Parallel Algorithmic Techniques for Combinatorial Computation

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

[1] Computer Science Department, Columbia University, New York, NY 10027
[2] Computer Science Department, Tel Aviv University, Tel Aviv, Israel

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.

A drawback to the use of parallel computers is that they are harder to program. For this reason, parallel computation is 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 important problems are more complex. In particular, problems may be structured as graphs or trees, rather than in the regular order of a matrix.

We describe a number of techniques that have been developed for solving such combinatorial problems. Our intent is to show how these tools can be used as building blocks for higher level algorithms, and to provide pointers to the literature for the details 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 particular we give very little attention to sorting, although it is used as a subroutine in a number of the algorithms we describe.

We use a shared memory model of parallelism; nevertheless we hope that the techniques described will be useful not only on shared memory machines, but also on other types of parallel computers, either through simulations of shared memory [48, 70, 35, 57], or through analogous techniques for different models of parallel computation (e.g. see [30]).

1. THE MODEL OF PARALLELISM

The model of parallel computation we use is the shared memory parallel random access machine (PRAM). This model consists of a collection of identical processors and a separate collection of memory cells; any processor can access any memory cell in unit time. Processors are allowed to perform any other operation that a typical sequential processor could do. Processors can know the size of their input, and the number of processors is assumed to be a function of that size, but the program controlling the processors is the same for all input sizes.

Practically, one would like a PRAM algorithm to be as efficient as possible; that is, we want to minimize the number of operations, which may be computed as the product of the algorithm time and the number of processors. Clearly there must be at least as many operations as the best known sequential time; a parallel algorithm that meets this bound is called *optimal*. A less restrictive condition is that the operations are within a polylogarithmic factor of optimality; we call such algorithms *almost optimal*.

Theoretically, we would like as small a running time as possible. In particular, an important class of algorithms, NC, is defined to be those taking polylogarithmic time with polynomially many processors.

Many PRAM algorithms are both in NC and almost optimal. 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. We restrict our attention to NC algorithms, but we attempt to keep the number of processors small enough that the algorithms remain useful in practice.

1.1 Types of Concurrent Memory Access

An important distinction among PRAM models is how to resolve attempts by several processors to access the same memory cell simultaneously. It is convenient to pick different resolutions to such conflicts for different algorithms; a number of simulation results let algorithms written for one choice work with machines that use a different choice.

One 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 (EREW). This is the weakest of the various types of PRAM, so an algorithm that works in this model is preferable to one that achieves the same time and processor bounds on a stronger model.

The next type of PRAM, known as a concurrent read exclusive write (CREW), 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.

In the strongest version of the PRAM, the concurrent read concurrent write machine (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; we now summarize some of the possibilities.

(1) Weak CRCW. Concurrent writes are allowed only if all values being written are zero. Sometimes such writes can only occur in a set of special cells that can only contain zero or one; but this restriction turns out not to affect the power of the model.

(2) Common-mode CRCW. 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; an arbitrary one of the values is written. The result may be different if the same step is executed another time.

(4) Priority CRCW. The processors have unique priorities, assigned arbitrarily. The result of a concurrent write is the value from the writer with the largest priority. It is not clear a priori which value will be chosen as the result; however a repetition of the same concurrent write will give the same result.

(5) Strong CRCW. The value written in any concurrent write is the largest (or equivalently smallest) of any of the values the processors are attempting to write.

Each of these models is at least as strong as the previous ones, so the various models form a hierarchy. Grolmusz and Ragde [28] describe other CRCW models intermediate in power among the ones we have listed, which do not fall into a hierarchy with them.

1.2 Randomized Parallel Algorithms

We sometimes allow randomness in parallel computation, by giving each processor its own random number generator. The distribution of numbers drawn at any one processor is independant of that drawn at all other processors. The algorithm's time bound is then the expected time of termination for the worst case input of a given size.

An important use of randomness is symmetry breaking; for instance, we may need to assign different colors to vertices of a graph having similar local neighborhoods. By flipping coins we can with high probability get different results at each vertex. We show later some cases in which these coin flips can be replaced by deterministic techniques.

Randomness can improve the bounds of an algorithm in several ways. First, a randomized algorithm may solve in polylogarithmic time a problem for which there is no known NC algorithm. This is the case for matching [36, 23, 53] and for the construction of depth first search trees [1]. Karp et al. [37] describe a problem in a model of parallel computation with oracles which no deterministic PRAM can solve in NC, but which can be solved using a probabilistic algorithm.

Second, a randomized parallel algorithm may be more efficient than the best deterministic algorithm, may take less time, or may use a weaker model. Examples of improved efficiency in randomized parallel computation are various algorithms for finding maximal independent sets: the randomized algorithm of Luby [45] uses less operations than the deterministic algorithm of the same paper, and runs more quickly than the newer deterministic algorithm of Goldberg and Spencer [27]. Other problems with randomized algorithms more efficient than their deterministic counterparts include finding connected components [24], and sorting integers [59].

Finally, a randomized algorithm may have the same bounds as a deterministic algorithm, but may be much simpler. Also, randomized algorithms may be discovered before any equally efficient deterministic solution. Examples include algorithms for list ranking and for tree contraction, discovered first in randomized versions [52, 71], and later made deterministic [6, 14, 15, 25].

1.3 Simulations Among PRAM Models

An algorithm designed for a weak PRAM model can clearly be used in a stronger model. But we would like to use any PRAM algorithms on any model, so it is important to simulate stronger models by weaker ones. A particularly important case is when the simulating PRAM uses the same type of memory access, but has fewer processors.

Theorem 1 [9]. If a parallel computation can be performed in time t, using q operations on any number of PRAM processors of a given type, then it can be performed in time $t + (q - t)/p$ using p processors of the same type.

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

The theorem assumes we can split the operations of a given time step into blocks of a given size, and assign processors to the operations within each block. In most applications this scheduling problem is easy, although this is not always true. The corollary follows without assumption.

Now let us look at simulations between different types of PRAM. In all these results, a simulated NC algorithm remains in NC.

Theorem 2 [4, 54, 69]. 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.

Theorem 3. A parallel computation that can be performed in time t, using p strong CRCW processors, can also be performed in time (a) $O(t)$, using $O(p^2)$ weak CRCW processors [41]; (b) $O(t \log \log p)$, using $O(p)$ arbitrary-winner processors [10]; (c) $O(t \log p)$, using $O(p/ \log p)$ probabilistic arbitrary-winner processors [59]; and (d) $O(t \log p/(k \log(\log p)/k))$, using kn common-mode processors [11].

As special cases of (d), common-mode CRCWs can take time $O(t \log p/ \log \log p)$ with $O(p)$ processors [20], or time $O(t)$ with $O(p \log p)$ processors [10].

These results can be improved if we know the pattern of concurrent memory access. In particular, many uses of strong concurrent write reduce to finding a single maximum.

Theorem 4 [68, 19, 63]. 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, if the values can be represented by $O(\log n)$ bits each.

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, $\ldots v_n$ and an associative binary operation $\circ$ on those values. The result of the computation is a sequence of prefix sums $p_1 = v_1$, $p_2 = v_1 \circ v_2$, $\ldots p_n = v_1 \circ v_2 \circ \cdots \circ v_n$. A simple algorithm for prefix computation was discovered by Ladner and Fischer [42].

(1) For each v_i, if i is even, calculate $x_{i/2} = v_{i-1} \circ v_i$.

(2) Recursively calculate the prefix sums of the $\lfloor n/2 \rfloor$ values of x_i into y_i.

(3) For each 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 [42]. 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.

Some prefix operations can be performed even more efficiently. Prefix maximum may be performed in $O(\log \log n)$ time, using linear weak CRCW operations [60]. Reif [58] showed that certain prefix computations can be performed in $O(\log \log n)$ expected time, again with $O(n)$ operations. The result is deterministic, but for the average case in which the inputs are drawn from a random distribution. It does not work for prefix addition, but it does for other operations, including one used to construct circuits for bitwise binary addition.

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 show later how to deal with linked lists. Another possible representation of a list is as a binary tree with the sequence stored at the leaves. The algorithm above can be adapted to perform prefix computations on such a representation in time proportional to the height of the tree.

2.2 Applications of Prefix Computation

One of the most important applications of prefix computation occurs in theorem 2, the simulation of concurrent memory access by exclusive access. Another application is in list manipulation [62]. Assume that we have a list of data values, represented as an array. Certain members of the list are marked; 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. The result of a prefix sum is, for each marked value, the position that value has in the sublist.

Prefix computation may also be used to simulate the action of a finite state automaton. A typical use for such simulation would be lexical analysis in a compiler [3]. To calculate the state of an automaton after each prefix of the input, first replace each input character by the corresponding transition function of the automaton. Now compose these functions in a prefix computation. The result of these prefix composition functions applied to the initial state gives the state of the automaton after each input string prefix.

This algorithm for finite automaton simulation was used by Ladner and Fischer [42] to define Boolean circuits of size $O(n)$ for adding two binary numbers, each n bits long, in time $O(\log n)$; similar circuits had previously been described by Offman [55] and Krapchenko [39].

2.3 Ranking

We have seen that prefix computation is useful in a great variety of circumstances. But often we have sequences of values represented as linked lists, and the techniques above can not handle this. Here we describe techniques for prefix computation on linked lists. In particular we study the problem of ranking, which is a special case 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.

A simple algorithm for ranking, due to Wyllie [75], 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 like the array prefix computation algorithm, except that we don't distinguish between even and odd members of the linked list. The algorithm takes time $O(\log n)$, but it requires $O(n)$ processors, so it is less efficient than array prefix computation. Wyllie's algorithm was improved by Vishkin [71], who gave a randomized ranking algorithm for the EREW. This algorithm was later made deterministic [12, 13], made more efficient but at a cost in total time [40], and still later made faster with the same asymptotic number of operations [14, 6].

2.4 Randomized Ranking and Deterministic Coin Tossing

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 can perform ranking by turning 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 tree has logarithmic expected height. Further, 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. Therefore the algorithm as a whole takes time $O(\log^2 n)$, which is more than we would like.

A simple improvement is to only construct tree levels until the size of the remaining linked list is $\Theta(n/\log n)$. 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. This algorithm above was given by Vishkin [71], who also gave a more complicated probabilistic algorithm taking time $O(\log n \log^* n)$.

If addresses take most $O(\log n)$ bits, we can achieve the same bounds as above deterministically. Instead of flipping coins we find a maximal independent set in the linked list. The algorithm below, and its application to ranking, was discovered by Cole and Vishkin [12, 13], who called it deterministic coin tossing. A sorting algorithm similar to the one in step (3) was described by Reif [59].

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

(2) For each element i let s_i be the position of the first bit at which the number i differs from the address of its predecessor, together with the value of that bit. This colors the list with $O(\log n)$ colors, such that adjacent elements have different colors.

(3) Sort the elements of the linked list by their color classes, as follows.

   (a) Arbitrarily divide the list elements into groups of size $\log n$. For each group g, and each s, calculate the number $c_{g,s}$ of elements with color equal to s. Also count for each i the number p_i of elements before it in the same group with the same color.

   (b) For each color s compute the prefix sum of the numbers $c_{g,s}$, into $d_{g,s}$, the number of elements up through group g having that color. Let d_s be the total number of elements in all groups having color s.

   (c) 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.

   (d) For each element i in group g, 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. 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.

All steps can be implemented in time $O(\log n)$ using $O(n/\log n)$ EREW processors. By using this routine in place of the coin tossing, we can rank a linked list in $O(\log n \log\log n)$ time with linear deterministic EREW operations. By repeatedly reducing the number of colors similarly to step (2), we could instead find the independent set S in time $O(\log^* n)$, using a linear number of processors. A more complicated algorithm uses this observation to reduce the total time for ranking to $O(\log n \log^* n)$ without sacrificing optimality.

The list ranking algorithm above is likely to be sufficient for all practical purposes. But there remains an irritating extra factor of $O(\log\log n)$ or $O(\log^* n)$ in the time. Cole and Vishkin [14] removed this factor: their list ranking algorithm uses logarithmic time and linear EREW operations. It operates by removing a sequence of elements from the list, reducing the problem to size $O(n/\log n)$. Then it performs Wyllie's algorithm on the reduced list; finally it undoes the removals to obtain the final ranking. Anderson and Miller [5, 6] have recently found randomized and deterministic algorithms for list ranking that are simpler, and therefore more practical, than that of Cole and Vishkin.

3. EULER TOURS, EAR DECOMPOSITION, AND RELATED TECHNIQUES

The prefix computation techniques above have wide application to problems in which the data is arranged in some linear order. Now we describe methods for handling the case that the data to be processed 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. We also describe algorithms for partitioning graph edges into a sequence of subgraphs, each of which has a simple and easily ordered structure.

Parallel algorithms for problems with tree-like structure can be quite similar to sequential tree traversals, such as pre-order, post-order, or in-order. Thus we would like to generate a list of vertices in these orders; this can be done using the Euler tour technique below. Many tree problems can be solved directly using Euler tours, without generating a traversal order.

Another technique has in many cases been supplanted by Euler tours. Many tree functions can be computed by processing each leaf, removing the leaves and repeating until the tree becomes empty. However, this takes time proportional to the depth of the tree, which may be too large. In an improvement to this method, alternately called centroid decomposition or tree contraction [64, 47, 52], 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 [15] and Gazit et al. [25] give centroid decomposition algorithms that take logarithmic time with a linear number of EREW operations, matching the bounds for Euler tours. We do not describe these algorithms; however we make use of them later to construct ear decompositions.

For sequential algorithms on general graphs, one would typically traverse the edges of the graph using a depth first search [65]. The best known parallel algorithm for constructing such a tree [1] requires randomization, and its bounds are far from optimal. As replacements for depth first search, we describe algorithms for finding Euler tours, pseudo-forest decompositions, and ear decompositions.

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. We describe algorithms for these problems later, in the section on matrix methods.

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. The vertices of the tree typically appear many times, so it is not a simple cycle. Euler tours for binary trees were first used by Wyllie [75]. The Euler tour technique for general trees was introduced by Tarjan and Vishkin [66], 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.

To compute the tour 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 $(p(v), p(p(v)))$. All such links may be computed in constant time with $O(n)$ EREW processors. Typically we 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 [66]. 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: Therefore we describe here the construction of a post-order traversal; the pre-order construction is symmetrical.

We perform a prefix computation on the edges of the tour; the data at each edge is 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.

After we apply this operation to a sequence of edges, the resulting bit tells whether all the edges in the sequence run down. The edge name is that of the edge appearing before the sequence of downward edges containing the last edge, or the last edge itself if that edge runs up.

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) is (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 [66]. Given an Euler tour of a tree, we can calculate the number of vertices in each subtree, 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 difference between the sums at edges $(p(v), v)$ and $(v, p(v))$ gives the size of the subtree rooted at v. •

We may use prefix computation on Euler tours to compute many other tree functions. Tarjan and Vishkin [66] used Euler tours to find the depth of each vertex, along with two more functions which they used to compute the biconnected components of a graph. Tsin [67], and independently Vishkin [72], used Euler tours to compute least common ancestors of pairs of vertices, which they both then usedto compute strong orientations of undirected graphs. Schieber and Vishkin [61] improved the least common ancestor algorithm, again using Euler tours.

3.2 Euler Tours for Graphs

An Euler tour of an undirected or directed graph is again a cycle such that each edge appears exactly once. If the graph is directed we further require that the cycle be directed. 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. 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.

Awerbuch et al. [8] 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 [7] 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.

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 trade successors.

We define the graph CG of an Euler partition as follows. Vertices of CG correspond to 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. CG may have self loops as well as multiple adjacencies, but this is not a problem.

Theorem 8 [8]. CG is connected.

Theorem 9 [8]. 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 is a single cycle.

Thus 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 [14]. 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 of this connected components algorithm, the numbers of vertices and edges are proportional to the number of edges in the original graph.

Finally, step (4) can be executed as follows. Let $e_i, e_{i+1}, \dots e_j$ be a maximal sublist of the edges coming in to vertex v, such that $(e_i, e_{i+1}), (e_{i+1}, e_{i+2}), \dots (e_{j-1}, e_j)$ 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 by a prefix computation on the lists of edges incoming to each vertex v.

Theorem 10 [8]. 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. 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.

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 correspond to one tree of the forest, and each tree 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. Together with the directed Euler tour algorithm this gives us an Euler tour of the original undirected graph.

Not every graph has an Euler tour, so we cannot use this algorithm directly on an arbitrary graph. Further, if as for trees we split every edge in two, one possible tour could be constructed from a tour of a spanning tree of the graph, so any resulting tour would be no more useful than a spanning tree. A better way to make a graph Eulerian is to create a new dummy vertex, and add an edge between that vertex and each original vertex having an odd degree. Israeli and Shiloach [33] use this construction as part of an algorithm to find a maximal matching for an arbitrary graph.

3.3 Pseudo-Forest Decomposition

A *pseudo-forest* is 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. We now describe extensions of the deterministic coin tossing technique of Cole and Vishkin [12, 13] to pseudo-forests, found by Goldberg et al. [26]. This leads to efficient algorithms for graphs of bounded degree, by partitioning the edges of the graph into a number of pseudo-forests.

By a k-coloring of a graph or pseudo-forest below, we mean an assignment of k colors to the vertices such that no two adjacent vertices have the same color. Let n be the number of vertices of a given pseudo-forest or graph, and m be to the number of edges of the graph. The number of edges in a pseudo-forest is at most n.

Lemma [26]. Given a k-coloring of a pseudo-forest, we can compute a $(2\lceil\log k\rceil)$-coloring 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 11 [26]. A 3-coloring of a given pseudo-forest can be found in time $O(\log^* n)$ with $O(n)$ CREW processors. If the maximum degree of any vertex in the pseudo-forest is bounded by Δ, then the algorithm can be performed on $O(n/\log\Delta)$ EREWs using time $O(\log\Delta\log^* n)$.

Proof: Start with the n-coloring given by the names of the vertices, and repeat the reduction of the lemma above. After $O(\log^* n)$ iterations, only 6 colors remain. We now remove one color 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 have at most two different colors. Then the vertices of the color we wish to eliminate choose new colors differing from their two neighboring colors. •

Theorem 12 [26]. Given a graph with maximum vertex degree Δ, with $\Delta = O(\log n)$, we can calculate a $\Delta + 1$-coloring of the graph, and a maximal independent set, 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. 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.

To combine colorings, each vertex first takes as its color the pair of colors from the two colorings we are combining; this gives a $3(\Delta + 1)$-coloring. We now process each vertex, grouped according to color. Each vertex looks at its neighbors and chooses the least color not already taken. Because we process only vertices from a single color class at a time, every two adjacent vertices are processed at different times, and therefore pick different colors. Further, the vertices with the lowest numbered color form a maximal independent set. •

Goldberg et al. [26] give similar algorithms 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 a graph. 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 [44], indicate the importance of ear decompositions to questions of graph connectivity. When we say an ear decomposition starts from a vertex v or edge e, we mean that the first ear of the decomposition is v or e respectively. An open ear decomposition is one in which all ears are open.

Theorem 13 [74]. An undirected graph is bridgeless (2-edge-connected) if and only if it has an ear decomposition starting from a single vertex.

Theorem 14 [74]. An undirected graph is biconnected (2-vertex-connected) if and only if it has an open ear decomposition starting from a single edge.

Theorem 15 (Folklore). A directed graph is strongly connected if and only if it has an ear decomposition starting from a single vertex.

Lovász [44] gave a parallel algorithm for finding the ear decompositions described in these theorems. Maon et al. [46] gave more efficient algorithms for both undirected cases, using Euler tours. Miller and Ramachandran [50] independently discovered similarly efficient algorithms for ear decomposition and open ear decomposition, instead using centroid decomposition.

We describe here the algorithm of Maon et al. for ear decomposition of bridgeless graphs. It is similar to those of Tsin [67] and Vishkin [72] for strongly directing bridgeless graphs, mentioned as applications of Euler tours. In fact, if one applies the cycle directing algorithm, described for undirected graph Euler tours, to the ear decomposition of a bridgeless graph, the result is a strongly connected directed graph.

(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, and the distance $level(u, v)$ from a to the root.

(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 edge identifiers.

(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.

Least common ancestors can be computed using an algorithm of Schieber and Vishkin [61], which uses prefix computation on Euler tours of the tree. The remaining steps can be performed by further Euler tour computations together with centroid decomposition. Thus we have

Theorem 16 [46]. 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.

Maon et al. [46] also give an algorithm for finding an open ear decomposition of a biconnected graph. This algorithm is like the one above, with the identifiers used to break ties chosen so that all of the resulting ears are open.

A property of biconnected graphs closely related to ear decomposition is *st*-numbering. This is an ordering of the vertices such that there is an edge between the first and last vertices s and t, and such that each other vertex has neighbors both before and after it in the ordering. An *st*-numbering can be computed from an open ear decomposition [18]; Maon et al. [46] showed how to do this in parallel. Klein and Reif [38] used *st*-numbering as part of a parallel algorithm for embedding planar graphs.

Ear decompositions have also been used as part of other graph algorithms, in particular for the connectivity of a graph. The algorithms we described above determine whether a graph is bridgeless or biconnected, although in the latter case a simpler algorithm was known [66]. Miller and Ramachandran [51] used ear decomposition as part of an algorithm for testing graph 3-vertex-connectivity; this algorithm has recently been improved by Ramachandran and Vishkin [56]. Kanevsky and Ramachandran [34] again used ear decomposition to test 4-vertex-connectivity.

Eppstein [17] has recently found another application of open ear decomposition, to the recognition and decomposition of series parallel graphs. His algorithm runs in time $O(\log n)$ with linear CRCW processors, improving a previous algorithm of He and Yesha [29] that took $O(\log^2 n)$ time with linear EREW processors.

4. MATRIX METHODS

One usually thinks of matrix calculations in the context of scientific computation, in which the matrix corresponds to an approximation of some physical system. Certainly this is a common use of matrices, and as such matrix systems 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. Linial et al. [43] characterize vertex connectivity using a generalization of *st*-numbering; they give randomized parallel algorithms for computing the connectivity of a graph using various matrix operations. Another example is the computation of shortest paths in a network (graph with edge weights). We 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. In other problems the values of the matrix cells need not even be numeric; in the next section we describe matrix computations over a wide class of algebraic structures.

4.1 Closed Semiring Systems

A *semiring* is a triple $S = (X, +, \times)$ where $+$ (addition) and $\times$ (multiplication) are associative binary operations on the set X, with multiplication distributing over addition. Addition must be commutative, but we do not require the same of multiplication. Let S be a semiring with the following further properties. First, addition should be idempotent: for any a, $a + a = a$. Second, any countably infinite sum $a_1 + a_2 + \cdots + a_i + \cdots$ must have a unique solution; the commutative, associative, and distributive laws should apply to infinite as well as finite sums. We call such a system a *closed semiring* [2].

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 of all the labels of the edges of the path. Given such a labeled graph, we want 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.

Theorem 17 [41]. Assume that the given semiring has the property that, for all a, $1 + a = 1$. Let A be an $n \times n$ matrix. For each edge (i, j) in the graph, labeled with some value v, let $A[i, j] = v$. For each i, j with no edge between them, we let $A[i, j] = 0$. For each i, $A[i, i] = 1$. Let $B = A^m$ for any $m \geq n$. Then B gives the semiring closure of the graph.

Corollary 1. Let the given semiring satisfy the hypothesis of theorem 17, 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 time $T(n) \log n$, using $M(n)$ processors.

This algorithm is almost optimal. It is not clear whether it can be extended to more general closed semiring systems.

Now let us describe some applications of closed semirings. 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 takes 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.

A second example is the transitive closure of a directed graph. We take as our semiring Boolean algebra: the addition operator is logical or and the multiplication operator logical and. 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. A version of this algorithm for the EREW model was given by Hirschberg [31].

Closed semirings have a less obvious application to finding topological orderings of directed acyclic graphs. This can easily be performed in linear time sequentially [2], but the algorithm does not lend itself to parallelism. A topological ordering can, however, be found by sorting the vertices by their in-degree in the transitive closure of the graph [41]. This takes $O(\log n)$ time with $O(n^3)$ weak CRCWs.

4.2 Matching

Another important problem that can be solved using matrix techniques is 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 related problems, in each of which the task is to construct a matching with certain properties:

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

(2) Maximum matching. This is a matching with the largest possible number of edges, generalizing case (1) to graphs without perfect matchings.

(3) Maximal matching. Each edge must have at least one endpoint covered by the matching, so no more edges can be added to the matching. A maximum matching must be maximal, but the converse need not be true.

(4) 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.

Matching is interesting in its own right, but it is also closely related to other problems. A maximum flow on a bipartite graph can be used to find a maximum matching; conversely matchings can be used to find certain types of flows. Aggarwal and Anderson [1] use flows calculated by matchings to construct depth first search trees in parallel.

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 [22, 53]; here w is the largest weight on any edge.

Finding a maximal matching seems to be much easier. Israeli and Shiloach [33] give an algorithm which takes $O(\log^3 m)$ time with $O(m + n)$ CRCW processors. Israeli and Itai [32] give a randomized algorithm using the same number of processors, but taking time $O(\log m)$.

Here we describe a parallel perfect matching algorithm due to Mulmuley et al. [53]. The problem is reduced to the problem of finding a minimum weight perfect matching, with the further assumption that there is exactly one such matching.

Theorem 18 [53]. If each edge of a graph is given a random weight from 1 to $2m$, there is a unique minimum weighted matching with probability at least 1/2. Therefore, if we can find a unique minimum weight perfect matching in parallel, we can find a perfect matching in the same expected time.

We now 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, where k is the maximum weight on any edge. Let $B_{i,j}$ be the matrix formed by removing row i and column j from matrix B. The adjoint $\mathrm{adj}(B)$ is defined by $\mathrm{adj}(B)[j, i] = \det(B_{i,j})$. Also, let w be the weight of the minimum weight perfect matching M of G.

Theorem 19 [53]. 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.

Theorem 20 [53]. 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.

Galil and Pan [23] give a randomized algorithm for inverting $n \times n$ matrices of k-bit rational numbers in time $O(\log^2 n)$ using $O(knM(n))$ processors. Their algorithm also computes the adjoint. Here $M(n)$ denotes the current best known sequential time for multiplying two matrices, $O(n^{2.376})$ [16]. Thus we achieve the following:

Theorem 21 [53]. 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.

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.

Galil and Pan [23] give a different perfect matching algorithm, based on that of Karp et al. [36], 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 [49].

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. These algorithms seem likely to be useful not only in their own right, but also as examples of ways to break up other problems into parts suitable for parallel solution.

ACKNOWLEDGEMENTS

We would like to thank Rich Anderson, Lane Hemachandra, John Kender, Victor Pan, Baruch Schieber, Marc Snir, and Moti Yung, who read earlier versions of this paper and supplied many helpful comments. This research was supported in part by NSF grants DCR-85-11713, CCR-86-05353, and CCR-88-14977, and by DARPA contract N00039-84-C-0165.

REFERENCES

[1] A. Aggarwal and R.J. Anderson, A Random NC Algorithm for Depth First Search. *19th Symp. Theory of Computing*, 1987, 325–334.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.

[3] A.V. Aho and J.D. Ullman, Principles of Compiler Design, Addison Wesley, 1977.

[4] M. Ajtai, J. Komlós, and E. Szemerédi, An $O(n \log n)$ Sorting Network. *15th Symp. Theory of Computing*, 1983, 1–9.

[5] R.J. Anderson and G.L. Miller, A Simple Randomized Parallel Algorithm for List-Ranking. *Inf. Proc. Lett.*, to appear.

[6] R.J. Anderson and G.L. Miller, Deterministic Parallel List Ranking. *3rd Aegean Worksh. Comput.*, Springer-Verlag LNCS 319, 1988.

[7] M.J. Atallah and U. Vishkin, Finding Euler Tours in Parallel. *J. Comput. Sys. Sci. 29*, 1984, 330–337.

[8] B. Awerbuch, A. Israeli, and Y. Shiloach, Finding Euler Circuits in Logarithmic Parallel Time. *16th Symp. Theory of Computing*, 1984, 249–257.

[9] R.P. Brent, The Parallel Evaluation of General Arithmetic Expressions. *J. ACM 21(2)*, 1974, 201–206.

[10] B.S. Chlebus, K. Diks, T. Hagerup, and T. Radzik, Efficient Simulations between Concurrent-Read Concurrent-Write PRAM Models. *13th Symp. Math. Found. Comput. Sci.*, Springer-Verlag LNCS 324, 1988, 231–239.

[11] B.S. Chlebus, K. Diks, T. Hagerup, and T. Radzik, New Simulations between CRCW PRAMs. Manuscript.

[12] R. Cole and U. Vishkin, Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms. *18th Symp. Theory of Computing*, 1986, 206–219.

[13] R. Cole and U. Vishkin, Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Inf. and Control 70*, 1986, 32–53.

[14] R. Cole and U. Vishkin, Approximate and Exact Parallel Scheduling with Applications to List, Tree and Graph Problems. *27th Symp. Found. Comput. Sci.*, 1986, 478–491.

[15] R. Cole and U. Vishkin, Optimal Parallel Algorithms for Expression Tree Evaluation and List Ranking. *3rd Aegean Worksh. Comput.*, Springer-Verlag LNCS 319, 1988.

[16] D. Coppersmith and S. Winograd, Matrix Multiplication via Arithmetic Progressions. *19th Symp. Theory of Computing*, 1987, 1–6.

[17] D. Eppstein, Parallel Recognition of Series-Parallel Graphs. Submitted.

[18] S. Even and R.E. Tarjan, Computing an *st*-numbering. *Theor. Comput. Sci. 2*, 1976, 339–344.

[19] F.E. Fich, R.L. Ragde, and A. Wigderson, Relations Between Concurrent-Write Models of Parallel Computation. *SIAM J. Comput. 17*, 1988, 606–627.

[20] F.E. Fich, R.L. Ragde, and A. Wigderson, Simulations Among Concurrent-Write PRAMs. *Algorithmica 3*, 1988, 43–51.

[21] Z. Galil, Optimal Parallel Algorithms for String Matching. *16th Symp. Theory of Computing*, 1984, 240–248, and *Inf. and Control 67*, 1986, 144–157.

[22] Z. Galil, Sequential and Parallel Algorithms for Finding Maximum Matchings in Graphs. *Ann. Rev. Comput. Sci. 1*, 1986, 197–224.

[23] Z. Galil and V. Pan, Improved Processor Bounds for Algebraic and Combinatorial Problems in RNC. *26th Symp. Found. Comput. Sci.*, 1985, 490–495.

[24] H. Gazit, An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph. *27th Symp. Found. Comput. Sci.*, 1986, 492–501.

[25] H. Gazit, G.L. Miller, and S.H. Teng, Optimal Tree Contraction in EREW Model. Manuscript.

[26] A. Goldberg, S. Plotkin, and G. Shannon, Parallel Symmetry-Breaking in Sparse Graphs. *19th Symp. Theory of Computing*, 1987, 315–324.

[27] M. Goldberg and T. Spencer, A New Parallel Algorithm for the Maximal Independent Set Problem. *28th Symp. Found. Comput. Sci.*, 1987, 161–165.

[28] V. Grolmusz and P. Ragde, Incomparibility in Parallel Computation. *28th Symp. Found. Comput. Sci.*, 1987, 89–98.

[29] X. He and Y. Yesha, Parallel Recognition and Decomposition of Two Terminal Series Parallel Graphs, *Inf. and Comput. 75*, 1987, 15–38.

[30] W.D. Hillis and G.L. Steele, Data Parallel Algorithms. *C. ACM 29(12)*, 1986, 1170–1183.

[31] D.S. Hirschberg, Parallel Algorithms for the Transitive Closure and the Connected Component Problems. *8th Symp. Theory of Computing*, 1976, 55–57.

[32] A. Israeli and A. Itai, A Fast and Simple Randomized Parallel Algorithm for Maximal Matching. *Inf. Proc. Lett. 22*, 1986, 77–80.

[33] A. Israeli and Y. Shiloach, An Improved Parallel Algorithm for Maximal Matching. *Inf. Proc. Lett. 22*, 1986, 57–60.

[34] A. Kanevsky and V. Ramachandran, Improved Algorithms for Graph Four-Connectivity. *28th Symp. Found. Comput. Sci.*, 1987, 252–259.

[35] A.R. Karlin and E. Upfal, Parallel Hashing – An Efficient Implementation of Shared Memory. *18th Symp. Theory of Computing*, 1986, 160–168.

[36] R.M. Karp, E. Upfal, and A. Wigderson, Constructing a Perfect Matching is in Random NC. *17th Symp. Theory of Computing*, 1985, 22–32.

[37] R.M. Karp, E. Upfal, and A. Wigderson, Are Search and Decision Problems Computationally Equivalent? *17th Symp. Theory of Computing*, 1985, 464–483.

[38] P.N. Klein and J.H. Reif, An Efficient Parallel Algorithm for Planarity. *27th Symp. Found. Comput. Sci.*, 1986, 465–477.

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

[40] C.P. Kruskal, L. Rudolph, and M. Snir, The Power of Parallel Prefix. *IEEE Trans. Comput. C-34*, 1985, 965–968.

[41] L. Kučera, Parallel Computation and Conflicts in Memory Access. *Inf. Proc. Lett. 14(2)*, 1982, 93–96.

[42] R.E. Ladner and M.J. Fischer, Parallel Prefix Computation. *J. ACM 27(4)*, 1980, 831–838.

[43] N. Linial, L. Lovász, and A. Wigderson, A Geometric Interpretation of Graph Connectivity and its Algorithmic Applications. *27th Symp. Found. Comput. Sci.*, 1986, 39–48.

[44] L. Lovász, Computing Ears and Branchings in Parallel. *26th Symp. Found. Comput. Sci.*, 1985, 464–467.

[45] M. Luby, A Simple Parallel Algorithm for the Maximal Independent Set Problem. *17th Symp. Theory of Computing*, 1985, 1–10.

[46] Y. Maon, B. Schieber, and U. Vishkin, Parallel Ear Decomposition Search (EDS) and ST-Numbering in Graphs. *VLSI Alg. and Arch.*, Springer-Verlag LNCS 227, 1986, 34–45.

[47] N. Megiddo, Applying Parallel Computation Algorithms in the Design of Serial Algorithms. *J. ACM 30(4)*, 1983, 852–865.

[48] K. Mehlhorn and U. Vishkin, Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica 21*, 1984, 339–374.

[49] S. Micali and V.V. Vazirani, An $O(\sqrt{|V|}|E|)$ Algorithm for Finding Maximum Matchings in General Graphs. *21st Symp. Found. Comput. Sci.*, 1980, 17–27.

[50] G.L. Miller and V. Ramachandran, Efficient Parallel Ear Decomposition with Applications. Manuscript.

[51] G.L. Miller and V. Ramachandran, A New Graph Triconnectivity Algorithm and Its Parallelization. *19th Symp. Theory of Computing*, 1987, 335–344.

[52] G.L. Miller and J.H. Reif, Parallel Tree Contraction and its Application. *26th Symp. Found. Comput. Sci.*, 1985, 478–489.

[53] K. Mulmuley, U.V. Vazirani, and V.V. Vazirani, Matching is as Easy as Matrix Inversion. *19th Symp. Theory of Computing*, 1987, 345–354, and *Combinatorica 7*, 1, 1987, 105–114.

[54] D. Nassimi and S. Sahni, Data Broadcasting SIMD Computers. *IEEE Trans. Comput. C-30*, 1981, 101–107.

[55] Y.P. Offman, On the Algorithmic Complexity of Discrete Functions. *Dokl. Sov. Acad. Sci. 145(1)*, 1962, 48–51. English translation in *Sov. Phys. Dokl. 7(7)*, 1963, 589–591.

[56] V. Ramachandran and U. Vishkin, Efficient Parallel Triconnectivity in Logarithmic Time. *3rd Aegean Worksh. Comput.*, Springer-Verlag LNCS 319, 1988.

[57] A.G. Ranade, How to Emulate Shared Memory. *28th Symp. Found. Comput. Sci.*, 1987, 185–194.

[58] J.H. Reif, Probabilistic Parallel Prefix Computation. Manuscript.

[59] J.H. Reif, An Optimal Parallel Algorithm for Integer Sorting. *26th Symp. Found. Comput. Sci.*, 1985, 496–504.

[60] B. Schieber, Design and Analysis of some Parallel Algorithms. Ph.D. Dissertation, 1987, Tel Aviv University.

[61] B. Schieber and U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization. Ultracomputer Note 118, 1987, Courant Institute, New York University, New York.

[62] J.T. Schwartz, Ultracomputers. *ACM Trans. Prog. Lang. and Sys. 2*, 1980, 484–521.

[63] Y. Shiloach and U. Vishkin, Finding the Maximum, Merging and Sorting in a Parallel Computation Model. *J. Algorithms 2*, 1981, 88–102.

[64] M. Snir and A.B. Barak, A Direct Approach to the Parallel Evaluation of Rational Expressions with a Small Number of Processors. *IEEE Trans. Comput. C-26*, 1977, 933-937.

[65] R.E. Tarjan, Depth-first Search and Linear Graph Algorithms. *SIAM J. Comput. 15(3)*, 1972, 814–830.

[66] R.E. Tarjan and U. Vishkin, Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time. *25th Symp. Found. Comput. Sci.*, 1984, 12–20

[67] Y.H. Tsin, An Optimal Parallel Processor Bound Strong Orientation of an Undirected Graph. *Inf. Proc. Lett. 20*, 1985, 143–146.

[68] L.G. Valiant, Parallelism in Comparison Problems. *SIAM J. Comput. 4(3)*, 1975, 348–355.

[69] U. Vishkin, Implementation of Simulations Memory Access Models that Forbid it. *J. Algorithms 4*, 1983, 45–50.

[70] U. Vishkin, A Parallel-Design Distributed-Implementation (PDDI) General-Purpose Computer. *Theor. Comput. Sci. 32*, 1984, 157–172.

[71] U. Vishkin, Randomized Speed-ups in Parallel Computation. *16th Symp. Theory of Computing*, 1984, 230–239.

[72] U. Vishkin, On Efficient Parallel Strong Orientation. *Inf. Proc. Lett. 20*, 1985, 235–240.

[73] J. von zur Gathen, Parallel Algebraic Computations. Unpublished course notes, 1984.

[74] H. Whitney, Non-Separable and Planar Graphs. *Trans. AMS 34*, 1932, 339–362.

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