree of additions to perform the sum for each element of C, the result can be computed in time $O(\log n)$, using $mnp/\log n$ EREW processors. We are unlikely to do better than this for arbitrary semirings, but if more is known we can sometimes improve this result. For instance, if the addition operation of the semiring is finding the minimum of $\log n$-bit integers, as in the case of the breadth first search algorithm given later, we can compute the matrix product in constant time with $O(mnp)$ weak CRCW operations. Or, if the semiring is in fact a ring (i.e. additive inverses exist), we can use various more complicated algorithms to reduce the total number of operations; the best known bound on the number of processors needed to multiply two $n \times n$ matrices over a ring in logarithmic time is $O(n^{2.376})$ [Coppersmith and Winograd 1987].

An important special case of matrix multiplication that we will use later is when the two matrices to be multiplied are lower triangular Toeplitz matrices. In a Toeplitz matrix A, for each diagonal of the matrix, each cell of the diagonal has the same value as each other cell. If the matrix is lower triangular, it can be represented by the values at $A[i,1]$; all other values can be derived from these using the formula $A[i,j] = A[i-j+1,1]$.

Theorem 21 [Berkowitz 1984]. Given two lower triangular Toeplitz matrices A and B, of dimensions $m \times n$ and $n \times p$, the product $C = AB$ is lower triangular and Toeplitz, and can be computed in time $O(\log n)$ with $O(mn/\log n)$ EREW processors.

Proof: It can be seen from the definitions of matrix multiplication and of Toeplitz matrices that $C[i,1] = \sum_{j=1}^{\min(i,n)} A[i-j+1,1] \times B[j,1]$. Each of these coefficients can be calculated independently with a prefix computation in the bounds of the theorem. •

Again one can make improvements in certain special cases; in particular if the base field of the matrices supports a fast Fourier transform we may compute the product in the same time as before, but using only $O(n)$ processors. Also note that the product of non-triangular Toeplitz matrices is not necessarily Toeplitz.

The following result simplifies and improves by a factor of n^ϵ a theorem of Berkowitz [1984]. It was discovered by von zur Gathen [1984], and independently by the first author. In this and the following theorems, $M(n)$ denotes the number of processors needed to multiply two $n \times n$ matrices in time $O(\log n)$. Note that $M(n)$ is at least $\Omega(n^2/\log n)$.

Theorem 22. Let R be a $1 \times n$ matrix (i.e. a row vector), A an $n \times n$ matrix, and S an $n \times 1$ matrix (column vector). We can compute $\{R \times A^i\}$, for all i from 0 to n, simultaneously, using $O(\log^2 n)$ time and $O(M(n))$ processors. Further, we may compute $\{R \times A^i \times S\}$ in the same bounds.

Proof: The algorithm for computing $R \times A^i$ proceeds in $\log n$ stages, each consisting of two matrix multiplications. Prior to stage k we will have computed the values of $R \times A^i$ for i from 0 to $2^{k-1} - 1$, arranged as a $2^{k-1} \times n$ matrix X. We will also have computed $A^{2^{k-1}}$. In particular prior to the first stage we already know R and A, as these are the input to the algorithm. Then at stage k we compute $Y = X \times A^{2^{k-1}}$. It can be seen that the rows of Y are exactly $R \times A^i$, for i from 2^{k-1} to $2^k - 1$; therefore combining X with Y gives $R \times A^i$ for all i from 0 to $2^k - 1$. To complete the required input for the next stage, we also compute A^{2^k} by squaring $A^{2^{k-1}}$.

Each stage takes time $O(\log n)$ using $O(M(n))$ processors, and there are $\log n$ stages, so the bounds of the theorem follow. Computing $R \times A^i \times S$ may be accomplished with one further matrix multiplication. •

An important function of a square $n \times n$ matrix A, closely related to the determinant and inverse, is the characteristic polynomial $p(\lambda) = \det(A - \lambda \cdot I)$, where det stands for the determinant operation and I is the identity matrix (having the multiplicative identity 1 on the main diagonal, and the additive identity 0 in all other cells). Let $\chi(A)$ be the column vector $(p_0, p_1, \ldots p_n)$ of coefficients of $p(\lambda)$; p_0 is the coefficient of λ^n in $p(\lambda)$, p_1 that of λ^{n-1}, and so on.

Define A_i to be the square submatrix of A formed by taking rows $i+1$ through n and columns $i+1$ through n of A. Also let R_i be the row vector $(A[i,i+1], A[i,i+2], \ldots A[i,n])$, and let S_i be the column vector $(A[i+1,i], A[i+2,i], \ldots A[n,i])$.

Theorem 23 [Berkowitz 1984]. The coefficients $\chi(A)$ of the characteristic polynomial of A can be computed in time $O(\log^2 n)$ using $O(M(n) \cdot n)$ processors.

Proof: Define C_i, an $(n-i+2) \times (n-i+1)$ dimensional lower triangular Toeplitz matrix, as follows. $C_i[1,1] = -1$; $C_i[2,1] = A[i,i]$; for each j greater than 2, $C_i[j,1] = R_i \times M_i^{j-3} \times S_i$. By theorem 22, we can calculate the coefficients of each C_i in time $O(\log^2 n)$ using $M(n)$ processors, so we may calculate all of the C_i using a factor of n more processors. Berkowitz [1984] showed that $\chi(A) = \prod_{i=1}^{n} C_i$; this can be calculated in time $O(\log^2 n)$ using $O(n^3/\log^2 n)$ processors, which by assumption is less than $O(M(n) \cdot n)$. •

Theorem 24 [Berkowitz 1984]. The determinant and inverse of an $n \times n$ matrix over an arbitrary field can be calculated in time $O(\log^2 n)$ with $O(nM(n))$ processors.

Proof: The determinant is simply the constant term p_n of the characteristic polynomial. By the Cayley-Hamilton theorem, we may compute A^{-1} as

$$A^{-1} = \frac{-1}{p_n} \sum_{i=0}^{n-1} A^{n-i-1} p_i.$$

Again the computation can be performed in time $O(\log^2 n)$ using $O(M(n) \cdot n)$ processors. •

As with the other matrix computations described earlier, the computation of the determinant and inverse described above can be made more efficient for certain restrictions on the field the matrix cell values are taken from. In particular, if the field has characteristic 0, Preparata and Sarwati [1978] improve an algorithm of Csanky [1976] to achieve the same time using $O(M(n)\sqrt{n})$ processors; Galil and Pan [1985b] give a small improvement to this result.

If the entries in the matrices are k-bit integers or quotients of k-bit integers, the inverse contains numbers which may be longer by a factor of $O(n \log n)$, so the assumption of unit time arithmetic operations no longer holds. Galil and Pan [1985a] give a randomized algorithm for inverting matrices of k-bit rational numbers in time $O(\log^2 n)$ using $O(knM(n))$ processors, using only single bit operations.

4.2 Closed Semiring Systems

Let S be a semiring as defined above, but with the following further properties. First, the addition property should be idempotent; that is, for any a, $a + a = a$. Second, any countably infinite sum $a_1 + a_2 + \cdots + a_i + \cdots$ should have a unique solution; further, the usual commutative, associative, and distributive laws should apply to infinite as well as finite sums. We call such a system a closed semiring [Aho et al. 1974]. By the properties of a closed semiring, the following function, known as closure, is well defined. For any element a in the ring, let a^*, the closure of a, be $1+a+a^2+a^3+\cdots$, where 1 is the multiplicative identity of the semiring. We assume that closure as well as addition and multiplication can be performed in constant time by a single processor.

A closed semiring system consists of a semiring, together with a directed graph labeled on the edges with elements of the semiring. We define the label of a path in the graph to be the product in the semiring of all the labels of the edges of the path, in order as they appear in the path. The problem we wish to solve is, given such a labeled graph, to find the closure of the graph; that is, for each pair of vertices, the sum of the labels on all possible paths between those two vertices.

Note that we don't require the paths to be simple, so the sum as described above is infinite. Nevertheless it is possible to compute this sum for all pairs of vertices of the graph sequentially in time $O(n^3)$. In fact, this may be computed in the same sequential time as a single matrix multiplication over the given semiring [Aho et al. 1974]. The sequential algorithm for solving closed semiring systems uses a form of dynamic programming, and seems to be inherently sequential.

We describe below a parallel algorithm, due to Kučera, that works for those semirings in which, for all a, $1 + a = 1$ (so $a^* = 1$). This algorithm uses a number of operations that is within a logarithmic factor of the best known sequential time for these problems. It is not clear whether it could be extended to a parallel algorithm for more general closed semiring systems.

Theorem 25 [Kučera 1982]. Assume that the given semiring has the property that, for all a, $1 + a = 1$, and denote the parallel time and number of processors for multiplication of $n \times n$ matrices over the semiring by $T(n)$ and $M(n)$. Then we can perform a closed semiring system computation on a graph of n vertices in parallel time $T(n) \log n$, using $M(n)$ processors.

Proof: Let A be the $n \times n$ matrix constructed as follows. For each edge (i, j) in the graph, labeled with some value v, we let $A[i, j] = v$. For each i, j with no edge between them, we let $A[i, j] = 0$, the additive identity. For each i, $A[i, i] = 1$, the multiplicative identity. Let m be the least power of 2 greater than or equal to n, and let $B = A^m$, computed in $\log n$ multiplications by repeated squaring.

In fact the algorithm described by Kučera [1982] repeats, instead of squaring, the following step: $A \leftarrow A + A^2$. But it can be seen that, in the algorithm we give, $A^k[i, i] = 1$ always, and so the extra addition performs no useful work.

We claim that for each k, $A^k[i, j]$ will hold the sum of all path labels for paths from i to j using at most 2^k edges. Clearly this is the case for $k = 0$. For $k > 0$, note that each path from i to j having at most 2^k edges can be formed as a path from i to some vertex h having at most 2^{k-1} edges, together with another path from h to j again having at most 2^{k-1} edges. $A^k[i, j] = \sum_{h=1}^{n}(A^{k-1}[i, h] + A^{k-1}[h, j])$, which by induction is the sum of the labels of all ways of combining two paths of length at most 2^{k-1}. The matrix multiplication may compute the label for a given path in several different ways, but by idempotence the sum of any number of copies of the path label is that path label itself.

Finally, note that, again using the fact that $1 + a = 1$, the sum of the label of any non-simple path with that of a simple path using a subset of the same edges will be the label of the simple path; therefore since all simple paths use at most $n - 1$ edges we need not take A to any higher power. •

Now let us describe some applications of closed semiring systems. First consider finding the shortest path between each pair of vertices in the graph, and assume the edge lengths are given as $\log n$-bit integers. The corresponding semiring has as its addition operation the integer minimum function, and as its multiplication operation integer addition. The multiplicative identity is the integer 0; we add a special infinite value to be our additive identity.

Matrix multiplication in the above semiring can be seen to take constant time using $O(n^3)$ weak CRCW processors. Therefore we can find all shortest paths in the graph in time $O(\log n)$, and the same number of processors.

As a second example, consider finding the transitive closure of a directed graph. We take as our semiring Boolean algebra: the addition operator will be logical or and the multiplication operator logical and. The additive identity is the value representing falsehood, and the multiplicative identity that representing truth. Again matrix multiplication can be performed in constant time with $O(n^3)$ weak CRCW processors, so finding the transitive closure takes $O(\log n)$ time with the same number of processors. Essentially this algorithm, in a version for EREW processors, was given by Hirschberg [1976].

As a less obvious example of the use of semiring systems, consider finding a topological ordering of a directed acyclic graph. This can easily be performed in linear time sequentially, but the algorithm does not obviously lend itself to parallelism. The following algorithm, due to Kučera [1982], does find a topological ordering in parallel; however the efficiency of the algorithm is much less than that of the sequential topological sorting algorithm.

We first find the transitive closure of the graph, using the above semiring algorithm. For each vertex v, we compute the in-degree of v in the transitive closure. Finally we sort the vertices by their computed in-degrees. If a vertex v is an ancestor of another vertex w in the original graph, then in the transitive closure w will have incoming edges from all the ancestors of v, plus one from v itself, with possibly still others. Therefore sorting by in-degrees in fact results in a topological ordering. Each step can be performed in $O(\log n)$ time; the step that requires the most processors is that of computing the transitive closure, which as we have seen takes $O(n^3)$ weak CRCWs.

4.3 Matching

Another important problem that can be solved in parallel using matrix techniques is that of matching. A matching is a subset M of the edges of an undirected graph, such that no two edges in M share a vertex. There are a number of closely related problems one would like solved, all involving finding matchings of a certain sort. In particular, we might want to find any of the following types of matching.

(1) A perfect matching. Each vertex must appear in some edge of the matching.

(2) A maximum matching. This is a matching with the largest number of edges over all possible matchings in the graph. If there is a perfect matching, it is obviously a maximum matching, but there are graphs in which no perfect matching exists.

(3) A maximal matching. With such a matching, each edge of the graph has at least one of its vertices covered by the matching, so no more edges can be added to the matching. A maximum matching must be maximal, but the converse is not necessarily true.

(4) A minimum (or maximum) weight perfect matching. Each edge is given an integer weight; the task is to find a matching such that the sum of the edge weights is minimized.

(5) A minimum weight perfect matching, assuming that there is only one perfect matching having the minimum weight.

In problems (4) and (5) we restrict the weights to be integers bounded by a polynomial in the input size; it is not clear whether the problems can be solved quickly and efficiently in parallel for more general weights.

Matching is interesting in its own right, but it is also closely related to a number of other problems. In particular if we can calculate maximum flows on a bipartite graph, we can use that calculation to find a maximum matching on that graph; and conversely matchings can be used to find certain types of flows. Aggarwal and Anderson [1987] used flows calculated by matchings in their algorithm for computing a depth first search tree. If we could compute matchings more efficiently, we could in turn use that computation to compute depth first search trees more efficiently.

Here we describe a solution to problems (1) and (5), due to Mulmuley et al. [1987]. Problem (1) is reduced to problem (5) by an appropriate choice of relatively small random weights; then problem (5) is solved deterministically using inversion of a matrix derived from the edge weights. We should note that the same paper also describes a solution of problems (2) and (4) using similar techniques.

Problem (3), finding a maximal matching, seems to be much easier; it can be solved efficiently and deterministically in parallel with an algorithm based on the Euler tour technique for graphs [Israeli and Shiloach 1986].

Finally we should note that an important special case of all of the above problems is finding the appropriate matching for a bipartite graph. We shall not discuss this case further here.

Theorem 26 [Mulmuley et al. 1987]. Let $S = \{x_1, x_2, \ldots, x_n\}$ be a finite set, and let $F = \{S_1, S_2, \ldots S_k\}$, with $S_j \subset S$, be a family of subset of S. Let weights w_i, drawn randomly and independently from a uniform distribution on the integers from 1 to $2n$, be assigned to each of the x_i, and define the weight $w(S_j)$ of a set in F to be the sum of the weights of its members. Then the probability that there is exactly one member of F having the minimum weight is at least 1/2.

Proof: Let P be the probability that more than one minimum weight matching exists. Let E_i be the event that x_i is both in some minimum weight matching and not in some other minimum weight matching, and let P_i be the probability of event E_i. We first show a bound on P_i, and then from this derive a bound on P.

Assume a fixed assignment of weights w_i. For each i, define F_i to be the collection of sets from F containing x_i. Let $m(X)$ for $X \subset F$ be the minimum weight set in X. Then x_i can only be included in some minimum weight set and not included in some other such set when $m(F - F_i) = m(F_i)$; or equivalently when $w_i = m(F - F_i) - (m(F_i) - w_i)$. But w_i is independent of both $m(F - F_i)$ and $m(F_i) - w_i$, so P_i, the probability of w_i being chosen to equal their difference, is at most $1/2n$.

If there are two or more minimum weight sets in F, they must differ in at least one x_i; that is, at least one of the events E_i must occur. These events are not independent, but the probability of any of them occuring can be no more than the sum of their individual probabilities. That is, $P \leq \sum_{i=1}^{n} P_i \leq n/2n = 1/2$. •

Corollary. If we can solve problem (5) above in time t using p CRCW processors, we can solve problem (1) in expected time $O(t)$ using the same number of processors.

Proof: Assume the graph has n vertices and m edges. We give each edge a random weight from 1 to $2m$; this takes linear operations, so it must be within the bounds for solving problem (5). By the theorem above, there is a unique minimum weighted matching with probability at least 1/2. Therefore with probability 1/2 the algorithm for problem (5) finds a perfect matching; if the minimum matching is not unique we don't know what the algorithm will do, but we can stop it after the appropriate amount of time and check in linear (CRCW) operations whether it has in fact found a perfect matching. Again, since we assume nothing about the algorithm for problem (5), if the minimum matching is not unique it might attempt some concurrent memory operations that would not happen otherwise; but with CRCW processors this can cause no harm. The expected number of times we need perform this sequence of operations is at most 2. •

Now assume we have a graph G with n vertices and m edges, each edge (i, j) marked with an integer weight $1 \leq w_{i,j} \leq k$, and that there is exactly one minimum weight perfect matching in G. We now show, following Mulmuley et al. [1987], how to find that perfect matching in parallel.

First define the skew-symmetric matrix B as follows. If (i, j) is not an edge of G, or if $i = j$, then $B[i, j] = 0$. Otherwise, if $i < j$, $B[i, j] = 2^{w_{i,j}}$, or if $i > j$, $B[i, j] = -2^{w_{i,j}}$. The value at each cell is an integer representable in $O(k)$ binary digits. Also, let w be the weight of the minimum weight perfect matching M of G.

Theorem 27 [Mulmuley et al. 1987]. If G has a unique minimum weight perfect matching, and B is defined as above, then the determinant of B is divisible by 2^{2w}, and by no higher power of 2.

Proof: For each permutation σ of the numbers from 1 to n, let $value(\sigma) = \pm \prod_{i=1}^{n} B[i, \sigma(i)]$, with the sign positive if σ is an even permutation, or negative if it is odd. Then $\det(B) = \sum_\sigma value(\sigma)$.

Now let σ be an arbitrary permutation. If σ takes any number i to itself, then $value(\sigma)$ includes $B[i, i] = 0$ and is therefore zero. If σ contains an odd cycle of length more than one, then it can be paired with another permutation σ' which is identical except that it contains the inverse cycle. If there is more than one odd cycle we choose the one containing the smallest of the numbers being permuted; in this way we make the pairing of permutations one-to-one. Where one cycle contributes $B[i, j]$ to the value of its permutation, the one it is paired with contributes $B[j, i] = -B[i, j]$. Because the cycles are odd, the total number of such terms contributed by either cycle is odd, so $value(\sigma) = -value(\sigma')$, and the two values cancel out in $\det(B)$.

The remaining case is that σ contains only even cycles. Then split each cycle into odd and even edges; this gives rise to two perfect matchings M_1 and M_2 such that $value(\sigma) = \pm 2^{w(M_1)+w(M_2)}$. If $M_1 = M_2 = M$ is the minimum weight matching, with weight w, then $value(\sigma) = \pm 2^{2w}$, and σ is the permutation taking each vertex to its match in M. Only one such permutation exists because of the assumption that the minimum weight perfect matching is unique. Otherwise, $w(M_1) + w(M_2) > 2w$.

We have shown that $\det(B)$ is the sum of a number of terms, some of which cancel, some of which are divisible by a higher power of 2 than $2w$, and exactly one of which is $\pm 2^{2w}$. Therefore the highest power of 2 dividing $\det(B)$ is as claimed exactly $2w$. •

Now define $B_{i,j}$ to be the matrix formed by removing row i and column j from matrix B. The adjoint adj(B) is defined by adj$(B)[j, i] = \det(B_{i,j})$. It turns out that $B\,\text{adj}(B) = \det(B) \cdot I$, so the adjoint can be found by omitting the division by the determinant from the matrix inversion algorithm described earlier. In this particular case, we are better off using the matrix inversion algorithm of Galil and Pan [1985a], which also computes the adjoint before dividing it by the determinant.

Theorem 28 [Mulmuley et al. 1987]. An edge (i, j) is in the unique minimum weight perfect matching M, having weight w, exactly when $\det(B_{i,j}) 2^{w_{i,j}} / 2^{2w}$ is odd.

Proof: Assume without loss of generality that $i < j$. Note that $\det(B_{i,j}) 2^{w_{i,j}} = \sum_\sigma value(\sigma)$, where the sum is over those permutations σ such that $\sigma(i) = j$.

As in the previous theorem, those σ taking some k to itself contribute nothing to the sum. If some σ has an odd cycle of length greater than one, then because n is even either σ takes some

k to itself or it contains another odd cycle. In the latter case, at most one of the odd cycles can contain i and j. We find the odd cycle not containing i and j and having as a member of the cycle the smallest of the numbers from 1 to n among all such odd cycles, and pair σ with another permutation obtained by reversing this cycle; again as in the previous theorem, this permutation exactly cancels $value(\sigma)$.

Finally if σ has only even cycles, then again as in the previous theorem it can be decomposed into two matchings, and as before it will contribute 2^{2w} exactly for the permutation corresponding to the minimum weight perfect matching (if that matching contains edge (i,j)) and higher powers of two for all other permutations.

Therefore, if $(i,j) \in M$, the highest power of 2 dividing $\det(B_{i,j})2^{w_{i,j}}$ will be exactly $2w$, and if $(i,j) \notin M$, then $\det(B_{i,j})2^{w_{i,j}}$ will be divisible by a higher power of 2 than $2w$. •

The above theorems lead us to the following algorithm, for finding a minimum weight perfect matching whenever this matching is unique:

(1) Compute B, $\det(B)$, and $\operatorname{adj}(B)$.

(2) Find w from $\det(B)$ as in theorem 27.

(3) Determine for each edge (i,j) whether it is in the minimum weight perfect matching using theorem 28.

Theorem 29 [Mulmuley et al. 1987]. A unique minimum weight perfect matching on a graph with n vertices, with integer edge weights from 1 to k, can be found in expected time $O(\log^2 n)$ using $O(knM(n))$ CRCW processors.

Proof: We use the integer matrix inversion algorithm of Galil and Pan [1985a], which computes the determinant and adjoint of B in the above bounds. The remaining steps may easily be performed in the same bounds. •

Corollary. A perfect matching of an undirected graph, if one exists, may be found in expected time $O(\log^2 n)$ using $O(mnM(n))$ CRCW processors.

We should note that Galil and Pan [1985a] give a different perfect matching algorithm, based on that of Karp et al. [1985a], which takes $O(\log^3 n)$ expected time with $O(nM(n))$ processors. Neither algorithm is as efficient as the best sequential algorithm, which takes $O(m\sqrt{n})$ time [Micali and Vazirani 1980].

As we previously noted, Israeli and Shiloach [1986] gave an algorithm for maximal matching which takes $O(\log^3 m)$ time with $O(m+n)$ processors. Maximum matching can be solved by first computing the size of the matching from the rank of the adjacency matrix, and then computing a perfect matching on a graph derived from the original graph; the resulting algorithm takes the same bounds as perfect matching. The minimum weight perfect matching problem can be solved using a factor of $O(nw)$ more processors than needed to find a perfect matching [Galil 1986b; Mulmuley et al. 1987]; here w is the largest weight on any edge.

5. CONCLUSIONS

We have described a number of algorithmic tools that have been found useful in the construction of parallel algorithms; among these are prefix computation, ranking, Euler tours, ear decomposition, and matrix calculations. We have also described some of the applications of these tools, and listed many other applications.

The algorithms and algorithmic tools we have given seem likely to be useful not only in their own right for the problems they solve, but also as examples of ways to break up other problems into parts suitable for parallel solution.

A number of open problems remain. It is likely that many of the problems for which we have described algorithms can be solved more quickly, more efficiently, or with a weaker type of PRAM. In particular, we would like to reduce the processor bounds and remove the requirement for random numbers of the algorithms for matching and finding depth first search trees. In the weighted matching algorithms, we would also like to remove the linear dependence on the weights from the bound on the number of processors. Another algorithm we gave that was much less efficient than its sequential counterpart was for topologically sorting a directed acyclic graph; perhaps this problem can be solved more efficiently. A related problem is that of finding shortest paths from a single source in a graph; the best known parallel solution is no better than finding the paths from all sources, using the parallel closed semiring system algorithm we gave. This algorithm itself solves an important special case of closed semiring systems; can the general case of these systems be solved quickly and efficiency in parallel?

REFERENCES

AGGARWAL, A., AND ANDERSON, R. 1987. A Random NC Algorithm for Depth First Search. In *Proceedings of the 19th Annual Symposium on Theory of Computing.* ACM, New York, p. 325.

AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. 1974. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading, Mass.

AHO, A.V., AND ULLMAN, J.D. 1977. *Principles of Compiler Design.* Addison Wesley, Reading, Mass.

AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. 1983. An $O(n \log n)$ Sorting Network. In *Proceedings of the 15th Annual Symposium on Theory of Computing.* ACM, New York, pp. 1-9.

ANDERSON, R. 1985. A Parallel Algorithm for the Maximal Path Problem. In *Proceedings of the 17th Annual Symposium on Theory of Computing.* ACM, New York, pp. 33-37.

ATALLAH, M.J., AND VISHKIN, U. 1984. Finding Euler Tours in Parallel. *J. Comput. Sys. Sci. 29*, pp. 330-337.

AWERBUCH, B., ISRAELI, A., AND SHILOACH, Y. 1984. Finding Euler Circuits in Logarithmic Parallel Time. In *Proceedings of the 16th Annual Symposium on Theory of Computing.* ACM, New York, pp. 249-257.

BERKOWITZ, S.J. 1984. On Computing the Determinant in Small Parallel Time Using a Small Number of Processors. *Info. Proc. Letters 18*, pp. 147-150.

BRENT, R.P. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM 21*, 2 (April), pp. 201-206.

COLE, R. 1986. Parallel Merge Sort. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, IEEE, New York, pp. 511-516.

COLE, R., AND VISHKIN, U. 1986a. Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms. In *Proceedings of the 18th Annual Symposium on Theory of Computing.* ACM, New York, pp. 206-219.

COLE, R., AND VISHKIN, U. 1986b. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Info. and Control 70*, pp. 32-53.

COLE, R., AND VISHKIN, U. 1986c. Approximate and Exact Parallel Scheduling with Applications to List, Tree and Graph Problems. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 478-491.

COLE, R., AND VISHKIN, U. 1987. The Accelerated Centroid Decomposition Technique for Optimal Parallel Tree Evaluation in Logarithmic Time. Manuscript.

COPPERSMITH, D., AND WINOGRAD, S. 1987. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of the 19th Annual Symposium on Theory of Computing.* ACM, New York, pp. 1-6.

CSANKY, L. 1976. Fast Parallel Matrix Inversion Algorithms. *SIAM J. Comput. 5*, 4 (December). pp. 618–623.

EVEN, S., AND TARJAN, R.E. 1976. Computing an *st*-numbering. *Theoretical Computer Science 2*, pp. 339–344.

FICH, F.E., RAGDE, R.L., AND WIGDERSON, A. 1983. Relations Between Concurrent-Write Models of Parallel Computation. Manuscript.

GALIL, Z. 1986a. Optimal Parallel Algorithms for String Matching. *Info. and Control 67*, pp. 144–157. A preliminary version appeared in *Proceedings of the 16th Annual Symposium on Theory of Computing*, 1984. ACM, New York, pp. 240–248.

GALIL, Z. 1986b. Sequential and Parallel Algorithms for Finding Maximum Matchings in Graphs. *Ann. Rev. Comput. Sci. 1*, pp. 197–224.

GALIL, Z., AND PAN, V. 1985a. Improved Processor Bounds for Algebraic and Combinatorial Problems in RNC. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 490–495.

GALIL, Z., AND PAN, V. 1985b. Improving the Efficiency of Parallel Algorithms for the Evaluation of the Determinant and the Inverse of a Matrix. Tech. Rep. 85-5, Computer Science Dept., SUNY, Albany, New York.

GAZIT, H. 1986. An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 492–501.

GOLDBERG, A., PLOTKIN, S., AND SHANNON, G. 1987. Parallel Symmetry-Breaking in Sparse Graphs. In *Proceedings of the 19th Annual Symposium on Theory of Computing.* ACM, New York, pp. 315–324.

GOLDBERG, M., AND SPENCER, T. 1987. A New Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 161–165.

GROLMUSZ, V., AND RAGDE, P. 1987. Incomparibility in Parallel Computation. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 89–98.

HILLIS, W.D., AND STEELE, G.L. 1986. Data Parallel Algorithms. *Commun. ACM 29*, 12 (December), pp. 1170–1183.

HIRSCHBERG, D.S. 1976. Parallel Algorithms for the Transitive Closure and the Connected Component Problems. In *Proceedings of the 8th Annual Symposium on Theory of Computing.* ACM, New York, pp. 55–57.

ISRAELI, A., AND SHILOACH, Y. 1986. An Improved Parallel Algorithm for Maximal Matching. *Info. Proc. Letters 22*, pp. 57–60.

KANEVSKY, A., AND RAMACHANDRAN, V. 1987. Improved Algorithms for Graph Four-Connectivity. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 252–259.

KARLIN, A.R., AND UPFAL, E. 1986. Parallel Hashing - An Efficient Implementation of Shared Memory. In *Proceedings of the 18th Annual Symposium on Theory of Computing.* ACM, New York, pp. 160–168.

KARP, R.M., UPFAL, E., AND WIGDERSON, A. 1985a. Constructing a Perfect Matching is in Random NC. In *Proceedings of the 17th Annual Symposium on Theory of Computing.* ACM, New York, pp. 22–32.

KARP, R.M., UPFAL, E., AND WIGDERSON, A. 1985b. Are Search and Decision Problems Computationally Equivalent? In *Proceedings of the 17th Annual Symposium on Theory of Computing.* ACM, New York, pp. 464–483.

KLEIN, P.N., AND REIF, J.H. 1986. An Efficient Parallel Algorithm for Planarity. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 465–477.

KRAPCHENKO, A.N. 1970. Asymptotic Estimation of Addition Time of a Parallel Adder. English translation in *Syst. Theory Res. 19*, pp. 105–122.

KUČERA, L. 1982. Parallel Computation and Conflicts in Memory Access. *Info. Proc. Letters 14*, 2 (April), pp. 93–96.

LADNER, R.E., AND FISCHER, M.J. 1980. Parallel Prefix Computation. *J. ACM 27*, 4 (October), pp. 831–838.

LINIAL, N., LOVÁSZ, L., AND WIGDERSON, A. 1986. A Geometric Interpretation of Graph Connectivity and its Algorithmic Applications. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 39–48.

LOVÁSZ, L. 1985. Computing Ears and Branchings in Parallel. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science.* IEEE, New York, pp. 464–467.

LUBY, M. 1985. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the 17th Annual Symposium on Theory of Computing.* ACM, New York, pp. 1–10.

MAON, Y., SCHIEBER, B., AND VISHKIN, U. 1986. Parallel Ear Decomposition Search (EDS) and ST-Numbering in Graphs. In *VLSI Algorithms and Architectures*, Lecture Notes in Computer Science 227, Springer-Verlag, Berlin, pp. 34–45.

MEGIDDO, N. 1983. Applying Parallel Computation Algorithms in the Design of Serial Algorithms. *J. ACM 30*, 4 (October), pp. 852–865.

MEHLHORN, K., AND VISHKIN, U. 1984. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica 21*, pp. 339–374.

MICALI, S., AND VAZIRANI, V.V. 1980. An $O(\sqrt{|V|}|E|)$ Algorithm for Finding Maximum Matchings in General Graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*. IEEE, New York, pp. 17–27.

MILLER, G.L., AND RAMACHANDRAN, V. 1987. A New Graph Triconnectivity Algorithm and Its Parallelization. In *Proceedings of the 19th Annual Symposium on Theory of Computing*. ACM, New York, pp. 335–344.

MILLER, G.L., AND REIF, J.H. 1985. Parallel Tree Contraction and its Application. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*. IEEE, New York, pp. 478–489.

MULMULEY, K., VAZIRANI, U.V., AND VAZIRANI, V.V. 1987. Matching is as Easy as Matrix Inversion. *Combinatorica 7*, 1, pp. 105–114. A preliminary version appeared in *Proceedings of the 19th Annual Symposium on Theory of Computing*. ACM, New York, pp. 345–354.

PREPARATA, F.P., AND SARWATI, D.V. 1978. An Improved Parallel Processor Bound in Fast Matrix Inversion. *Info. Proc. Letters 7*, 3 (April), pp. 148–150.

RANADE, A.G. 1987. How to Emulate Shared Memory. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. IEEE, New York, pp. 185–194.

REIF, J.H. 1983. Probabilistic Parallel Prefix Computation. *J. ACM*, submitted.

REIF, J.H. 1985. An Optimal Parallel Algorithm for Integer Sorting. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*. IEEE, New York, pp. 496–504.

REISCHUK, R. 1981. A Fast Probabilistic Parallel Sorting Algorithm. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. IEEE, New York, pp. 212–219.

SCHIEBER, B. 1987. Design and Analysis of some Parallel Algorithms. Ph.D. Dissertation, Tel Aviv University.

SCHIEBER, B., AND VISHKIN, U. 1987. On Finding Lowest Common Ancestors: Simplification and Parallelization. Ultracomputer Note 118, Courant Institute, New York University, New York.

SCHWARTZ, J.T. 1980. Ultracomputers. *ACM Trans. Prog. Lang. and Sys. 2*, pp. 484–521.

SCHILOACH, Y., AND VISHKIN, U. 1981. Finding the Maximum, Merging and Sorting in a Parallel Computation Model. *J. Algorithms 2*, pp. 88–102.

TARJAN, R.E. 1972. Depth-first Search and Linear Graph Algorithms. *SIAM J. Comput. 15*, 3, pp. 814–830.

TARJAN, R.E., AND VISHKIN, U. 1984. Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*. IEEE, New York, pp. 12–20

TSIN, Y.H. 1985. An Optimal Parallel Processor Bound in Strong Orientation of an Undirected Graph. *Info. Proc. Letters 20*, pp. 143–146.

VALIANT, L.G. 1975. Parallelism in Comparison Problems. *SIAM J. Comput. 4*, 3 (September), pp. 348–355.

VISHKIN, U. 1984a. A Parallel-Design Distributed-Implementation (PDDI) General-Purpose Computer. *Theor. Comput. Sci. 32*, pp. 157–172.

VISHKIN, U. 1984b. On Choice of a Model of Parallel Computation. Manuscript.

VISHKIN, U. 1984c. Randomized Speed-ups in Parallel Computation. In *Proceedings of the 16th Annual Symposium on Theory of Computing.* ACM, New York, pp. 230–239.

VISHKIN, U. 1985. On Efficient Parallel Strong Orientation. *Info. Proc. Letters 20*, pp. 235–240.

VON ZUR GATHEN, J. 1984. Parallel Algebraic Computations. Unpublished course notes.

WHITNEY, H. 1932. Non-Separable and Planar Graphs. *Trans. Amer. Math. Soc. 34*, pp. 339–362.

WINOGRAD, S. 1975. On the Evaluation of Certain Arithmetic Expressions. *J. ACM 22*, 4 (October), pp. 477–492.

WYLLIE, J.C. 1979. The Complexity of Parallel Computation. Tech. Rep. TR 79-387, Dept. of Computer Science, Cornell University, Ithaca, New York.