CHAPTER 17

Parallel Algorithms for Shared-Memory Machines

Richard M. KARP

Department of Computer Science, University of California at Berkeley, Berkeley, CA 94720, USA

Vijaya RAMACHANDRAN

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA

Contents

HANDBOOK OF THEORETICAL COMPUTER SCIENCE
Edited by J. van Leeuwen

1. Introduction

Parallel computation is rapidly becoming a dominant theme in all areas of computer science and its applications. It is likely that, within a decade, virtually all developments in computer architecture, systems programming, computer applications and the design of algorithms will be taking place within the context of parallel computation.

In preparation for this revolution, theoretical computer scientists have begun to develop a body of theory centered around parallel algorithms and parallel architectures. Since there is no consensus yet on the appropriate logical organization of a massively parallel computer, and since the speed of parallel algorithms is constrained as much by limits on interprocessor communication as it is by purely computational issues, it is not surprising that a variety of abstract models of parallel computation have been pursued.

Closest to the hardware level are the VLSI models, which focus on the technological limits of today's chips, in which gates and wires are packed into a small number of planar layers. At the next level of abstraction are those models in which a parallel computer is viewed as a set of processors interconnected in a fixed pattern, with each processor having a small number of neighbors.

At one further remove from physical reality is the parallel random-access machine (PRAM), in which it is assumed that, in addition to the private memories of the processors, there is a shared memory, and that any processor can access any cell of that memory in unit time. The PRAM cannot be considered a physically realizable model, since, as the number of processors and the size of the global memory scales up, it quickly becomes impossible to provide a constant-length data path from any processor to any memory cell. Nevertheless, the PRAM has proved to be an extremely useful vehicle for studying the logical structure of parallel computation in a context divorced from issues of parallel communication, and it is the focus of attention in the present survey. Algorithms developed for other, more realistic, models are often based on algorithms originally designed for the PRAM.

Many studies of algorithms and complexity in the PRAM model focus on the trade-off between the time for a parallel computation and the number of processors required. In a practical situation the number of processors available is fixed, but our theoretical studies are enriched if we let the number of processors grow as a function of the size of the problem instance being solved. Of particular interest are so-called *efficient* algorithms, which run in *polylog time* (i.e., the parallel computation time is bounded by a fixed power of the logarithm of the size of the input), and in which the processor–time product exceeds the number of steps in an optimal sequential algorithm by at most a polylog factor. Section 2 surveys efficient PRAM algorithms for bookkeeping operations such as compacting an array by squeezing out its "dead" elements, for evaluating algebraic expressions, for searching a graph and decomposing it into various kinds of components, and for sorting, merging and selection. These efficient parallel algorithms are typically completely different from the best sequential algorithms for the same problems, and their discovery has required the creation of a new set of paradigms for the construction of parallel algorithms.

In the PRAM model there is the possibility of read- and write-conflicts, in which two or more processors try to read from or write into the same memory cell concurrently. Distinctions in the way these conflicts are handled lead to several different variants of the model. The weakest of these is the exclusive-read exclusive-write (EREW) PRAM, in which concurrent reading or writing are forbidden; of intermediate strength is the concurrent-read exclusive-write (CREW) PRAM, which allows concurrent reading but not concurrent writing; and strongest of all is the concurrent-read concurrent-write (CRCW) PRAM, which permits both kinds of concurrency. Several varieties of CRCW PRAMs have been defined; they differ in their method of resolving write conflicts. In Section 3 it is shown that, although these variants do not differ very greatly in computation speed, the CREW PRAM is strictly more powerful than the EREW PRAM and strictly less powerful than the CRCW PRAM. It is also shown that certain simple tasks, such as multiplying two n-bit numbers, inherently require time $\Omega(\log n/\log\log n)$ even on the strongest PRAM model provided the number of processors is polynomial-bounded in the size of the input.

Section 3 goes on to study the relationship between the PRAM and other abstract models of parallel computation, such as Boolean circuits, alternating Turing machines and vector machines. It turns out that all these models are equivalent in their ability to solve problems in polylog time using a polynomial-bounded number of computing elements (processors or gates). This motivates the definition of NC as the class of problems that can be solved within these resource bounds by deterministic algorithms. Two important refinements of this result are presented, each showing that certain parallel computation models are equivalent in their ability to solve problems in time $O(\log^k n)$, where k is a fixed positive integer, using a polynomial-bounded amount of hardware. For PRAMs the amount of hardware is measured by the number of processors, for Boolean circuits by the number of wires and for alternating Turing machines by the number of possible configurations, which is exponential in the space. The first refinement, by Ruzzo [203], states that alternating Turing machines are equivalent to bounded fan-in circuits; the second, by Stockmeyer & Vishkin [220], states that CRCW PRAMs are equivalent to unbounded fan-in circuits.

Section 4 gives a survey of important problems that lie in the class NC. Among these are the basic arithmetic operations, transitive closure and Boolean matrix multiplication, the computation of the determinant, the rank or the inverse of a matrix, the evaluation of certain classes of straight-line programs, and the construction of a maximal independent set of vertices in a graph. Section 4 also presents randomized algorithms that operate in polylog time using a polynomial-bounded number of processors for problems such as finding a maximum matching in a graph.

The study of parallel complexity within the PRAM model has led to some important negative results. Using a theory of reducibility analogous to the theory of NP-completeness, it has been possible to identify certain problems as P-complete; such problems are solvable sequentially in polynomial time, but do not lie in the class NC unless every problem solvable in sequential polynomial time lies in NC. This is evidence that the P-complete problems are inherently resistent to ultrafast parallel solution. Our survey concludes, in Section 4, by exploring this concept and deriving a number of examples of P-complete problems, including the maximum-flow problem and the problem of evaluating the output of a monotone Boolean circuit when all the inputs are fixed at constant values.

2. Efficient PRAM algorithms

2.1. The PRAM model and the concepts of efficient and optimal algorithms

The primary model of parallel computation that we will be working with is the PRAM (*P*arallel *R*andom-*A*ccess *M*achine) (Fortune & Wyllie [85]; Goldschlager [106]; Savitch & Stimson [206]). This is an idealized model, and can be viewed as the parallel analogue of the sequential RAM (Cook & Reckhow [64]). A PRAM consists of several independent sequential processors, each with its own private memory, communicating with one another through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM operation, and write into one gobal or local memory location.

PRAMs can be classified according to restrictions on global memory access. An Exclusive-Read Exclusive-Write (or EREW) PRAM is a PRAM for which simultaneous access to any memory location by different processors is forbidden for both reading and writing. In a Concurrent-Read Exclusive-Write (or CREW) PRAM simultaneous reads are allowed but no simultaneous writes. A Concurrent-Read Concurrent-Write (or CRCW) PRAM allows simultaneous reads and writes. In this case we have to specify how to resolve write conflicts. Some commonly used methods of resolving write conflicts are

(a) all processors writing into the same location must write the same value (the COMMON model);

(b) any one processor participating in a common write may succeed, and the algorithm should work correctly regardless of which one succeeds (the ARBITRARY model);

(c) there is a linear ordering on the processors, and the minimum numbered processor writes its value in a concurrent write (PRIORITY model).

Even though there is a variety of PRAM models, they do not differ very widely in their computational power. We show in Section 3 that any algorithm for a CRCW PRAM in the PRIORITY model can be simulated by an EREW PRAM with the same number of processors and with the parallel time increased by only a factor of $O(\log P)$, where P is the number of processors; further, any algorithm for a PRIORITY PRAM can be simulated by a COMMON PRAM with no loss in parallel time provided sufficiently many processors are available.

Define *polylog*$(n) = \bigcup_{k>0} O(\log^k n)$. Let S be a problem whose fastest sequential algorithm runs in time proportional to $T(n)$ (although, by Blum's Speed-up Theorem (cf. Seiferas [210]), there are pathological problems for which no fastest sequential algorithm exists, this phenomenon will not trouble us in practice). Many problems have the property that any algorithm that solves them would have to look at all of the input in the worst case, and hence $T(n) = \Omega(n)$ in such cases. A PRAM algorithm A for S, running in parallel time $t(n)$ with $p(n)$ processors is *optimal* if

(a) $t(n) = polylog(n)$; and

(b) the *work* $w(n) = p(n) \cdot t(n)$ is $O(T(n))$.

An optimal parallel algorithm achieves a high degree of parallelism in an optimal way. Analogously, we can define an *efficient* parallel algorithm for problem S as one for which the work $w(n)$ is $T(n) \cdot polylog(n)$ with the parallel time $t(n) = polylog(n)$; i.e., an efficient parallel algorithm is one that achieves a high degree of parallelism and comes within a polylog factor of optimal speed-up. A major goal in the design of parallel algorithms is to find optimal and efficient algorithms with $t(n)$ as small as possible. Clearly it is easier to design optimal algorithms on a CRCW PRAM than on a CREW or EREW PRAM. However, the simulations between the various PRAM models make the notion of an *efficient* algorithm invariant with respect to the particular PRAM model used. Thus this latter notion is more robust.

Consider a computation that can be done in t parallel steps with x_i primitive operations at step i. If we implement this computation directly on a PRAM to run in t parallel steps, the number of processors required would be $m = \max x_i$. If we have $p < m$ processors, we can still simulate this computation effectively by observing that the ith step can be simulated in time $\lceil x_i/p \rceil \leqslant (x_i/p) + 1$, and hence the total parallel time to simulate the computation with p processors is no more than $\lceil x/p \rceil + t$ where $x = \Sigma_i x_i$. This observation, known as *Brent's scheduling principle* (Brent [42]), is often used in the design of efficient parallel algorithms. It should be noted that this simulation assumes that *processor allocation* is not a problem, i.e., that it is possible for each of the p processors to determine, on-line, the steps it needs to stimulate. We will see below that this is sometimes a nontrivial task.

Brent's scheduling principle implies that, when processor allocation is not a problem, a parallel algorithm requiring work $w(n)$ and time $t(n)$ can be simulated using p processors in time $w(n)/p + t(n)$. In practice, p is often small compared to n, and in such cases the ratio $w(n)/p$ is typically much greater than $t(n)$. In view of this fact, Kruskal, Rudolph & Snir [245] suggest that it may be unnecessarily restrictive to concentrate on parallel algorithms with $t(n) = polylog(n)$, and they emphasize instead the concept of polynomial speed-up. For a problem requiring sequential time $T(n)$, a parallel algorithm running in time $t(n)$ is said to have *polynomial speed-up* if, for some $\varepsilon < 1$, $t(n) = O(T(n^{\varepsilon}))$. Algorithms with polynomial speed-up are further classified according to their *innefficiency*. For a problem whose best sequential algorithm runs in time $T(n)$, a parallel algorithm that performs an amount of work $w(n)$ is said to have *constant inefficiency* if $w(n) = O(T(n))$, *polylog inefficiency* if

$$w(n) = O(T(n)\ polylog(T(n)))$$

and *polynomial inefficiency* if $w(n)$ is bounded by a polynomial in $T(n)$. Kruskal, Rudolph & Snir argue that algorithms with polynomial speed-up and constant or polylog inefficiency may be extremely useful, even if they do not run in polylog time.

For PRAM algorithms we would like to have simple algorithms that are easy to specify and code. Most of the algorithms we describe will have this feature.

There are a few key methods that have emerged as fundamental subroutines in the design of efficient and optimal parallel algorithms. In the following subsections, we review these basic techniques and algorithms.

2.2. *Basic PRAM techniques*

2.2.1. *Prefix sums*

The first problem we consider is *prefix sums*. Let $*$ be an associative operation over a domain D. Given an array $[x_1, \ldots, x_n]$ of n elements from D, the prefix sums problem is to compute the n prefix sums $S_i = x_1 * x_2 * \cdots * x_i = \Sigma_{j=1}^{i} x_j$, $i = 1, \ldots, n$. This problem has several applications. For example, consider the problem of compacting a sparse array: given an array of n elements, many of which are zero, we wish to generate a new array containing the non-zero elements in their original order. We can compute the position of each non-zero element in the new array by assigning value 1 to the non-zero elements, and computing prefix sums with $*$ operating as regular addition. An application to the problem of adding two n-bit numbers is given in Section 4.2. Further, recognition of any regular language whose input size is restricted to n can be viewed as a prefix sums problem [150].

There is a simple sequential algorithm to solve the prefix sums problem using $n-1$ operations, by computing S_i incrementally from S_{i-1}, for $i = 2, \ldots, n$. Unfortunately this algorithm has no parallelism in it since one of the two operands for the ith $*$ operation is the result of the $(i-1)$st operation.

We now describe a simple parallel algorithm to compute prefix sums in parallel (Ladner & Fischer [150]). For simplicity we assume that n is a power of 2.

```
PARALLEL PREFIX ALGORITHM
  Input: an array $[x_1, \ldots, x_n]$ of elements from domain $D$.
if $n = 1$ then $S_1 := x_1$ else
  (1) for $i = 1, \ldots, n/2$ compute $y_1 := x_{2i-1} * x_{2i}$
  (2) recursively compute prefix sums $S_i, i = 1, \ldots, n/2$, for the new array
      $[y_1, \ldots, y_{n/2}]$
  (3) for even $i$ in $\{1, 2, \ldots, n\}$ set $S_i := S_{i/2}$
  (4) for odd $i$ in $\{1, 2, \ldots, n\}$ set $S_i := S_{(i-1)/2} * x_i$.
```

This algorithm runs on an EREW PRAM since there are no conflicts in the memory accesses. The parallel time $t(n)$ satisfies the recurrence $t(n) = t(n/2) + O(1)$ with $t(1) = 0$, and the work satisfies the recurrence $w(n) = w(n/2) + O(n)$ with $w(1) = 0$. Thus $t(n) = O(\log n)$ and $w(n) = O(n)$. By invoking Brent's scheduling principle we see that this is an optimal EREW PRAM algorithm for $p(n) = O(n/\log n)$. Processor allocation is straightforward in this case and we illustrate it by a standard technique used to implement Brent's principle: Let the number of processors available be $p \leqslant n/\log n$, and let $q = [n/p]$. We first assign the ith processor to elements $x_{(i-1)\cdot q+1}, x_{(i-1)\cdot q+2}, \ldots, x_{i\cdot q}$, $i = 1, \ldots, p$. The ith processor stores these values in an array in its local memory, combines these q values (sequentially) using $*$ in $O(q)$ time and places the result in y_i. Now the array has only p elements in it, and the parallel prefix algorithm is used to compute these prefix sums in $O(\log n)$ time using p processors. Finally, in additional $O(n/p)$ time, the ith processor computes the prefix sums for its local array with S_{i-1} as the first element of the array. This algorithm runs in $O(n/p)$ time with p processors on an EREW PRAM for $p \leqslant n/\log n$.

On a CRCW PRAM, the above algorithm can be modified to run optimally in $O(\log n/\log\log n)$ time when the x_i are $O(\log n)$-bit numbers and $*$ is ordinary addition (Reif [196]; Cole & Vishkin [56]).

Since the prefix sums problem is an important one, much attention has been given to fine-tuning the constants in the time and processor bounds (see, e.g., [150, 80]).

2.2.2. *List ranking*

A problem closely related to the prefix sums problem is the *list ranking problem*: Given a linked list of n elements, compute the suffix sums of the last i elements of the list, $i=1,\ldots,n$. This is a variant of prefix sums, in which the ordered sequence of elements is given in the form of a linked list rather than an array, and the sums are computed from the end, rather than from the beginning. The term "list ranking" is usually applied to the special case of this problem in which all elements have value 1, and $*$ stands for addition (and hence the result of the list ranking computation is to obtain, for each element, the number of elements ahead of it in the list, i.e., its rank in the list); however, the technique we shall present easily adapts to our generalization.

We assume that the linked list is represented by a contents array $c[1,\ldots,n]$ and a successor array $s[1,\ldots,n]$: here, $c(i)$ gives the value of the element at location i, and $s(i)$ gives the location j of the successor to $c(i)$ on the list. Without loss of generality we assume that our domain has a zero element z, that the last element on the list $c(i_l)$ has value z, and that $s(i_l)=i_l$. The following simple algorithm solves the list ranking problem on an EREW PRAM in $O(\log n)$ time with n processors (see, e.g., [242]).

```
BASIC LIST RANKING ALGORITHM
for $\lceil \log n \rceil$ iterations repeat
  in parallel for $i=1,\ldots,n$ do
    $c(i) := c(i)*c(s(i))$;
    $s(i) := s(s(i))$;
for $i=1,2,\ldots,n$ output $c(i)$ as the rank of element $i$.
```

The operation used in this algorithm of replacing each pointer $s(i)$ by the pointer's pointer $s(s(i))$ is called *pointer jumping*, and is a fundamental technique in parallel algorithm design. Let the *rank* $r(i)$ of the element in location i be the distance of this element from the end of the input linked list. The correctness of this algorithm follows from the observation that at the start of each step, $c(i)$ equals the sum of elements in the input list with ranks $r(i), r(i)-1, \ldots, r(s(i))+1$ for the current $s(i)$; and after $\lceil \log n \rceil$ iterations, $s(i)=i_l$ for all i. By assigning a processor to each location i, we obtain an n-processor $O(\log n)$ time parallel algorithm. Observe, however, that the work done by this algorithm is $\Theta(n \log n)$ and hence this algorithm as it stands does not lead to an optimal parallel algorithm (since there is a simple linear-time sequential algorithm for the list ranking problem).

The list ranking problem is similar to the prefix sums problem, which has a simple optimal parallel algorithm. The optimal parallel prefix algorithm reduces the problem of computing prefix sums on n elements to one of computing prefix sums on the $n/2$ elements at even positions on the array. This reduction is done in constant time and is data-independent in the sense that the locations of the $n/2$ elements in the reduction list are predetermined. If we try to implement a list ranking algorithm with this property, we run into the problem that a given element has no way of knowing whether it is at an odd or even position on the list. Except for the beginning and end elements, there is no way to determine this information from the local environment of an element.

In order to overcome this problem, we note that we need not necessarily locate the elements at even positions. It suffices to construct a set S of no more than $c \cdot n$ elements in the list, with $c<1$, such that the distance in the list between any two consecutive elements of S is small. The list ranking problem can then be solved as follows:

(i) *List contraction:* Create a contracted list composed of the elements of S, in which each element of S has as its successor the first element of S that follows it in the original list, and a value equal to its own value in the original list, plus the sum of the values of the elements that lie between it and this successor;

(ii) Recursively, solve the list ranking problem for the contracted list. The suffix sum for each element in the contracted list is the same as its suffix sum in the original list;

(iii) Extend this solution to all elements of the original list. The time to do this is proportional to the maximum distance between two elements of S in the original list, and the work is proportional to the length of the original list.

We shall present an optimal $O(\log n)$ time randomized list ranking algorithm that takes this approach, but with the following exception: once a contracted list of length less than $n/\log n$ is obtained, list contraction is no longer used; instead, the list ranking problem for this contracted list is solved using the Basic List Ranking Algorithm. This requires time $O(\log n)$ using $n/\log n$ processors.

It is necessary to specify how the list contraction step is carried out. This entails giving a method for choosing the set S, and a method for the compaction process needed to place the elements of S in consecutive locations, in preparation for the recursive solution of the list ranking problem on the compacted list.

We can construct S by the following simple randomized algorithm (see e.g., Vishkin [235]; Miller & Reif [171]) called the *random mate algorithm*. Each element chooses a gender, female or male, with equal probability. An element is in set S if and only if it is female or has a male predecessor. It is easy to see that, with probability $1-o(1)$, the size of S is not more than $15n/16$, and each element in S can find its successor in S in constant time, since the distance to its successor is at most 2. With random mating each list contraction tends to shrink the length of the list by a constant factor, and thus the number of contractions needed to pass from the original list of length n to a list of length less than $n/\log n$ is $O(\log \log n)$.

The process of compacting S into consecutive locations could be done by the $O(\log n)$ time optimal method for prefix sums that we saw earlier, but this method would lead to a list ranking algorithm, running in time $O(\log n \log \log n)$, since $\log \log n$ list contractions need to be performed. Instead, we can use either an optimal $O(\log \log n)$ time randomized algorithm on an ARBITRARY CRCW PRAM that approximately compacts an array (Miller & Reif [171]), or the optimal $O(\log n/\log\log n)$ time deterministic prefix sums algorithm of Cole & Vishkin [56], which runs on a COMMON PRAM. Either of these leads to a method that, with high probability, solves the list ranking problem in time $O(\log n)$ and work $O(n)$. Thus, using Brent's scheduling principle, one obtains an optimal $O(\log n)$ time randomized list ranking algorithm using $n/\log n$ processors.

In order to obtain an optimal $O(\log n)$ time deterministic list ranking algorithm it is necessary to replace the random mating procedure by a deterministic method of isolating a contracted set of elements S. A symmetry breaking technique known as *deterministic coin tossing* can be used for this purpose (Cole & Vishkin [55]). The technique is based on the concept of an *r-ruling set* [55]. Given an n-element list, a subset S of these elements is an r-ruling set if S contains no two adjacent elements of the list, and every element not in S is at a distance no more than $c \cdot r$ on the list from an element in S, where c is a suitable constant.

Define $\log^{(k)} n$ to be the log function iterated k times, and let $r = \log^{(k)} n$. The following algorithm finds an r-ruling set in an n-element linked list in $O(k)$ time using n processors [55] (see also Goldberg, Plotkin & Shannon [102]). We assume that the linked list is doubly linked with successor pointer $s(i)$ and predecessor pointer $p(i)$.

```
RULING SET ALGORITHM
    *Input*: $n$-element linked list with successor pointers $s(i)$ and predecessor pointers $p(i)$;
integer $k$ to set $r = log^{(k)} n$.
(1) **for** $i = 1, \ldots, n$ initialize $c(i) := i$.
(2) **for** $k$ iterations **do**
        **in parallel for** each $i$ **do**
            determine the rightmost bit position $q$ such that the $q$th bit of $c(i)$ differs from
            the $q$th bit of $c(s(i))$; let $b$ be the $q$th bit of $c(i)$; $c(i) := b$ concatenated with the
            binary representation of $q$;
(3) **in parallel for** each $i$ **do**
        **if** $c(p(i)) \leqslant c(i)$ **and** $c(s(i)) \leqslant c(i)$ **then** assign $i$ to the ruling set.
```

It is straightforward to verify that in this algorithm $c(i) \neq c(s(i))$ at every iteration. Further, the maximum value of any $c(i)$ at the end of the jth iteration is $B_j = O(\log^{(j)} n)$. Finally, the distance between two local maxima at the end of the jth iteration is no more than $2 \cdot B_j$, and hence at the end of the algorithm, any element on the list is within distance $O(\log^{(k)} n)$ of an element in the ruling set.

Two special cases of the ruling set algorithm deserve special attention. When $k = c$, a constant, the algorithm obtains an $O(\log^{(k)} n)$ ruling set in constant time using n processors. Define $\log^* n$ as the minimum value of k such that $\log^{(k)} n \leqslant 3$; $\log^* n$ is a very slowly growing function of n. Using the ruling set algorithm we can obtain an $O(1)$ ruling set in $O(\log^* n)$ time with n processors. Since no two elements in a ruling set are adjacent, the size of any r-ruling set is at most one more than half the number of elements in the list. In additional $O(r)$ time, each element in the ruling set can locate its successor in the ruling set by following the successor pointers in the linked list, thus forming a contracted list.

The ruling set algorithm with appropriately chosen values of k has been used in rather elaborate procedures to obtain optimal O(log n) deterministic EREW PRAM algorithms for list ranking (Cole & Vishkin [58]; Anderson & Miller [12]).

2.2.3. *Tree contraction*

There are several applications that require computation on a rooted tree. One such problem is the *expression evaluation problem*: Given a parenthesized arithmetic expression (using + and · operations) with values assigned to the variables, evaluate E and all subexpressions of E. Note that the prefix sums problem is the expression evaluation problem on the parenthesized expression $(\ldots(x_1+x_2)+x_3\ldots)+x_n)$.

Associated with a parenthesized expression is a binary tree with n leaves that specifies the parenthesization. As in the prefix sums problem there is a simple linear-time sequential algorithm for the expression evaluation problem: compute the values on the internal nodes of the expression tree from the leaves upward to the root. The value at the root gives the value of the expression, and the value at each internal node is the value of the subexpression rooted at that node. However, if the tree is highly imbalanced, i.e., its height is large in relation to its size, then this method performs poorly in parallel.

Tree contraction is a method of evaluating expression trees efficiently in parallel. The method transforms the input tree in stages using local operations in such a way that an n-node tree is contracted into a single node in O(log n) stages, each of which takes constant time on a PRAM. An efficient method for tree contraction was first introduced by Miller & Reif [171]. There are several optimal tree contraction algorithms that run in O(log n) time on EREW PRAM (Gibbons & Rytter [98]; Cole & Vishkin [57]; Abrahamson, Dadoun, Kirkpatrick & Przytycka [1]; Kosaraju & Delcher [145]; Gazit, Miller & Teng [97]). We describe the method reported independently in [1] and [145].

The tree contraction algorithm works on a rooted, ordered binary tree, i.e., a rooted ordered tree in which every vertex is either a *leaf*, having no children, or an *internal node*, having exactly two children, and each arc points from a child to its parent. Let l be a leaf in an n-leaf binary tree T. The SHUNT operation applied to l results in a contracted tree T' in which l and $p(l)$, the parent of l in T, are deleted, and the other child l' of $p(l)$ has the parent of $p(l)$ as its parent, while leaving the relative ordering of the remaining leaves unchanged (see Fig. 1).

We now describe the tree contraction algorithm.

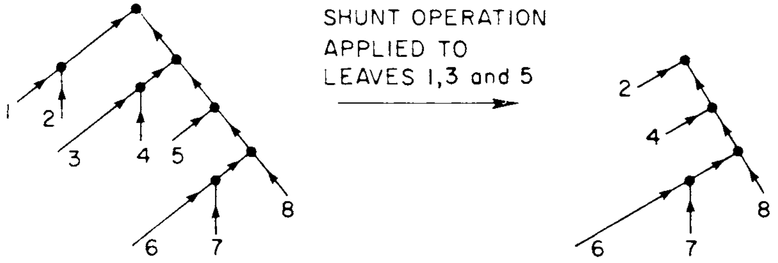

Fig. 1. The SHUNT operation.

```
TREE CONTRACTION ALGORITHM
  Input: a rooted, ordered, binary $n$-leaf tree $T$.
(1) Preprocess: label the leaves in order from left to right as $1, \ldots, n$.
(2) for $\lceil \log n \rceil$ iterations do
    (a) apply SHUNT in parallel to all odd numbered leaves that are the left child of
        their parent;
    (b) apply SHUNT in parallel to all odd numbered leaves that are the right child of
        their parent;
    (c) shift out the rightmost bit in the labels of all remaining leaves.
```

It is straightforward to see that the operations in this algorithm can be implemented on an EREW PRAM. After each iteration, half of the leaves are deleted from the current tree, and no new leaves are created. Hence after $\lceil \log n \rceil$ iterations, the tree is contracted to a single node.

Step 1 can be implemented optimally in $O(\log n)$ time on an EREW PRAM using the *Euler tour technique* (Tarjan & Vishkin [221]), which we describe in Section 2.3. Then, in constant time, the leaves can be placed in an array A in the order in which they will be processed. The total work done in step 2 is

$$O\left(\sum_{i=1}^{\lceil \log n \rceil} n/2^i\right) = O(n)$$

and processor allocation is no problem since we have the array A. Thus this gives an optimal $O(\log n)$ tree contraction algorithm on an EREW PRAM.

By associating appropriate computations with the SHUNT operation, we can evaluate an arithmetic expression while performing tree contraction on its associated tree. We associate with each arch (u, v) an ordered pair of values (a, b) with the interpretation that if the value of u is x, then the operand supplied to v along arc (u, v) is $a \cdot x + b$. Initially, every arc has the ordered pair $(1, 0)$. Thus, initially, the value of each node in the tree is exactly the value of its subexpression.

Now consider a SHUNT operation on a leaf l with parent p, sibling s and grandparent q. Let the value of leaf l be v, and let arc (l, p) have value (a_1, b_1), arc (s, p) have value (a_2, b_2) and arc (p, q) have value (a_3, b_3). In the contracted tree, all three of these arcs are deleted and replaced by the arc (s, q). Let a and b be constants such that

$$a \cdot y + b = a_3 \cdot ((a_1 \cdot v + b_1) * (a_2 \cdot y + b_2)) + b_3,$$

where $*$ represents the operation at p and y is an indeterminate. Then it is easy to see that assigning the value (a, b) to the newly introduced arc (s, q) leaves the values of the vertices remaining in the new tree unaltered. Thus we obtain an optimal $O(\log n)$ time EREW PRAM algorithm for expression evaluation. All subexpressions in the expression tree can be evaluated within the same bounds by having an expansion phase at the end, similar to that in the parallel prefix algorithm. This technique works for the evaluation of expressions over a semiring, ring or field (in the case of a field, the value on each edge is an ordered set of four values, to represent the ratio of two linear forms). If the input is in the form of a parenthesized expression, the tree form can be extracted from it optimally in $O(\log n)$ time using an algorithm by Bar-On & Vishkin [24].

In addition to expression evaluation, tree contraction has been applied to a wide variety of problems. The technique easily generalizes to arbitrary (nonbinary) trees, and has been used to derive parallel algorithms for various graph-theoretic computations on trees such as maximum matching, minimum vertex cover and maximum independent set [115]. Other applications of tree contraction and its variants can be found in [101, 171, 189]. A generalization of the SHUNT operation is used in the more general problem of straight-line program evaluation, or evaluation of a DAG (see Section 4); in fact, the SHUNT operation was first introduced in this more general setting [170].

2.2.4. Conclusion

We have described optimal parallel algorithms for three basic problems: prefix sums, list ranking and expression evaluation.

In Section 3 we show that it requires $\Omega(\log n)$ time to compute the OR of n bits on a CREW PRAM, regardless of the number of processors available. Since the three problems we considered in this section are at least as difficult as the OR function, the lower bound applies for these problems as well. Thus the results we have given are optimal with maximum possible speed-up.

A lower bound of $\Omega(\log n/\log\log n)$ time holds for computing the parity of n bits on a PRIORITY CRCW PRAM with a polynomial number of processors (Beame & Hastad [29]). Since we could solve the parity problem if we could solve an arbitrary prefix sums, list ranking or tree contraction problem, this lower bound applies to these problems as well. While an optimal $O(\log n/\log\log n)$ time CRCW PRAM algorithm is known for the prefix sums problem under certain conditions, it is not known if the list ranking and tree contraction problems have sublogarithmic time algorithms on a CRCW PRAM with a polynomial number of processors.

2.3. Efficient graph algorithms

Graphs play an important role in modeling real-world problems, and sequential algorithms for graph problems have been studied extensively. Almost without exception, all of the optimal (i.e., linear-time) sequential algorithms for these problems use one of two methods to search a graph: *depth-first search* or *breadth-first search.* At present, neither of these techniques has an efficient parallel algorithm. The best polylog-time PRAM algorithm known for breadth-first search of an n-node graph uses $M(n)$ processors, where $M(n)$ is the number of processors required to multiply two $n \times n$ matrices over a general ring in $O(\log n)$ time. The best upper bound presently known for $M(n)$ is $O(n^{2.376})$ (Coppersmith & Winograd [66]). For depth-first search there is currently no deterministic polylog-time parallel algorithm known that uses a polynomial number of processors. For more on parallel breadth-first search and depth-first search, see Section 4.

The early work on parallel graph algorithms [48, 76, 192, 205, 221, 224] used various methods to circumvent the lack of an efficient parallel method of searching a graph. More recently, a new efficient graph searching technique called *ear decomposition* [241, 156, 168, 162] has been developed for undirected graphs. Using this technique, efficient parallel algorithms for several graph problems including strong orientation, biconnectivity, triconnectivity, four-connectivity, and *s*-*t* numbering have been developed. Several of these algorithms also have optimal sequential implementations, thus giving us new algorithms for the sequential case. This is an example of a new emerging discipline enriching an existing one. We briefly survey the results in the following, where we assume that the input graph G has n vertices and m edges.

2.3.1. *Connected components*

The problem of computing connected components is often considered the most basic graph problem. While it is not known how to apply depth-first search or breadth-first search to obtain an efficient parallel connected-components algorithm, the following approach (Hirschberg, Chandra & Sarwate [118]) does give an efficient parallel algorithm for the problem. The algorithm works in $O(\log n)$ stages. At each stage, the vertices of G are organized in a forest of directed trees, with a directed arc from each vertex to its parent in the tree. All vertices in any given tree in the forest are known to be in the same connected component of G. In the first stage of the algorithm, each vertex is in a tree by itself, and at the end of the last stage, all vertices in a connected component are in a tree of height 1. In going from stage i to stage $i+1$, some of the trees containing adjacent vertices in G are linked by a *hooking* process and then the heights of the resulting new trees are compressed by pointer jumping, i.e., each vertex that is not a root or a child of a root in the new tree chooses the parent of its parent to be its new parent. The hooking process has to be implemented properly in order to maintain the tree structure and to guarantee termination of the algorithm in $O(\log n)$ stages. Various implementations of the above basic idea for the CREW PRAM [118], EREW PRAM (Nath & Maheshwari [176]) and CRCW PRAM (Awerbuch & Shiloach [21]; Shiloach & Vishkin [212]) are available. The implementation bound is $O(\log n)$ time using $O(m+n)$ processors on an ARBITRARY CRCW PRAM (recall that by the simulations between the PRAMs, this implies an $O(\log^2 n)$ time algorithm using $O(m+n)$ processors on an EREW PRAM). This algorithm is easily extended to obtain a spanning tree for G with the same time and processor bounds.

By applying more elaborate techniques based on those for optimal list ranking to the basic algorithm we have outlined, the connected components of a graph can be determined on an ARBITRARY CRCW PRAM in $O(\log n)$ time with

$$O((m+n)\alpha(m,n)/\log n)$$

processors (Cole & Vishkin [56]), where $\alpha(m,n)$ is the inverse Ackermann function, which is an extremely slowly growing function of m and n. There is also a randomized optimal $O(\log n)$ algorithm for finding connected components on an ARBITRARY CRCW PRAM (Gazit [94]).

2.3.2. *Euler tour technique*

The starting point for most other graph algorithms is the construction of a rooted spanning tree T, and the computation of simple tree functions such as pre- and postorder numbering of vertices in the tree, the level and height of each vertex in the tree, and the number of descendants of each vertex in the tree. For this, we can use the

Euler tour technique on trees (Tarjan & Vishkin [221]), which we describe below. This method works by reducing the computation of these tree functions to list ranking.

Given an unrooted tree T we can convert it into an Eulerian directed graph D by replacing each edge $\{u, v\}$ by two directed arcs, one from u to v and the other from v to u. Let $E=\langle e_1, e_2, \ldots\rangle$ be an Euler tour of D, with e_1 being the directed arc from u to v. Then it is easy to see that E represents a depth-first search traversal of T with u as the root. Each undirected edge $\{x, y\}$ appears once on the list as the directed arc (x, y), and once as the directed arc (y, x). If (x, y) appears before (y, x) in E, then x is the parent of y in T rooted at u, since in this case, (x, y) represents the forward traversal of the edge in the depth-first search and (y, x) represents the backtracking along the edge in the depth-first search. Thus parent–child relationships in T rooted at u can be determined once we have ranked the elements in list E.

Given a tree T, finding an Euler tour E for its directed Eulerian version is very simple. We assume that the tree is specified by an adjacency list for each vertex, which can be interpreted as a list of outgoing arcs from that vertex. This automatically gives us two arcs directed in opposite directions for each edge. We assume that there is a pointer from each edge to its reversal. With this representation we can obtain an Euler tour E in constant time with n processors by specifying, for each edge e, its next edge on E as the edge next to $\bar{e}$, the reversal of e, in the adjacency list containing $\bar{e}$ (if $\bar{e}$ is the last edge on its adjacency list, its next edge is the first edge on this list): To obtain a depth-first search from a given root u, we simply pick any arc (u, v) as the starting arc of the tour. Now we can use list ranking to determine the position of each arc in E, and hence the parent–child relation. Other tree functions such as preorder number, level, and number of descendants of each vertex can be determined by list ranking using appropriate initial weights. For instance, to compute preorder numbers, we can assign a weight of 1 to forward arcs and a weight of 0 to back arcs. Then, for each forward arc (u, v), the preorder number of v is $n-1-$(the weighted rank of (u, v) in the list). The optimal list ranking algorithm implies optimal $O(\log n)$ time EREW PRAM algorithms for these problems.

We can also use the Euler tour technique on trees to implement the preprocessing step in the Tree Contraction Algorithm of the previous section optimally. For this we merely need to give a weight of 1to leaves and a weight of 0 to internal nodes, and then compute the weighted rank of each leaf in the Euler tour.

The Euler tour technique on trees generalizes to finding Euler tours in general Eulerian graphs with the same complexity bounds as the connected-components algorithm (Atallah & Vishkin [18]; Awerbuch, Israeli & Shiloach [20]). For this, we construct the tour E as above by specifying for each edge, the edge that follows it in E. For a general Eulerian graph, this results in a collection of edge disjoint (possibly nonsimple) cycles. Two cycles having a common vertex u can be "stitched together" by swapping the successor edges of the two incoming arcs to u in the Euler tour E. The algorithm obtains an Euler tour for the graph by stitching all the cycles together into a single connected structure through an appropriate choice of such swaps.

2.3.3. *Ear decomposition*

An *ear decomposition* $D=[P_0, \ldots, P_{r-1}]$ of an undirected graph $G=(V, E)$ is a partition of E into an ordered collection of edge-disjoint simple paths $P_0, \ldots, P_{r-1}$

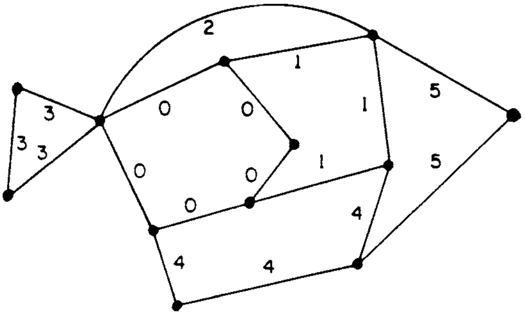

Fig. 2. An ear decomposition.

called *ears*, such that P_0 is a simple cycle, and for $i > 0, P_i$ is a simple path (possibly a simple cycle) with each endpoint belonging to a lower-numbered ear, and with no internal vertices belonging to lower-numbered ears (see Fig. 2). An ear with no internal vertex is called a *trivial ear.*

An *open ear decomposition* is an ear decomposition in which none of the $P_i, i > 0$, is a simple cycle.

It is known that a graph has an ear decomposition if and only if it is 2-edge connected and a graph has an open ear decomposition if and only if it is biconnected, i.e., 2-vertex connected (Whitney [241]).

Let T be a spanning tree of an undirected, 2-edge connected graph $G = (V, E)$, and let N be the set of non-tree edges in G, i.e., the edges in $G-T$. Then each edge e in N completes exactly one cycle in $T \cup \{e\}$, called a *fundamental cycle of G with respect to T*. It is easy to see that the number of ears in any ear decomposition or open ear decomposition of an n-vertex, m-edge graph is $m-n+1$, which is also the number of fundamental cycles in the graph. In the ear decomposition algorithm presented below (Maon, Schieber & Vishkin [162]; Miller & Ramachandran [168]), each ear is generated as part of a fundamental cycle of G with respect to T, and contains the non-tree edge in that fundamental cycle.

```
EAR DECOMPOSITION ALGORITHM
Input: Undirected, 2-edge connected graph $G=(V, E)$.
(1) Preprocess $G$:
    (a) Find a spanning tree $T$ for $G$;
    (b) Root $T$ and number the vertices in preorder;
    (c) label each non-tree edge by the least common ancestor (lca) of its endpoints in $T$.
(2) Assign ear numbers to non-tree edges: number non-tree edges from 0 to $r-1$ in
    nondecreasing order of their lca labels.
(3) Assign ear numbers to tree edges: number each tree edge with the number of the
    minimum-numbered non-tree edge whose fundamental cycle it belongs to.
```

The correctness of this algorithm follows from a straightforward induction on ear number. It is easy to see that the edges with ear number 0 form a simple cycle passing through the root of T, and if we assume inductively that edges with ear numbers 0 to i satisfy the definition of an ear decomposition, it is not difficult to show that the edges with ear number $i+1$ have the desired properties as well.

All of the steps in the algorithm can be implemented using the Euler tour algorithm, together with efficient parallel algorithms for finding a spanning tree, sorting, prefix sums, and finding lca's. The algorithm runs in $O(\log n)$ time while performing $O(m+n \log n)$ work. If consecutive ear numbers are not required, but only distinct labels from a totally ordered set, then the parallel sorting algorithm is not required and the algorithm can be implemented to run in $O(\log n)$ time while performing the same amount of work as the connected-components algorithm, i.e., $O((m+n)\alpha(m,n))$ work, by using an optimal $O(\log n)$ time parallel algorithm to find lca's (Schieber & Vishkin [207]). A randomized version of the ear decomposition algorithm with consecutive ear numbers runs in $O(\log n)$ time while performing $O(m+n)$ work, by using optimal $O(\log n)$ time randomized parallel algorithms for connectivity (Gazit [94]) and for sorting integers in the range 1 to n (Reif [196]).

The ear decomposition algorithm, in general, does not give an open ear decomposition, but can be modified to do so (Maon, Schieber & Vishkin [162]; Miller & Ramachandran [168]) with the same parallel complexity by refining the numbering in step 2 so that non-tree edges with the same lca labels are further ordered in such a way that the resulting ear numbers give an open ear decomposition.

2.3.4. *Applications of ear decomposition*

The efficient parallel ear decomposition algorithm implies efficient parallel algorithms for 2-edge connectivity and biconnectivity. Other efficient parallel algorithms for biconnectivity are known [48, 221].

We describe two other applications of ear decomposition. A *strong orientation* of an undirected graph $G=(V,E)$ is an assignment of a direction to each edge of G such that the resulting directed graph is strongly connected. A graph has a strong orientation if and only if it is 2-edge connected. To strongly orient a 2-edge connected graph, we find an ear decomposition for it, and then orient the edges in each ear from one (arbitrary) endpoint of the ear to the other. Parallel algorithms for strong orientation are reported in [13, 236].

We now briefly describe how to use open ear decomposition to obtain an efficient parallel algorithm to test triconnectivity, and to find the triconnected components of a biconnected graph (Miller & Ramachandran [169]). Let $D=[P_0,\ldots,P_{r-1}]$ be an open ear decomposition of a biconnected graph $G=(V,E)$. If G is not triconnected, then it contains a pair of vertices x, y whose removal separates the graph into two or more pieces. For such a pair of vertices x, y, it is easy to see that there must be some ear P_i in D that contains both of them, such that the portion of P_i between x and y is separated from the rest of P_i when x and y are removed from G. The triconnectivity algorithm tests if such a separating pair of vertices exists by looking for them in each ear in parallel. It does so by constructing, for each nontrivial ear, its *ear graph*, which is a graph derived from the input graph that contains the necessary information about separating pairs of vertices, if any, on the ear. The algorithm then further processes the ear graph to obtain its *coalesced graph*, using which all separating pairs on the ear can be determined quickly. Efficient parallel algorithms for finding the ear graphs of all nontrivial ears and the coalesced graph of each ear graph are given in [169, 190], leading to a parallel graph triconnectivity algorithm that runs in $O(\log n)$ time with $O(m \cdot \log^2 n)$ work on a CRCW PRAM. These ideas generalize to efficient parallel algorithms for finding all separating pairs of vertices in a biconnected graph and for finding the triconnected (or Tutte) components of a biconnected graph. These ideas also generalize to testing for graph four-connectivity (Kanevsky & Ramachandran [129]) giving a parallel algorithm that runs in $O(\log^2 n)$ time with $O(n^2)$ processors on an ARBITRARY CRCW PRAM. This approach also gives an $O(n^2)$ sequential algorithm for the problem, which is an improvement over previously known sequential algorithms for the problem.

Open ear decomposition has been used to obtain a parallel algorithm for finding an s-t numbering in a biconnected graph (Maon, Schieber & Vishkin [162]) with the same complexity as the connected-components algorithm. This efficient parallel s-t numbering algorithm has been used in conjunction with an efficient parallel implementation of PQ trees, to obtain a parallel planarity algorithm (Klein & Reif [143]) that runs in $O(\log^2 n)$ time using $O(m+n)$ processors on a CREW PRAM.

At present there is no efficient graph search technique known for directed graphs. Thus for most problems on directed graphs, including the basic one of testing if one vertex is reachable from another in a directed graph, the best polylog-time parallel algorithm currently known needs on the order of $M(n)$ processors.

2.3.5. *Conclusion*

In this section we have presented efficient parallel algorithms for several problems on undirected graphs.

Using the fact that graph problems are typically at least as hard as computing the OR of n bits or the parity of n bits, it follows from the lower bounds of Section 3 that graph problems need $\Omega(\log n)$ parallel time on a CREW or EREW PRAM with no restriction on the number of processors, and $\Omega(\log n/\log\log n)$ parallel time on a CRCW PRAM with a polynomial number of processors. In practice, graph problems seem to need at least $\log^2 n$ time on a CREW and EREW PRAM and $\log n$ time on a CRCW PRAM. We have stated many of the results in this section only for CRCW PRAMs, when it is the case that, by the simulations between the various types of PRAMs, this gives the best bounds known for CREW and EREW PRAMs as well.

2.4. *Sorting, merging and selection*

In this section we discuss parallel algorithms for which the input is an array of n elements from a linearly ordered set. In the sorting problem, the task is to rearrange the elements into nondecreasing order. In the merging problem, the input array is partitioned into two subarrays, each of which is known to be in nondecreasing order, and the task is to rearrange the entire array into nondecreasing order. In the selection problem an integer k between 1 and n is given, and the task is to find the kth-smallest element of the array. Except for a brief remark in Section 2.4.3. we restrict attention to comparison algorithms, in which the only means of gathering information about the elements is through pairwise comparisons (i.e., tests of the form "is x less than y?", where x and y are elements of the array). For convenience we also assume that the n elements are distinct.

Valiant [229] proposed a model of parallel comparison algorithms in which, at each step, p comparisons are performed, where p is a parameter that we shall call the *degree of parallelism*. It is not required that the p pairs of elements compared at a given step be disjoint. The choice of the p comparisons to be performed at a given step can depend in an arbitrary manner on n, the number of elements, and on the outcomes of previous comparisons. The algorithm terminates when it has acquired enough information about the input to specify the answer (the identity of the kth smallest element in the case of selection, and the permutation required to put the elements in increasing order, in the case of sorting or merging). The execution time of an algorithm is the number of steps performed. This model is called the *parallel comparison model.*

A second model, the *comparator network*, is a restriction of the parallel comparison model. The basic operation in a comparator network is the i–j comparison-exchange operation, where i and j are distinct integers between 1 and n. Such an operation has the following interpretation: compare the contents of location i with the contents of location j; store the smaller of the two values in location i and the larger of the two in location j. If a comparator network has degree of parallelism p, then each of its steps is specified by at most p disjoint i–j pairs, and consists of the simultaneous execution of the comparison-exchange operations corresponding to these pairs. The network is oblivious, in the sense that the set of pairs of locations compared in any given step does not depend on the outcomes of the comparisons performed at earlier steps. A comparator network can be represented by a diagram reminiscent of a sheet of music paper, in which each memory cell is represented by a horizontal line and each i–j comparison-exchange operation is represented by a vertical line segment between horizontal lines i and j, with an arrow directed toward i, the line that will receive the smaller of the two values being compared. The order in which the comparison-exchange operations affecting a given cell are executed is given by the left-to-right order of the corresponding vertical line segments. From this description it should be clear that, in contrast to the more general parallel comparison model, comparator networks can easily be realized in hardware.

A comparator network is called a *sorting network* if it is guaranteed to rearrange the contents of locations 1 to n into nondecreasing order. Figure 3 depicts an 8-element

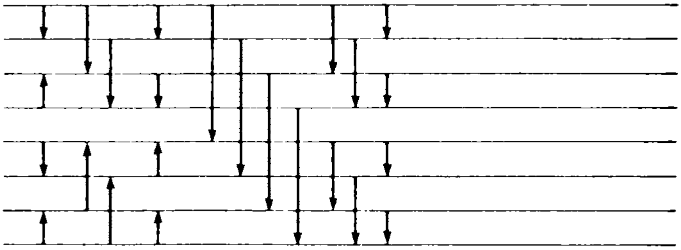

Fig. 3. A sorting network.

comparator network with degree of parallelism 4 and execution time 6. Our discussion of bitonic sort in Section 2.4.2 will establish that this network is a sorting network.

A third model is the comparison PRAM. This is a PRAM for which the input is an array of n elements from a linearly ordered set. In addition to its usual instruction set, the comparison PRAM is provided with instructions for comparing the elements of the input array and for moving them around in memory. Elements of the input array have no interpretation as bit strings or integers, and thus cannot be used as addresses or as arguments for the arithmetic or shift instructions of the PRAM. The comparison PRAM is a more realistic model than the parallel comparison model, because the time required to move data around, as well as the time required to decide which comparisons are to be performed and which processors will perform them, are counted in the cost of computation along with the cost of the comparisons themselves.

Since the parallel comparison model is more permissive than the comparator network or comparison PRAM, lower bounds on the complexity of comparison problems within that model are valid for the other models as well.

2.4.1. The complexity of finding the maximum and merging

In this section we derive tight upper and lower bounds on the complexity of two problems: finding the largest element in an array and merging two sorted arrays. Our lower bounds are for the parallel comparison model, and our upper bounds are for the comparison PRAM. Much of our discussion is based on [229].

We begin by presenting an algorithm for finding the maximum of n elements on a COMMON CRCW comparison PRAM with p processors. At a general step within the algorithm, the search for the maximum will have been narrowed down to a set S of elements from the input array. Initially, S consists of all n elements, and the computation terminates when S becomes a singleton set.

Let x be the smallest integer such that, when S is partitioned into x blocks of size $\lfloor |S|/x \rfloor$ or $\lceil |S|/x \rceil$, p comparisons suffice for the direct comparison of each pair of elements that lie in a common block. Then, in one step, the algorithm determines the maximum element in each block of such a partition, and thus eliminates all but x elements of S.

It can be shown that, when $n/2 \leqslant p \leqslant \binom{n}{2}$, the number of iterations required to reduce the cardinality of S from n to 1 is $\log\log n - \log\log(2p/n) + O(1)$. Also, the computations required at each step for each of the p processors to determine which pair of elements to compare and for the set of block maxima to be assembled into an array can be carried out in constant time. This establishes that the complexity of finding the maximum on a p-processor COMMON CRCW comparison PRAM is $O(\log\log n - \log\log(2p/n))$.

To prove a lower bound in the parallel comparison model we use a graph-theoretic argument. At the beginning of any step of any comparison-based algorithm, there will remain a set S of candidates for the maximum; an element will lie in S if it has never been compared with a larger element. At a general step, the comparisons performed between candidate elements can be represented by the edges of a graph G whose vertices are the candidate elements. An adversary can choose any set S' which is independent in G (i.e.. no two elements of S' are compared with each other in the step), and can consistently specify the outcomes of comparisons so that all the elements of S' remain candidates. A theorem of Turan in extremal graph theory shows that, in any graph G having v vertices and e edges, there is an independent set of size at least $v^2/(v+2e)$. Thus, the adversary can ensure that the candidate set S' is of size at least $|S|^2/(|S|+2p)$. It follows that, against such an adversary strategy, any algorithm requires $\log\log n - \log\log(2p/n)+\mathrm{O}(1)$ steps when $n/2 \leqslant p \leqslant \binom{n}{2}$. Hence, for this range of values of p, the algorithm described above is optimal in the parallel comparison model and optimal within a constant factor on a COMMON CRCW comparison PRAM. Interestingly, on a CREW comparison PRAM, finding the maximum requires time $\Omega(\log n)$, even when no limit is placed on the number of processors, and there is a simple optimal $\mathrm{O}(\log n)$ time EREW PRAM algorithm for the problem.

For the problem of selecting the median of n elements with degree of parallelism n, a lower bound of $\Omega(\log\log n)$ is implied by the lower bound for the maximum problem. A matching upper bound for the comparison model has been given by Ajtai, Komlós, Steiger & Szemerédi [6] building on earlier work by Cole & Yap [59]. It follows from [29] (Beame & Hastad) that finding the median on a CRCW comparison PRAM using a polynomial-bounded number of processors requires time $\Omega(\log n/\log\log n)$.

We now turn to the problem of merging nondecreasing sequences of lengths n and m, where $n \leqslant m$. We begin by giving an algorithm within the parallel comparison model that has degree of parallelism $n+m$ and execution time $\mathrm{O}(\log\log n)$. Let the two sequences be A and B, where $A=(a_1, a_2, \ldots, a_n)$ and $B=(b_1, b_2, \ldots, b_m)$. The algorithm is as follows.

(1) Divide A into $\sqrt{n}$ blocks of length $\sqrt{n}$, and B into $\sqrt{m}$ blocks of length $\sqrt{m}$ (here we ignore the simple modifications needed when the length of A or the length of B is not a perfect square).

(2) Let α_i be the first element of the ith block of A, and β_j the first element of the jth block of B. **In parallel**, compare each α_i with each β_j (the number of processors required is $\sqrt{n}\sqrt{m}$, which is at most $(m+n)/2$).

(3) **In parallel for** each α_i **do**
let $j(i)$ be the unique index such that $\beta_{j(i)} < \alpha_i < \beta_{j(i)+1}$ (here we use the convention that $\beta_0 = -\infty$ and $\beta_{\sqrt{m+1}} = +\infty$).

(4) Compare α_i with each element of the block starting with $\beta_{j(i)}$.

At this point, the algorithm has determined where each α_i fits into B, and thus the problem has been reduced to a set of disjoint merging problems, each of which involves merging a block of length $\sqrt{n}$ from A with some consecutive subsequence of B. Recursively, solve each of these subproblems, using a degree of parallelism equal to the number of elements.

Let $T(n)$ be the time required by this algorithm to merge a sequence of length n with a sequence of length m, where $n \leqslant m$, using $n+m$ processors. Since the parallel computation model charges only for the comparison steps, and not for the bookkeeping involved in keeping track of the results of comparisons and determining which comparisons to perform next, we have $T(1)=1$ and $T(n) \leqslant 2 + T(\sqrt{n})$. This gives $T(n) = \mathrm{O}(\log\log n)$.

This algorithm can be implemented to run in time O(log log n) time with $n+m$ processors on a CREW PRAM, although the processor allocation problems are not entirely trivial (Borodin & Hopcroft [41]).

Following Borodin & Hopcroft we now prove that, in the parallel comparison model, the time required to merge two sequences of length n using $2n$ processors is $\Omega(\log\log n)$. In order to prove the lower bound we consider the following generalization of the problem: given ordered sequences $A_1, B_1, A_2, B_2, \ldots, A_k, B_k$, each of length s, and the information that, for $i=1,2,\ldots,k-1$, each element of $A_i \cup B_i$ is less than each element of $A_{i+1} \cup B_{i+1}$, merge each of the pairs A_i, B_i using cks processors. We refer to this problem as the c, k, s problem.

Let the worst-case time to solve this problem in the parallel comparison model be $t(c, k, s)$, and let $T(c, s) = \min_k t(c, k, s)$ where k ranges over the positive integers. We shall use an adversary argument to prove that $T(c, s) \geqslant 1 + T(16c/7, s')$ where $s' = \sqrt{s/8c}$. Consider the first step in solving a c, k, s problem. We have $ks \times s$ merging problems and cks processors. For $i=1,2,\ldots,k$, partition A_i into $2\sqrt{2cs}$ blocks $A_i^1, A_i^2, \ldots, A_i^{2\sqrt{2cs}}$, each of size s', partition B_i similarly, form a $2\sqrt{2cs} \times 2\sqrt{2cs}$ matrix M_i, and mark the $[a, b]$ cell of this matrix if some element of A_i^a is compared with some element of B_i^b at the first step. Each of the M_i, $i=1,\ldots,k$ has $8cs$ cells, so that the total number of cells in all matrices is $8cks$. At most cks of these cells are marked. Also, each matrix has $4\sqrt{2cs}-1$ diagonals parallel to the main diagonal. By a simple averaging argument, it is possible to choose a diagonal in each of these matrices, such that the number of unmarked cells in the chosen diagonals is at least $7k\sqrt{2cs}/8$. An adversary can specify the outcomes of these comparisons so that each unmarked cell on the selected diagonals corresponds to an independent $s' \times s'$ merging problem at the next step. It follows that the adversary can leave the algorithm with k' independent merging problems to solve, each of which is $s' \times s'$, where $k' = 7k\sqrt{2cs}/8$. The number of processors is cks, which is equal to $16ck's'/7$. This leads to the inequality $T(c, s) \geqslant 1 + T(16c/7, s')$. It follows that $T(c, s) \geqslant d \log\log_c cs$, where d is a certain positive constant. In particular, the complexity of merging two ordered sequences of length n with $2n$ processors is at least $T(2, n)$ which is $\Omega(\log\log n)$. This establishes that the algorithm we have given is optimal up to a constant factor, both for the parallel comparison model and the CREW comparison PRAM.

2.4.2. Sorting networks

One of the classic parallel sorting methods is Batcher's bitonic sorting network [26]. This network is based on the properties of certain sequences. Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence of distinct elements of a linearly ordered set. For $i = 2, 3, \ldots, n-1$, call element a_i a *local minimum* if both a_{i-1} and a_{i+1} are greater than a_i, and a *local maximum* if both a_{i-1} and a_{i+1} are less than a_i. The sequence A is called *unimodal* if it has at most one element that is a local minimum or local maximum, and *bitonic* if it is a cyclic shift of a unimodal sequence. The following lemma is due to Batcher.

LEMMA. *Let $A = (a_1, a_2, \ldots, a_{2N})$ be a bitonic sequence of even length. Define the sequence $L(A)$ and $R(A)$ as follows:*

$$L(A) = (\min(a_1, a_{N+1}), \min(a_2, a_{N+2}), \ldots, \min(a_N, a_{2N}))$$

and

$$R(A)=(\max(a_1,a_{N+1}),\max(a_2,a_{N+2}),\dots,\max(a_N,a_{2N})).$$

Then the sequences $L(A)$ and $R(A)$ are bitonic, and each element of $L(A)$ is less than each element of $R(A)$.

The lemma suggests a parallel method of sorting a bitonic sequence A whose length n is a power of 2: if A is of length 1 then halt; else compare corresponding elements of the left and right half of A and form the arrays $L(A)$ and $R(A)$; in parallel, sort $L(A)$ and $R(A)$. This algorithm can be realized by a comparator network with degree of parallelism $n/2$ and execution time $\log n$.

Batcher's bitonic sorting network is built upon this algorithm for sorting a bitonic sequence. Starting with an unsorted n-element array, which can be regarded as a list of $n/2$ bitonic sequences of length 2, the algorithm constructs a list of $n/4$ bitonic sequences of length 4, then a list of $n/8$ bitonic sequences of length 8, and so forth until the array has been transformed into a single bitonic sequence of length n, which may then be sorted. The algorithm exploits the fact that the concatenation of an increasing sequence with a decreasing sequence is a bitonic sequence. Thus, to convert a list of $n/2^i$ bitonic sequences of length 2^i into a list of $n/2^{i+1}$ bitonic sequences of length 2^{i+1}, it suffices to sort the sequences of length 2^i alternately into increasing and decreasing order, using the algorithm for sorting a bitonic sequence. Figure 3 shows a bitonic sorting network for 8 elements.

We now analyze the execution time of the bitonic sorting algorithm. Let $B(n)$ be the time required to sort a bitonic sequence of length n using degree of parallelism $n/2$, and let $S(n)$ be the time required to sort an array of length n, again using degree of parallelism $n/2$. Then $B(2^k)=k$, and

$$S(2^k)=\sum_{i=1}^{k} B(2^i)=\binom{k+1}{2}$$

Thus bitonic sort requires time $O(\log^2 n)$ to sort n elements using degree of parallelism $n/2$. Bitonic sort is also easily implemented on an EREW comparison PRAM. Again, it runs in time $O(\log^2 n)$ using $n/2$ processors. As an aside, we mention that bitonic sort can also be implemented very neatly to run within these time and processor bounds on certain fixed-degree networks, such as the butterfly and shuffle-exchange network.

Batcher's bitonic sorting network requires $\theta(n\log^2 n)$ comparators. Since there exist sequential sorting algorithms that require only $O(n\log n)$ comparisons in the worst case, it was natural to ask whether one could improve on Batcher's construction by exhibiting sorting networks requiring only $O(n\log n)$ comparators. In 1983 this question was answered affirmatively by Ajtai, Komlós & Szemerédi [7]. The family of sorting networks given by these three authors also has theoretical advantages for parallel computation, since they execute in time $O(\log n)$ using degree of parallelism $n/2$. However, despite the substantial improvements obtained by later researchers (Paterson [183]), the constant implied by the "big O" is so large that bitonic sort is preferable for all practical values of n. Since the Ajtai–Komlós–Szemerédi networks are presented in detail in [186], we shall not attempt to describe them here.

2.4.3. Sorting on a PRAM

Since the sequential complexity of sorting by comparisons is $\theta(n \log n)$, it is of interest to investigate methods for sorting on a comparison PRAM that run in polylog time and have a processor–time product that is $O(n \log n)$. Randomized methods that use $O(n)$ processors and run in $O(\log n)$ time with high probability are given in [201, 200]. The first deterministic method to achieve such performance was an EREW comparison PRAM algorithm based on the Ajtai-Komlós–Szemerédi sorting network. It achieves an execution time of $O(\log n)$ using $O(n)$ processors; however, the constant factor in the time bound is so large as to render the method impractical. Using a new version of bitonic merging, Bilardi & Nicolau [35] give a sorting algorithm for the EREW comparison PRAM that achieves a processor–time product of $O(n \log n)$ using $O(n/\log n)$ processors. Moreover, the constant factor in the time bound is small, so that the method is attractive for practical use. Cole [53] has given a practical deterministic method of sorting on an EREW comparison PRAM in time $O(\log n)$ using $O(n)$ processors. His algorithm can be viewed as a pipelined version of merge sort. Finally, we point out that when the elements to be sorted are integers in a limited range, better bounds are achievable by using bucket sorting methods rather than comparison algorithms [196, 111].

The rest of this section is devoted to a presentation of Cole's sorting algorithm. For simplicity, we confine ourselves to describing a version of the algorithm that runs on a CREW comparison PRAM. We assume for convenience that the number of elements to be sorted is a power of 2. Let the 2^k elements to be sorted be placed in correspondence with the leaves of a complete binary tree T of height k; hereafter, we make no distinction between a leaf and the corresponding element. Each internal node v within T is the root of a subtree; let T_v be the set of leaves of that subtree. Then the task of node v is to arrange the elements of T_v into a sorted list.

An obvious method of creating the required lists would be to move up the tree level-by-level from the leaves to the root, using the merging algorithm of Section 2.4.1 to create the list for each node by merging the lists for its two children. Using the fact that the time to merge two sorted lists of length t using $2t$ processors is $O(\log \log t)$, a simple analysis shows that this obvious method runs in time $O(\log n \log \log n)$ using $O(n)$ processors. Cole improves on this approach by having the algorithm work on many levels of the tree at once, creating successively more refined approximations to the lists that the nodes must eventually produce. The method of approximation is chosen so that each approximation to the final list for a node can be obtained from the preceding approximation in constant time.

Associated with any time step s and internal node v is a list $LIST_v(s)$; this list is an increasing sequence of elements drawn from T_v. We say v is *finished* at time s if $LIST_v(s)$ contains all the elements of T_v. Node v is a *frontier node* when v is finished but its parent is not. Initially, the leaves of T are its frontier nodes. At any time s, all the frontier nodes are at the same distance from the root of T.

At every step, each frontier node or unfinished node passes a subsequence of its list up to its parent. For node v at step s we call this subsequence $UP_v(s)$. Node v forms $LIST_v(s)$ by merging $UP_x(s)$ with $UP_y(s)$, where x and y are the children of v. If x is unfinished at step $s-1$ then $UP_x(s)$ consists of every fourth element of $LIST_x(s-1)$; i.e., elements $1, 5, 9, 13, \ldots$ If x becomes finished at step s then $UP_x(s+1)$ consists of every fourth element of $LIST_x(s)$, $UP_x(s+2)$ consists of every second element of $LIST_x(s)$ and $UP_x(s+3)$ is equal to $LIST_x(s)$. It follows that a node becomes finished three steps after its children do. The sorting process is completed after $3k$ steps, when the root becomes finished.

It remains to show that each step can be completed in constant time on a CREW PRAM using $O(n)$ processors. At each step the sum of the lengths of all the lists associated with unfinished or frontier nodes is $O(n)$. Thus it suffices to give a method of merging the lists $UP_x(s)$ and $UP_y(s)$ in constant time, using one processor per list element. The key idea is to use information from previous steps. Given two ordered lists A and B, define a *cross-link array* from A into B to be an array of pointers from A into B such that each element a in A points to the least element in B that is greater than a (or, if no such element exists, to a sentinel placed at the end of the list B). If the ordered lists A and B are disjoint then, given cross-links from A into B and from B into A, each element can quickly calculate its rank order in the list that results from merging A and B, and thus A and B can be merged in constant time. Cole's algorithm maintains certain cross-link arrays in order to speed up the merging process. If nodes x and y are siblings in T then, upon the completion of step $s-1$, the algorithm maintains cross-links from $UP_x(s-1)$ into $UP_y(s-1)$ and $LIST_x(s-1)$, and, symmetrically, from $UP_y(s-1)$ into $UP_x(s-1)$ and $LIST_y(s-1)$. Given the cross-links from $UP_x(s-1)$ into $LIST_x(s-1)$ it is easy, in constant time, to create cross-links from $UP_x(s-1)$ into $UP_x(s)$. Similarly, cross-links can be constructed in constant time from $UP_y(s-1)$ into $UP_y(s)$. Moreover, it can be shown that at most four elements of $UP_x(s-1)$ point to any element of $UP_x(s)$. Given all this cross-link information, it is possible in constant time on a CREW comparison PRAM to create cross-links between $UP_x(s)$ and $UP_y(s)$, to merge those two sequences, and to create the cross-links required for the next step. It follows that each step can be executed in constant time using $O(n)$ processors on a CREW comparison PRAM, and thus Cole's algorithm sorts in time $O(\log n)$ using $O(n)$ processors on a CREW comparison PRAM.

2.5. *Further topics*

Efficient parallel algorithms have been developed for a number of other combinatorial problems, including string matching [88, 137, 151, 237, 33], set manipulation [17], computational geometry [4, 14, 15, 16, 51, 54, 108, 198] and graph algorithms [52, 112, 142]. In [78, 99] extensive surveys of efficient parallel algorithms are given including the topics of Sections 2.1–2.3.

Some of the challenges that have thus far eluded researchers in the area of efficient and optimal combinatorial parallel algorithms are the following:

(i) The construction of an efficient parallel $O(\log n/\log\log n)$ time CRCW PRAM algorithm for the n-element list ranking problem. At present it is not known how to achieve this time bound with any polynomial-bounded number of processors.

(ii) The construction of an efficient parallel algorithm for breadth-first search in an undirected graph. This problem can be solved on a CRCW PRAM in O(log n) time using a polynomial-bounded number of processors (see the discussion of the "transitive closure bottleneck" in Section 4.8.1).

(iii) The construction of efficient parallel algorithms for various reachability problems on directed graphs, including reachability from one given vertex to another, breadth-first search, strong connectedness and topological sorting of a directed acyclic graph. These problems can be solved on a CRCW PRAM in O(log n) time using a polynomial-bounded number of processors (see Section 4.8.1).

(iv) The construction of efficient algorithms to find the connected components of an undirected graph in time o($\log^2 n$) on a CREW PRAM or time o(log n) on a CRCW PRAM. The known lower bounds are $\Omega(\log n)$ and $\Omega(\log n/\log\log n)$ respectively.

(v) The construction of n-element sorting networks with degree of parallelism at most Cn and execution time $D \log n$, where C and D are moderate constants (rather than the very large constants achieved in the construction of Ajtai, Komlós, & Szemerédi [7]).

(vi) The construction of an optimal deterministic algorithm for sorting n integers in the range $[1, \ldots, n^k]$, for fixed k, on an EREW PRAM in time O(log n). Reif [196] gives an optimal randomized O(log n) time algorithm on an ARBITRARY CRCW PRAM to sort n integers in the rage $[1, \ldots, n]$, and Hagerup [111] gives a deterministic algorithm to sort n integers from $[1, \ldots, n^k]$ in O(log n) time using O($n \log\log n/\log n$) processors on a PRIORITY CRCW PRAM; this algorithm requires space $\Omega(n^{1+\varepsilon})$, where ε is an arbitrarily small positive constant.

3. Models of parallel computation

3.1. *Relations between PRAM models*

Our primary model for parallel computation is the PRAM family [85, 106, 206]. In Section 2.1 PRAMs were classified according to restriction on global memory access as EREW, CREW or CRCW, and CRCW PRAMs were further classified as COMMON, ARBITRARY or PRIORITY. It should be noted that this listing represents the PRAM models in increasing order of their power. Thus, any algorithm that works on an EREW PRAM works on a CREW PRAM, and in turn, any algorithm on a CREW PRAM works on a COMMON CRCW PRAM, and so on. The most powerful model in this spectrum is the PRIORITY CRCW PRAM.

We now show that any algorithm for a PRIORITY CRCW PRAM can be simulated by an EREW PRAM with the same number of processors and with the parallel time increased by only a factor of O(log P), where P is the number of processors [77, 234]. This is done as follows: Let $P_1, \ldots, P_r$ be the processors and $M_1, \ldots, M_s$ be the memory locations used by the PRIORITY algorithm. The simulating EREW PRAM uses r auxiliary memory locations $N_1, \ldots, N_r$ for simulating a write or read step. If processor P_i needs to access location M_j in the PRIORITY algorithm, then it writes the ordered pair (j, i) in location N_i. The array N_j, $j = 1, \ldots, r$ is then sorted in lexicographically increasing order in O(log r) time using the r processors [53]. Then, by reading adjacent entries in this sorted array, the highest priority processor accessing any given location can be determined in constant time. For a write instruction, these processors then execute the write as specified in the PRIORITY algorithm. For a read instruction, the processors read the specified locations, and then in additional O(log r) time, duplicate the value read so that there are enough copies of each value for all the processors that need to read it.

We also have the result that any algorithm for a PRIORITY PRAM can be simulated by a COMMON PRAM with no loss in parallel time provided sufficiently many processors are available [148]: Let $P_1, \ldots, P_r$ be the processors used by the PRIORITY algorithm. The simulating COMMON algorithm uses auxiliary processors $P_{i,j}$ and memory locations M_j, $1 \leqslant i,j \leqslant r$. The locations M_j are initialized to 0. Processor $P_{i,j}$, $i<j$ determines the memory addresses m_i and m_j that processors P_i and P_j were to access in the PRIORITY algorithm and writes a 1 in location M_j if $m_i = m_j$. Now P_j can ascertain if it is the lowest-numbered processor that needs to write into m_j by testing if M_j is still zero. If so, it writes into m_j the value it was supposed to write by the PRIORITY algorithm.

3.2. *Lower bounds for PRAMs*

There is a substantial body of literature which explores lower bounds on the time required by EREW, CREW or CRCW PRAMs to perform simple computational tasks. For the purpose of proving such lower bounds it is customary to adopt a model called the ideal PRAM, in which no limits are placed on the computational power of individual processors or on the capacity of a cell in the shared memory. Each processor in a PRAM can compute an arbitrary function of the values in its private memory at each step, and thus the ideal PRAM is much more powerful than the ordinary PRAM, whose processors are restricted to executing conventional RAM instructions. Lower bounds proved within such a powerful model of computation have great generality because they do not depend on particular assumptions about the instruction set or internal structure of the individual processors. Such lower bounds capture the intrinsic limitations of global memory as a means of communication between processors, and demonstrate clear distinctions in power among the various concurrent-read and concurrent-write arbitration mechanisms.

An *ideal PRAM* consists of processors which communicate through a global memory divided into cells of unbounded storage capacity. Each processor has a private memory of unbounded size and the ability to compute in unit time any function of the contents of its private memory. The input data are assumed to be stored in locations $M_1, M_2, \ldots, M_n$ of the global memory, and the computation is required to terminate with its output in location M_1. The computation proceeds in steps, with each step consisting of a read phase, a compute phase and a write phase. In the read phase, each processor reads into its private memory the contents of one cell in the gobal memory. In the compute phase, each processor computes some function of the contents of its private memory. In the write phase, each processor stores a value dependent on the contents of its private memory in some cell of global memory. An ideal PRAM is designated as EREW, CREW or CRCW according to whether concurrent reading and/or concurrent writing are permitted, and concurrent-write ideal PRAMs are further classified as COMMON, ARBITRARY, PRIORITY, etc. according to the method of write-conflict resolution. Even the weakest of these models, the EREW ideal PRAM, is so powerful that any function of n variables can be computed in time $O(\log n)$ using n processors, simply by assembling all the input data together in the private memory of one processor, which can then use its unlimited computation power to determine the output and store it in the global memory.

In the paper by Cook, Dwork & Reischuk [63] it is shown that the OR function requires time $\Omega(\log n)$ on an ideal CREW PRAM, no matter how many processors or memory cells are used. Here each input cell contains a bit. The output is 0 if all the input bits are zero, and 1 otherwise. Since this function can be computed in constant time by a COMMON CRCW PRAM with n processors and a very limited instruction set, this result clearly demonstrates that the concurrent-write mechanism is strictly more powerful than exclusive-write. This lower bound may appear obvious, since, at first sight, there seems to be no method of solving the problem on a CREW PRAM better than halving the number of inputs at each step by "OR"ring them together in pairs. But an alternate method can be given with runs in time $\log_{2.618} n + O(1)$ on a CREW PRAM, and thus beats the obvious halving method.

The lower bound proof requires the following definitions. Let us say that input bit i *affects* processor P at time t if the contents of the processor's private memory at time t differ according to whether input i is 1 or 0, when the other input bits are fixed at 0. Similarly, we can speak of an input bit affecting global memory cell M at time t. By induction on t, one can prove that the number of input bits affecting any processor or memory cell at time t is at most c^t, where c is a suitable constant. Since every input bit must affect the output cell at the end of the computation, it then follows that the computation time is at least $\log_c n$.

Extending the work of Cook, Dwork & Reischuk, Nisan [177] has recently given the following very precise characterization of the time required by an ideal CREW PRAM to compute a finite function f with domain B^n, where $B = \{0, 1\}$. For any $w \in B^n$ and any subset S of the index set $\{1, 2, \ldots, n\}$, let us say that f is *sensitive* to S at w if the value of f changes when w is changed by flipping those input bits with indices in S. We say that f is *k-block sensitive* at w if f is sensitive at w to each of k disjoint index sets. The *block-sensitivity* $bs(f)$ is defined as the largest k such that, for some w, f is k-block sensitive at w. Then the time required to compute f on an ideal CREW PRAM is bounded both above and below by bounds of the form $c \log(bs(f)) + d$, where c and d are constants.

The paper by Snir [217] studies the complexity of solving the following table look-up problem using p processors: given an array of distinct integers $\langle x_1, x_2, \ldots, x_n, y\rangle$ where $x_1 < x_2 < \cdots < x_n$, find the index i such that $x_i < y < x_{i+1}$ (by definition, $x_0 = -\infty$ and $x_{n+1} = \infty$). The problem can be solved on a CREW PRAM with a conventional instruction set in time $O(\log_{p+1} n)$ using a variant of binary search, and a simple adversary argument shows that $\log_{p+1} n$ is also a lower bound for the problem on an ideal CREW PRAM. Snir proves that the problem requires time $\Omega(\log n - \log p)$ on an ideal EREW PRAM, thereby showing that the concurrent-read PRAM is strictly more powerful than the exclusive-read PRAM.

One component of Snir's proof is a Ramsey-theoretic argument showing that, when the x_i are allowed to be arbitrarily large integers, one can restrict attention to algorithms in which the only information gathered about the x_i is obtained through comparing the x_i directly with y; i.e., for each algorithm that violates this restriction, there is another algorithm that respects the restriction and has equally good worst-case behavior. In view of this restriction one can rephrase the problem as the following *zero-counting problem*: given an array of m 0s followed by $n-m$ 1s, determine m. Here a 0 in position i means that $x_i < y$, and a 1 in position i means that $x_i < y$. At the end of the computation, cell M_m is to contain a 1 if the answer is m, and a 0 otherwise.

We sketch Snir's proof of a lower bound on the time required to solve the zero-counting problem on an ideal EREW PRAM with p processors. For any processor P and time t, say that input coordinate *m affects P* at time t if the contents of the private memory of P at time t is different on input $0^{m-1}1^{n-m+1}$ than it is on input $0^m 1^{n-m}$. Let $P(t)$ be the set of input coordinates that affect P at time t. Similarly, let $M(t)$ be the set of input coordinates that affect global memory cell M at time t. As a measure of the progress of the computation, define

$$c(t)=\sum_P |P(t)|+\sum_M \max(0,|M(t)|-1),$$

where the first summation is over all processors, and the second is over all global memory cells. Then $c(0)=0$, and if the computation halts at time T then $c(T)\geqslant n+1$. Snir proves the inequality $c(t+1)\leqslant 4c(t)+p$, and the lower bound $T=\Omega(\log n-\log p)$ follows. By a similar argument Snir proves that when the number of processors is unlimited, there is a lower bound of $\Omega(\sqrt{\log n})$ on the parallel time for the table look-up problem.

The paper by Beame & Hastad [29], improving an earlier result in Beame [27], proves that, on an ideal PRIORITY CRCW PRAM, the number of processors required to compute the parity of n bits in time T is at least $^{\lfloor (n^{1/T}/96)-2 \rfloor}$. It follows that the time required to compute the parity of n bits of an ideal CRCW PRAM using a polynomial-bounded number of processors in $\Omega(\log n/\log\log n)$.

Many further lower bounds for ideal PRAMs can be cited. The papers [36, 81, 82, 110, 153, 166, 238] study the effects of the size of global memory and the choice of a write-conflict mechanism on the time required to solve problems on an ideal CRCW PRAM. These results show that, in various settings, the ARBITRARY model is strictly more powerful than the COMMON model, but strictly less powerful than the PRIORITY model.

3.3. Circuits

So far we have mainly considered the PRAM model. There are several other models of parallel computation, and of these, the *circuit model* has emerged as an important medium for defining parallel complexity classes. By a *circuit* we mean a bounded fan-in combinational Boolean circuit. More formally, a *circuit* is a labeled directed acyclic graph (DAG). Nodes are labeled as *input*, *constant*, *AND*, *OR*, *NOT*, or *output* nodes. Input and constant nodes have zero fan-in, AND and OR nodes have fan-in of 2, NOT and output nodes have fan-in of 1. Output nodes have fan-out 0.

Let $B=\{0,1\}$. A circuit with n input nodes and m output nodes computes a Boolean function $f: B^n \rightarrow B^m$, where we assume the input nodes to be ordered as $\langle x_1, \ldots, x_n \rangle$ and similarly the output nodes as $\langle y_1, \ldots, y_m \rangle$. The *size* of a circuit is the number of edges in the circuit. The *depth* of the circuit is the length of a longest path from some input node to some output node. We note that the size of a circuit is a measure of its hardware content and its depth measures the time required to compute the output, assuming unit delay at each gate.

A rather general formulation of a *problem* is as a *transducer of strings over B*: i.e., as a function from B^* to B^*. By using a suitable encoding scheme, we can assume, without loss of generality, that the size n of the input string determines the size $l(n)$ of the output string. Let $C=\{C_i\}, i=1,2,\ldots$ be a family of circuits for which C_i has i input bits and $l(i)$ output bits. Then the family of circuits C *solves* a problem P if the function computed by C_i defines precisely the string transduction required by P for inputs of length i.

When using a family of circuits as a model of computation, it is necessary to introduce some notion of *uniformity* if we wish to correlate the size and depth of a family C that solves a problem P with the parallel time and hardware complexity of P. If not, we could construct a family of circuits of constant size and depth to solve an undecidable problem (for which all inputs of a given length have the same one-bit output; it is well-known that nonrecursive sets with this property exist). A notion of uniformity that is commonly accepted for parallel computation is *logspace uniformity* [203, 62]: A family of circuits C is logspace uniform if the description of the nth circuit C_n can be generated by a Turing machine using $O(\log n)$ workspace. By the description of a circuit, we mean a listing of its nodes, together with their type, followed by a listing of the inputs to each node. For $C(n) \geqslant n$ and $D(n) \geqslant \log n$, a problem P is said to be in the class $\mathrm{CKT}(C(n), D(n))$ if there is a logspace uniform family of circuits $C=\{C_n\}$ solving P such that C_n is of size $O(C(n))$ and depth $O(D(n))$.

The class NC^k, $k>1$, is the class of all problems that are solvable in $\mathrm{CKT}(poly(n), O(\log^k n))$, where $poly(n)=\bigcup_{k\geqslant 1} O(n^k)$. For technical reasons we define NC^1 to be the class of problems solvable by alternating Turing machines in time $O(\log n)$ (see Section 3.5). The class $\mathrm{NC}=\bigcup_{k\geqslant 1}\mathrm{NC}^k$ is generally accepted as a characterization of the class of problems that can be solved with a high degree of parallelism using a feasible amount of hardware (Cook [61], Pippenger [184]). As mentioned in Section 2, we refer to the quantity $\bigcup_{k\geqslant 1} O(\log^k n)$ as $polylog(n)$. Thus $\mathrm{NC}=\mathrm{CKT}(poly(n), polylog(n))$. For many commonly used models of parallel computation, the class of problems that can be solved in polylog time with a polynomial-bounded amount of hardware coincides with NC.

If we remove the fan-in restriction on AND and OR gates in the circuit model, we obtain the *unbounded fan-in circuit* model, where, as before, the size of the circuit is the number of edges in the circuit, and the depth is the length of a longest path from an input node to an output node in the unbounded fan-in circuit [47, 62, 86, 220].) A family of unbounded fan-in circuits is logspace uniform if the description of the ith circuit can be generated in logspace. By analogy with the bounded fan-in case, the class $\mathrm{UCKT}(C(n), D(n))$, where $C(n) \geqslant n$, is defined as the class of problems solvable by logspace-uniform circuit families whose nth element has size $O(C(n)$ and depth $O(D(n))$. For $k \geqslant 1$, *the class* $\mathrm{UCKT}(poly(n), \log^k n)$ is denoted AC^k. The class $\mathrm{AC}=\bigcup_{k\geqslant 1}\mathrm{AC}^k$.

Since any gate in an unbounded fan-in circuit of size $p(n)$ can have fan-in at most $p(n)$, each such gate can be converted into a tree of gates of the same type with fan-in 2, such that the output gate computes the same function as the original gate. By applying this transformation to each gate in an unbounded fan-in circuit in $\mathrm{UCKT}(p(n), d(n))$ we obtain a bouned fan-in circuit in

$$\mathrm{CKT}(\mathrm{O}(p(n)), \mathrm{O}(d(n) \cdot \log p(n))).$$

It is straightforward to see that this transformation is in logspace. Thus $\mathrm{AC}^k \subseteq \mathrm{NC}^{k+1}$ for $k \geqslant 1$. Clearly $\mathrm{NC}^k \subseteq \mathrm{AC}^k$. Thus we conclude that $\mathrm{AC} = \mathrm{NC}$. We also note that we can always compress $\mathrm{O}(\log \log n)$ levels of a bounded fan-in circuit into two levels of a polynomial-size, unbounded fan-in circuit [47] and hence

$$\mathrm{CKT}(poly(n), \log^k n) \subseteq \mathrm{UCKT}(poly(n), \log^k n / \log \log n).$$

3.4. Relations between circuits and PRAMS

If we assume a bounded amount of local computation per processor per unit time, we can establish a strong correspondence between the computational power of unbounded fan-in circuits and that of CRCW PRAMs [220]. More specifically, assume the PRIORITY write model, and that the instruction set of the individual RAMs consists of binary addition and subtraction of poly-size numbers, binary Boolean operations, left and right shifts, and conditional branching on zero, and that indirect memory access is allowed. Each instruction is assumed to execute in unit time. The input to a problem of size n is specified by n n-bit numbers. Let $\mathrm{CRCW}(P(n), T(n))$ be the class of problems solvable on such a PRAM in $\mathrm{O}(T(n))$ time with $\mathrm{O}(P(n))$ processors.

Given any unbounded fan-in circuit in $\mathrm{UCKT}(S(n), D(n))$ we can simulate it on a CRCW PRAM in time $\mathrm{O}(D(n))$ with $S(n)$ processors and $n + M(n)$ memory locations, where $M(n)$ is the number of gates in the circuit. Any bounded fan-in or unbounded fan-in circuit can be converted in logspace into an equivalent circuit of comparable size and depth for which negations occur only at the inputs. These negations can now be removed by supplying the complement value as input. So we can assume that our input circuit has no negations. In the simulation, each processor is assigned to an edge in the circuit and each memory location to a gate. Initially, the inputs are in memory locations 1 through n, and memory location $n + i$ is assigned to gate i in the circuit, $i = 1, \ldots, M(n)$; these latter locations are initialized to 0 for OR gates and to 1 for AND gates.

Each step in the simulation consists of performing the following three substeps in sequence:

(a) Each processor p determines the current value c on its edge $e = (u, v)$ by reading memory location $n + u$.

(b) If $c = 0$ and v is an OR gate, or if $c = 1$ and v is an AND gate then p writes value c into location $n + v$.

(c) If $c = 1$ and v is an OR gate, or if $c = 0$ and v is an AND gate then p writes c into location $n + v$.

Clearly, after $D(n)$ steps, memory location $n + i$ has the value of gate i for $i = 1, \ldots, M(n)$. Thus

$$\mathrm{UCKT}(S(n), D(n)) \subseteq \mathrm{CRCW}(S(n), D(n)).$$

For the reverse part, we note that each of the binary operations in the instruction set of the PRAM can be implemented by constant-depth, polynomial-size, unbounded fan-in circuits, as shown in Section 4. It is also fairly easy to implement the conditional branching by such circuits, by suitably updating the program counter. The nontrivial part of the simulation of a CRCW PRAM by an unbounded fan-in circuit lies in simulating the memory accesses. Since a combinational circuit has no memory, the simulation retains all values that are written into a given memory location during the computation, and has a bit associated with each such value that indicates whether the value is current or not. With this system, it is not difficult to construct a constant-depth unbounded fan-in circuit to implement reads and writes into memory. Thus a single step of the PRIORITY CRCW PRAM can be simulated by an unbounded fan-in circuit of polynomial size and constant depth, and it follows that

$$\mathrm{CRCW}(P(n), T(n)) \subseteq \mathrm{UCKT}(poly(P(n)), T(n)).$$

Let us call a PRAM algorithm *logspace uniform* if there is a logspace Turing machine that, on input n, generates the program executed by each processor on inputs of size n. (We note that all of the PRAM algorithms we describe have programs that are parametrized by n and the processor number, and such algorithms are clearly logspace uniform.) Define CRCW^k as the class of problems solvable by logspace uniform CRCW PRAM algorithms in time $\mathrm{O}(\log^k n)$ using a polynomial-bounded number of processors; let CREW^k and EREW^k be the analogous classes for CREW PRAMs and EREW PRAMs. We will see in Section 3.9 that NL (nondeterministic logspace) is in CRCW^1, and that L (deterministic logspace) is in EREW^1. Thus, by the results of Stockmeyer & Vishkin we have just described [220], we have $\mathrm{CRCW}^k = \mathrm{AC}^k$ for $k \geqslant 1$, and

$$\bigcup_{k>0} \mathrm{CRCW}(poly(n), \log^k n) = \mathrm{NC}.$$

Since we also noted earlier that simulations between the various types of PRAM result in only an $\mathrm{O}(\log P(n))$ increase in time and a squaring of the processor count, we have

$$\mathrm{PRAM}(poly(n), polylog(n)) = \mathrm{NC},$$

where the PRAM processor and time bounds can refer to any of the PRAM models.

It is shown in Hoover, Klawe & Pippenger [119] that any bounded fan-in circuit of size $S(n)$ and depth $D(n)$ can be converted into an equivalent circuit of size $\mathrm{O}(S(n))$ and depth $\mathrm{O}(D(n))$ having both bounded fan-in and bounded fan-out. Using this result, it is easy to see that $\mathrm{NC}^k \subseteq \mathrm{EREW}^k$. Thus we have the following chain of inclusions for $k \geqslant 1$:

$$\mathrm{NC}^k \subseteq \mathrm{EREW}^k \subseteq \mathrm{CREW}^k \subseteq \mathrm{CRCW}^k = \mathrm{AC}^k \subseteq \mathrm{NC}^{k+1}.$$

As noted earlier, we also have

$$\mathrm{NC}^k \subseteq \mathrm{UCKT}(poly(n), \log^k n/\log\log n).$$

Earlier in this section, we referred to the lower bound of $\Omega(\log n/\log\log n)$ for computing parity on an ideal PRIORITY CRCW PRAM with a polynomial number of processors [29]. This immediately gives the same lower bound for computing parity with an unbounded fan-in circuit of polynomial size. Historically this lower bound was first developed for the case of unbounded fan-in circuits in a sequence of papers [86, 5, 243, 113, 114, 216], thus implying the lower bound for CRCW PRAMs with restricted instruction set. The extension of the result to ideal PRAMs requires a substantial refinement of the probabilistic restriction techniques used to obtain the lower bounds for circuits. For bounded fan-in circuits, a lower bound of $\Omega(\log n)$ for the depth required to compute any function whose value depends on all n input bits is easily established by a simple fan-in argument.

3.5. *Alternating Turing machines*

Another important model of parallel computation is the *alternating Turing machine* (*ATM*) [46, 203]. An ATM is a generalization of a nondeterministic Turing machine whose states can be either existential or universal. Some of the states are *accepting* states. As with regular Turing machines, we can represent the *configuration* α of an ATM at a given stage in the computation by the current state, together with the current contents of worktapes and the current positions of the read and worktape heads. A configuration is *accepting* if it contains an accepting state. The space required to encode a configuration is proportional to the space used by the computation on the worktapes. Configuration β *succeeds* configuration α if α can change to β in one move of the ATM. For convenience we assume that an accepting configuration has transitions only to itself, and that each nonaccepting configuration can have at most two different configurations that succeed it.

We define the concept of an *accepting computation* from a given initial configuration α. If α is an accepting configuration, then α itself comprises an accepting computation. Otherwise, if α is existential, then there is an accepting computation from α if and only if there is an accepting computation from some configuration β that succeeds α; and if α is universal, then there is an accepting computation from α if and only if there is an accepting computation from every configuration β that succeeds α. Thus we can represent an accepting computation of an ATM on input x as a rooted tree whose root is the initial configuration, whose leaves are accepting configurations, and whose internal nodes are configurations such that if c is a node in the tree representing a configuration with an existential state, then c has one child in the tree which is a configuration that succeeds it, and if c represents a configuration with a universal state, then the children of c are all the configurations that succeed it. An ATM *accepts* input x if it has an accepting computation on input x. A node α at depth t in the computation tree represents the event that the ATM can reach configuration α after t steps of the computation, and will be denoted by the ordered pair (α, t). The *computation DAG* of an accepting computation of an ATM on input x is the DAG derived from the computation tree by identifying together all nodes in the tree that represent the same ordered pair.

Alternating Turing machines are generally defined as acceptors of sets. We can view them as transducers of strings by considering the input as an ordered pair $\langle w, i\rangle$ and the output bit as specifying the ith bit of the output string when the input string is w [61]. Here we assume that the length of the output is at most exponential in the input length.

An ATM M operates in time $T(n)$ if, for every accepted input of length n, M has an accepting computation of depth $O(T(n))$. Similarly, an ATM operates in space $S(n)$ if, for every accepted input of length n, M has an accepting computation in which the configurations require at most $O(S(n))$ space. We define $\mathrm{ATM}(S(n), T(n))$ to be the class of languages that are accepted by ATMs operating simultaneously in time $O(T(n))$ and space $O(S(n))$. Also, let $\mathrm{ATIME}(T(n))$ denote the class of languages accepted by ATMs operating in time $O(T(n))$.

In order to allow sublinear computation times, we use a *random-access model* to read the input tape: when in a specified *read* state, we allow the ATM to write a number in binary which is then interpreted as the address of a location on the input tape, whose symbol is then read onto the worktape in unit time. Thus $\log n$ time suffices to read any input symbol.

We shall now show that, for $k \geqslant 1$, $\mathrm{ATM}(\log n, \log^k n) = \mathrm{NC}^k$ (Ruzzo [203]). This can be seen as follows. Consider a language accepted by an ATM M in $\mathrm{ATM}(S(n), T(n))$, with $S(n) = \Omega(\log n)$. The *full computation DAG D* of M for inputs of length n is obtained by having a vertex for each pair (α, t), where α is a configuration of the ATM in space $O(S(n))$ and $0 \leqslant t \leqslant T(n)$, and an arc from vertex (α, t) to vertex $(\beta, t-1)$ if α succeeds β. We can construct a circuit from D by replacing each existential node in D by an OR gate, each universal node by an AND gate, each accepting leaf by a constant input 1, each nonaccepting leaf by a constant input 0, and each node in which the state is a read state by the corresponding input. The output of the circuit corresponds to vertex $(\alpha_0, 0)$, where α_0 is the initial configuration. It is not difficult to see that the output of this circuit is 1 if and only if the input is accepted by the ATM in the prescribed time and space bounds. Further the depth of this circuit is $O(T(n))$ and its size is $O(c^{S(n)})$ for a suitable constant c. Thus,

$$\mathrm{ATM}(S(n), T(n)) \subseteq \mathrm{CKT}(c^{S(n)}, T(n)).$$

Also, since the resulting family of circuits is easily shown to be logspace uniform, we have

$$\mathrm{ATM}(\log n, \log^k n) \subseteq \mathrm{NC}^k \quad \text{for } k \geqslant 2, \quad \text{and} \quad \mathrm{ATM}(\log n, \mathit{polylog}(n)) \subseteq \mathrm{NC}.$$

For the reverse, consider any logspace uniform circuit family with size $O(S(n))$ and depth $O(T(n))$. We may assume that all NOT gates have been pushed back to the inputs and eliminated. Let p be a sequence of Ls and Rs, where L stands for left input and R stands for right input. Given two gates g and h in the circuit C_n, we say that $h = g(p)$ if h is the gate reached starting from g and following the sequence p. The ATM simulating C_n computes a function $cv(n, g, p)$, which evaluates the output of $g(p)$ by guessing $g(p)$, recursively evaluating $cv(n, g, p\mathrm{L})$ and $cv(n, g, p\mathrm{R})$, and combining the results appropriately, depending on the gate type of $g(p)$, to obtain $cv(n, g, p)$. When the length of p grows longer than $\log n$, g is replaced by $g(p)$ and p is truncated to ε. The procedure for computing $cv(n, g, p)$ is as follows.

```
PROCEDURE CV($n, g, p$)
(1) Guess $g(p) = h$ (using existential states);
(2) **In parallel** (using universal states)
      verify that $g(p) = h$;
      **if** $|p| = \lceil \log n \rceil$
      **then** return CV($n, h, \varepsilon$)
      **else** return CV($n, h, p$L)$*$CV($n, h, p$R), where $*$ is the type of gate $h$
```

The initial call to the procedure is CV(n, *outputgate*, ε). Because $|p|$ is always at most $\lceil \log n \rceil$ and our circuit family is logspace uniform, the step which verifies that $g(p) = h$ can be performed in logspace, and hence in ATM($\log n$, $\log^2 n$) [46]. Hence a logspace uniform circuit family of size $S(n)$ and depth $T(n) \geqslant \log^2 n$ can be simulated in ATIME(O($T(n)$)) using space O($\log S(n)$). In particular, $\mathrm{NC}^k \subseteq$ ATM($poly(n)$, $\log^k n$) for $k \geqslant 2$. For $k = 1$ a stronger notion of uniformity is required for the above inclusion to hold [203]; alternatively, following [43, 44] we can *define* NC^1 to be ATM($\log n$, $\log n$) (which is the same as ATIME($\log n$) or *alternating logtime*). Let $\mathrm{ATM}^k =$ ATM($\log n$, $\log^k n$). The two results outlined above relating ATM computations with uniform families of circuits establish that $\mathrm{ATM}^k = \mathrm{NC}^k$ and ATM($\log n$, $polylog(n)$) = NC.

Computation on ATMs can also be related to unbounded fan-in circuits. A computation on an ATM is said to be in ALT($S(n)$, $f(n)$) if the configurations require O($S(n)$) space, and if any path in an accepting computation DAG has at most $f(n)$ alternations between existential and universal states. The result

$$\mathrm{UCKT}(c^{S(n)}, D(n)) = \mathrm{ALT}(S(n), D(n))$$

is readily established by observing that the unbounded fan-in circuit can be converted into a bounded fan-in circuit with the same number of alternations between AND and OR gates, which can then be simulated in a manner similar to the bounded fan-in case; and conversely the full computation DAG of such a resource-bounded ATM has O($c^{S(n)}$) nodes, c a constant. Thus, defining ALT^k to be ALT($\log n$, $\log^k n$), we have

$$\mathrm{ALT}^k = \mathrm{AC}^k \quad \text{for } k \geqslant 1, \quad \text{and} \quad \mathrm{ALT}(\log n, polylog(n)) = \mathrm{NC}.$$

3.6. *Vector machines*

The last model of parallel computation that we consider is the *vector machine* [187, 214], which consists of a collection of bit processors together with a collection of registers that can hold bit vectors. All processors contain the same program whose instruction set consists of binary Boolean operations on the registers, complementation of the contents of a register, conditional jump on zero, the right or left shift of the contents of a register by a shift parameter specified in a register, and a mask instruction that inhibits some processors from executing the next instruction. Some of the registers are identified as *input* or *output* registers. The inputs to the computation are supplied in the input registers. At a given instant of time, the ith processor reads the ith bit of the operands (if any) specified in the current instruction, executes the prescribed instruction and writes the result in the ith position of the output vector. In the case of a shift operation, the processor writes in the appropriately shifted bit position; this is the means by which interprocessor communication takes place. When the computation terminates, the results of the computation are available in the output registers. A vector machine algorithm is logspace uniform if there is a deterministic Turing machine operating in space $\log n$ that, on input n, generates the program for inputs of length n. As in the case of PRAMs, in practice, we would expect vector machine code for a problem to be fixed, with n as a parameter.

Let VM($S(n)$, $T(n)$) be the class of problems that can be solved on a vector machine with $O(S(n))$ processors (and hence with vectors of length $O(S(n))$) in $O(T(n))$ time, and let VM^k be the class of problems that have logspace uniform vector machine algorithms in $\mathrm{VM}(poly(n), \log^k n)$. It is readily seen that

$$\mathrm{VM}(S(n), T(n)) \subseteq \mathrm{UCKT}(poly(S(n)), T(n))$$

since the instruction set of vector machines can be simulated in constant depth and polynomial size by unbounded fan-in circuits. It can also be shown that

$$\mathrm{CKT}(S(n), T(n)) \subseteq \mathrm{VM}(poly(S(n)), T(n))$$

provided $T(n) \geqslant \log^2 S(n)$ [100, 222]. Thus, for $k \geqslant 2$, we have

$$\mathrm{NC}^k \subseteq \mathrm{VM}^k \subseteq \mathrm{AC}^k \quad \text{and} \quad \mathrm{VM}(poly(n), polylog(n)) = \mathrm{NC}.$$

Several other parallel models and complexity results can be found in [25, 61, 62, 73, 74, 182, 184, 202, 222, 223, 233] (See also Chapter 1 on machine models and simulations in this Handbook).

3.7. *Randomized complexity classes*

In discussing randomized algorithms, we limit ourselves to problems defined in terms of a binary input–output relation $S(x, y)$. On input x, the task is to find a y satisfying $S(x, y)$, if such a y exists. A randomized algorithm will output one of the following three answers:

(a) a suitable value for y;

(b) an indication that no suitable y exists;

(c) an indication of failure, i.e., inability to determine if a suitable y exists or not.

We distinguish between zero-error algorithms (also known as Las Vegas algorithms) and algorithms with one-sided error (also known as Monte Carlo algorithms). If, on input x, $S(x, y)$ holds for some y, then the two types of algorithms act alike: each type produces a suitable y with probability greater than $\frac{1}{2}$, and otherwise reports failure. On an input x such that there is no y satisfying $S(x, y)$, the two types of algorithms behave differently. A zero-error algorithm reports "No suitable y exists" with probability greater than $\frac{1}{2}$, and otherwise reports failure, but a one-sided-error algorithm always reports failure.

Each of the complexity classes we have defined has its zero-error and one-sided-error randomized counterparts, indicated by the prefixes Z and R respectively. These are the classes of problems that have randomized algorithms with the corresponding hardware and time bounds. For randomized algorithms on PRAMs each processor is assumed to have the capability to generate random ($\log n$)-bit numbers; for randomized computation on circuits, an n-input circuit is allowed $poly(n)$ additional random input bits. Thus, for example, a problem is in ZCREW^k if it is solvable by a zero-error randomized algorithm that runs in time $O(\log^k n)$ using $poly(n)$ processors on a CREW PRAM in which each processor can generate random ($\log n$)-bit numbers, and a problem is in RNC^k if it is solvable with one-sided error by a logspace-uniform family of bounded fan-in circuits which receive, in addition to the problem input, $poly(n)$ random input bits.

3.8. *Arithmetic models*

For computation involving elements from an arbitrary domain D, it is convenient to assume that certain specified binary operations on the elements take unit time. This leads to the definition of *arithmetic PRAMs*, *arithmetic circuits* and *arithmetic Boolean circuits* (von zur Gathen [93]). An arithmetic PRAM is a regular PRAM which in addition can execute certain binary operations on elements over D in unit time. Sometimes D is specified not as a specific structure but according to axiomatic properties; for example, as an abstract ring. An arithmetic Boolean circuit is analogous to a Boolean circuit except that the set of gates is augmented to include gates that compute the specified arithmetic operations on elements over D in unit time. The conversion from arithmetic to Boolean values is performed by gates that test for zero, detect the sign, etc. In the reverse direction, Boolean selection circuits are used to select an output from among several arithmetic inputs. A special case of an arithmetic Boolean circuit is an *arithmetic circuit*, which has no Boolean gates. For *unbounded fan-in arithmetic Boolean circuits* we allow the Boolean gates to have unbounded fan-in while the arithmetic gates continue to have fan-in 2. The complexity classes *arithmetic* NC^k and *arithmetic* AC^k are analogues of NC^k and AC^k for arithmetic Boolean circuits and unbounded fan-in arithmetic Boolean circuits respectively. Similarly we have the analogous PRAM classes *arithmetic* EREW^k, *arithmetic* CREW^k, and *arithmetic* CRCW^k. It should be noted that other authors (cf. [93]) define arithmetic NC^k in terms of arithmetic circuits (without Boolean gates) with a further restriction to polynomials or rational functions of polynomial-bounded degree. This restriction avoids certain trivial separations between complexity classes; for instance, without it, the function $x^{2^{\log^2 n}}$ would lie in arithmetic NC^2 but not arithmetic NC^1.

When dealing with arithmetic computation involving addition, subtraction and multiplication of numbers represented in binary, there is an obvious conversion from arithmetic circuits in Boolean ones that maintains size to within a polynomial, and increases depth by a factor of $\log\log s$, where s is a bound on the size of the arithmetic operands (and thus the number of bits needed to represent any operand is no more than $\lceil \log s \rceil$). The bounds on size and depth follow from well-known poly-size log-depth circuits for addition, subtraction and multiplication of two n-bit numbers (see Section 4.2).

3.9. *Parallel Computation Thesis*

Finally, an important connection between sequential and parallel computation is highlighted in the *Parallel Computation Thesis* [38, 46, 85, 106, 187]. Let us say that two functions are polynomially equivalent if each is bounded above by a polynomial function of the other. The Parallel Computation Thesis states that parallel time is polynomially equivalent to sequential space. This relationship has been established in many forms. Parallel time on a vector machine is polynomially equivalent to sequential space [187, 214]. An ATM can be viewed as a parallel machine, and the result follows since alternating time is polynomially equivalent to sequential space [46]. Computation on a PRAM can be simulated by a Turing machine with space polynomially bounded in the parallel time, and conversely, provided the number of processors is no more than an exponential in the parallel time [85, 106]. Finally any computation in nondeterministic space $S(n)$ can be simulated by a circuit of depth $S(n)^2$ and any circuit of depth $D(n)$ can be simulated in deterministic space $D(n)$ [38]. We conclude this section by illustrating the Parallel Computation Thesis for some of the models we have considered.

We first relate nondeterministic space to parallel time. Consider the computation of any nondeterministic $S(n)$ space-bounded Turing machine M. Given an input x to M, we can formulate the acceptance problem as a reachability problem on a directed graph G, whose vertices are the configurations of M, and for which there is an arc from vertex u to vertex v if and only if the configuration presented by v can be reached in a single step from the configuration represented by u. There is also a dummy vertex z with an arc into it from every vertex corresponding to an accepting configuration. If s is the vertex corresponding to the initial configuration of M on input x, then M accepts x if and only if z is reachable from s in G. The size of G is $O(c^{S(n)})$ for some constant c, since the number of different configurations of an $S(n)$ space bounded Turing machine is no more than an exponential in $S(n)$. The reachability problem can be solved in AC^1 (see Section 4) by finding the transitive closure of the adjacency matrix of G. Since this construction is logspace uniform, it follows that any computation in nondeterministic space $S(n)$ is in $CKT(c^{S(n)}, S^2(n))$ and in $UCKT(c^{S(n)}, S(n))$, and also in $CRCW(c^{S(n)}, S(n))$ and $EREW(c^{S(n)}, S^2(n))$. Thus, in these parallel models, nondeterministic space-bounded computation can be parallelized to run in time at most the square of the space bound, and using hardware at most exponential in the space bound.

A similar technique shows that deterministic space $S(n)$ is in $EREW(c^{S(n)}, S(n))$. In this case the computation of the space-bounded Turing machine can be modeled as a directed tree, and an input is accepted if and only if the final configuration is reachable from the initial one. Since reachability in a directed tree is a special case of expression evaluation, the tree contraction algorithm of Section 2.2 gives the required result.

For the reverse, consider a circuit in $CKT(S(n), D(n))$. Given a description of this circuit together with an input to it, a deterministic Turing machine can compute its output in $O(D(n))$ space by starting at the output and working its way back to the input, while using a stack to keep track of the path taken (using L for left input and R for right input), first testing left inputs and then the right ones. Since the depth of the circuit is $D(n)$, the stack has at most $D(n)$ entries, each of which is a constant. Hence the value computed by the circuit can be evaluated in $O(D(n))$ space. By using the results we described earlier in this section relating parallel time on various models to depth on a corresponding uniform family of bounded fan-in circuits, it is easy to verify that a parallel algorithm on all of the models we have considered implies a deterministic sequential algorithm that uses space at most the square of the parallel time.

4. NC-algorithms and P-complete problems

4.1. Introduction

In this section we locate several important problems within the hierarchy $\{NC^k\}$. We restrict ourselves to problems that are of central importance because they can be used as subroutines in the solution of a wide range of other problems. Among these are the basic arithmetic operations, Boolean matrix multiplication and transitive closure, the computation of the inverse and the rank of a matrix, the evaluation of straight-line programs, and the computation of a maximal independent set in a graph. We also introduce the concept of P-completeness, state the evidence that the P-complete problems are unlikely to lie in NC, and give several examples of P-complete problems.

The algorithms we give are not necessarily efficient, in the sense of Section 2, since their processor–time product may be asymptotically much larger than the execution time of the best sequential algorithm for the same problem. In this sense, our level of aspiration is lower than in Section 2, where we concerned ourselves with efficient and optimal algorithms.

4.2. *NC-circuits for arithmetic operations*

This section is concerned with Boolean circuits for addition, subtraction, multiplication and division of integers. We shall show that addition and subtraction are in AC^0, that multiplication is in NC^1 but not AC^0, and that division is realizable by a family of bounded fan-in circuits that is of logarithmic depth and polynomial size, but does not appear to be logspace uniform. Further information on circuits for arithmetic operations can be found in [9, 185, 204].

4.2.1. *Addition and subtraction*

The addition problem takes as input two n-bit binary numbers and produces as output their $(n+1)$-bit sum. We represent numbers as tuples in binary notation. Let the input numbers be $(x_{n-1}, x_{n-2}, \ldots, x_0)$ and $(y_{n-1}, y_{n-2}, \ldots, y_0)$, and let the output be $(z_n, z_{n-1}, \ldots, z_0)$. Let c_j denote the carry out of the jth position. Let $g_j = x_j y_j$ and $p_j = x_j \vee y_j$; g_j is called the jth *carry generate bit* and p_j is called the jth *carry propagate bit*. Then, letting $\oplus$ represent "exclusive or" and letting $c_{-1} = 0$, we have the recurrences $c_j = g_j \vee p_j c_{j-1}$ and $z_j = x_j \oplus y_j \oplus c_{j-1}$ for $j = 0, 1, \ldots, n$. It follows that $c_j = \bigvee_{i \leqslant j} g_i p_{i+1} \cdots p_j$. These formulas yield a polynomial-size constant-depth circuit for addition:

Stage 1: compute all carry generate bits g_j and carry propagate bits p_j.
Stage 2: compute all products of the form $g_i p_{i+1} \cdots p_j$.
Stage 3: compute all c_j as the OR of products computed in Stage 2.
Stage 4: compute each output bit z_j from x_j, y_j and c_{j-1}.

This establishes that addition is in AC^0, and hence, also in NC^1.

A more economical NC^1-circuit for addition can be obtained using the prefix sums algorithm of Section 2.2. That algorithm takes as input an array $(a_1, a_2, \ldots, a_n)$ and produces the array $(a_1, a_1 * a_2, \ldots, a_1 * a_2 * \cdots * a_n)$, where $*$ is an associative binary operation. Since the memory accesses are data independent in the PRAM algorithm for prefix sums in Section 2.2, that algorithm can also be represented as a circuit of depth $O(\log n)$ and size $O(n)$, with gates that compute $*$. To apply this construction to addition, let T_j be the transformation that computes carry bit c_j from carry bit c_{j-1}. This affine Boolean transformation is specified by the equation $c_j = g_j \vee p_j c_{j-1}$. Then c_j

is obtained by evaluating the transformation $T_j * T_{j-1} * \cdots * T_0$ at the point 0; here the associative operation $*$ is the composition of affine Boolean transformations. Using a parallel prefix circuit to compute the necessary compositions, one obtains a bounded fan-in Boolean circuit of size $O(n)$ and depth $O(\log n)$ which computes the carry bits c_j, and then the output bits z_j. A variant of this construction gives bounded fan-in circuits of linear size and logarithmic depth for subtraction in either the 1's complement or 2's complement representation.

In [45] it is shown that, if the semigroup defined by $*$ does not contain a nontrivial group as a subset then, for any strictly increasing primitive recursive function f, the prefix sums problem can be solved by an unbounded fan-in circuit family of constant depth and size $O(nf^{-1}(n))$. Each gate in such a circuit computes the semigroup product of its inputs. Dolev, Dwork, Pippenger & Wigderson [70] show that addition cannot be performed by unbounded fan-in circuits of constant depth and linear size.

4.2.2. *Multiplication*

The multiplication problem takes as input two n-bit binary numbers and produces as output their $2n$-bit product. Let the inputs be $(x_{n-1}, x_{n-2}, \ldots, x_0)$ and $(y_{n-1}, y_{n-2}, \ldots, y_0)$, and let the output be $(z_{2n-1}, z_{2n-2}, \ldots, z_0)$. The standard shift-and-add algorithm for multiplication can be implemented in the time required to add n binary numbers, each of length $2n$. This latter problem can be solved by a bounded fan-in circuit of depth $O(\log n)$ and polynomial size using the following *three-for-two trick* which reduces the problem of adding three n-bit numbers to the problem of adding two $(n+1)$-bit numbers [179, 240]. Let $a=(a_{n-1}, a_{n-2}, \ldots, a_0)$, $b=(b_{n-1}, b_{n-2}, \ldots, b_0)$ and $c=(c_{n-1}, c_{n-2}, \ldots, c_0)$ be the three n-bit numbers to be added. If we add the three bits a_i, b_i and c_i for $i=0, 1, \ldots, n-1$, we will obtain for each i a two-bit number whose upper and lower bits we shall denote by u_i and v_i. The u_i and v_i can be generated by a linear-size constant-depth circuit that, for each i, adds the three 1-bit numbers a_i, b_i and c_i. Then $a+b+c=u+v$, where $u=(u_{n-1}, u_{n-2}, \ldots, u_0, 0)$ and $v=(v_{n-1}, v_{n-2}, \ldots, v_0)$. The addition of n numbers, each of length $2n$, can be achieved by applying $O(\log n)$ iterations of the three-for-two trick, each of which reduces the number of numbers by a factor of $\frac{2}{3}$, followed by a final stage of adding two $O(n)$-bit numbers. This establishes that multiplication is in NC^1.

The best bounded fan-in circuit known for multiplication achieves $O(\log n)$ depth with $O(n \log n \log\log n)$ size [208]. The construction is logspace-uniform; it is based on circuits for computing the Discrete Fourier Transform over certain finite rings.

The paper by Furst, Saxe & Sipser [86] was the first to establish that multiplication is not in AC^0. This was done by showing that the n-input parity function, which is equal to 1 if and only if an odd number of its inputs are 1, does not lie in AC^0, and then showing that if multiplication were in AC^0, then parity would also lie in AC^0. These results are presented in [37]. As mentioned in Section 3, parity, and hence multiplication, requires unbounded fan-in circuits of depth $\Omega(\log n/\log\log n)$ if the size is to be polynomial. The circuit of Schönhage & Strassen [208] can be converted into an $O(\log n/\log\log n)$ depth unbounded fan-in circuit for multiplication of size $O(n^{1+\varepsilon})$ for any $\varepsilon > 0$, using a standard technique of compressing $O(\log\log n)$ levels of a bounded fan-in circuit [47].

4.2.3. *Division*

The input to the division problem is a pair of n-bit binary strings representing integers x and y with $y \neq 0$. The output is the n-bit binary representation of the integer part of x/y. We present a simplified version of a construction due to Beame, Cook & Hoover [28] which yields a family of bounded fan-in division circuits of polynomial size and logarithmic depth. The construction appears not to be logspace-uniform, and thus does not establish that division lies in NC^1.

To describe the construction we require the concept of NC^1-reducibility. Let A and B be functions from $\{0,1\}^*$ into $\{0,1\}^*$, each having the property that the length of the output is determined by the length of the input. For each positive integer n we may derive from B a function B_n from $\{0,1\}^n$ into $\{0,1\}^m$, where m is the length of the string $B(x)$ whenever the string x is of length n. Similarly, we may derive from A a family of functions A_n. The function A is said to be NC^1-reducible to B (denoted $A <_{\mathrm{NC}^1} B$) if the family of functions $\{A_n\}$ can be realized by a logspace-uniform family of bounded fan-in circuits of polynomial size and logarithmic depth, together with "oracle gates" realizing functions in the family $\{B_n\}$, subject to the restriction that no two oracle gates lie on the same input-output path. It follows from this definition that if $A <_{\mathrm{NC}^1} B$ and B is realizable by a bounded fan-in circuit family of polynomial size and logarithmic depth, then A is also realizable by such a circuit family; moreover, if the circuit family for B is logspace-uniform, then the circuit family for A will also be logspace-uniform.

We shall show that

$$\text{DIVISION} <_{\mathrm{NC}^1} \text{RECIPROCAL} <_{\mathrm{NC}^1} \text{POWERING} <_{\mathrm{NC}^1} \text{ITERATED PRODUCT},$$

where the reciprocal, powering and iterated product problems are defined as follows.

RECIPROCAL

Input: A non-zero n-bit integer y.

Output: An n-bit binary function $\tilde{y}^{-1}$ such that $y^{-1} - 2^{-n} \leqslant \tilde{y}^{-1} \leqslant y^{-1}$.

POWERING

Input: An n-bit integer x and an integer i, where $1 \leqslant i \leqslant n$.

Output: x^i expressed as an n^2-bit integer.

ITERATED PRODUCT

Input: n-bit binary integers $w_1, w_2, \ldots, w_n$.

Output: The n^2-bit product $w_1 w_2 \cdots w_n$.

The reductions are as follows.

DIVISION $<_{\mathrm{NC}^1}$ RECIPROCAL. It suffices to treat the case $y > 0$:

```
(1) using an oracle gate for RECIPROCAL, compute $\tilde{y}^{-1}$;
(2) using an NC¹-circuit, compute the product $x\tilde{y}^{-1}$;
(3) if $(x\tilde{y}^{-1})y \leqslant x$
    then $\lfloor xy^{-1} \rfloor = x\tilde{y}^{-1}$
    else $\lfloor xy^{-1} \rfloor = x\tilde{y}^{-1} - 1$.
```

RECIPROCAL $<_{\mathrm{NC}^1}$ POWERING

(1) let $y=2^j w$, where $\frac{1}{2}\leqslant w<1$, and let $t=1-w$;
(2) **in parallel**, using oracle gates for powering, compute $t^2, t^3, \ldots, t^n$;
(3) using an NC^1-circuit for iterated addition, compute $1+t+t^2+\cdots+t^n$;
(4) $\tilde{y}^{-1}=2^{-j}(1+t+t^2+\cdots+t^n)$, rounded to n-bit precision.

POWERING $<_{\mathrm{NC}^1}$ ITERATED PRODUCT. This is immediate, since powering is a special case of iterated product.

Given these reductions, the task remaining is to give a polynomial-size logarithmic-depth bounded fan-in circuit for iterated product. The method used by Beame, Cook & Hoover [28] for constructing this circuit depends on the following ancient theorem.

CHINESE REMAINDER THEOREM. *Let $c_1, c_2, \ldots, c_m$ be pairwise relatively prime and let $c=\Pi_{i=1}^m c_i$. Then*

(i) *there exist integers $v_1, v_2, \ldots, v_m$ such that, for $i=1,2,\ldots,m$ and $j=1,2,\ldots,m$,*

$$v_i \bmod c_j = \begin{cases} 1 & \textit{if} \quad i=j, \\ 0 & \textit{if} \quad i\neq j; \end{cases}$$

(ii) *for any integer u, $u \bmod c=\Sigma_{i=1}^m u_i v_i \bmod c$, where $u_i=u \bmod c_i$.*

The circuit for computing the product $w_1 w_w \cdots w_n$, where the w_i are n-bit integers, is as follows:

(i) Let $c_1, c_2, \ldots, c_m$ be distinct primes less than n^2 whose product is greater than 2^{n^2} (for all $n>5$ such a set of primes exists). Let $c=\Pi_{j=1}^m c_j$.
(ii) For $i=1,2,\ldots,n$ and $j=1,2,\ldots,m$ compute $w_i \bmod c_j$.
(iii) For $j=1,2,\ldots,n$ compute $w_1 w_2 \cdots w_n \bmod c_j$.
(iv) Using the Chinese Remainder Theorem, compute $w_1 w_2 \cdots w_n \bmod c$.

Each of the steps is carried out by a polynomial-size, logarithmic-depth circuit which has access to certain precomputed constants. We illustrate by describing step (iii), in which the essential task is to multiply together n elements x_i, each of which is a residue modulo a prime p which is less than n^2 (here p corresponds to c_j, and x_i to $w_i \bmod c_j$). Let $\mathbf{Z}_p^*$ be the field of non-zero residues modulo p. Then $\mathbf{Z}_p^*$ has a *primitive root*: i.e., an element g such that every element of $\mathbf{Z}_p^*$ is a power of g. If $x=g^y$ then y is called the (discrete) logarithm of x, and x is the antilogarithm of y. With the help of circuits of depth O(log n) for computing logarithms and antilogarithms, we can replace iterated product mod p by iterated summation mod$(p-1)$. The circuit for computing $x_1 x_2 \cdots x_n$ mod p has the following parts:

(i) For $i=1,2,\ldots,n$ compute y_i, the logarithm of x_i;
(ii) Using an NC^1-circuit for iterated addition, compute $y=y_1+y_2+\cdots+y_n$ mod$(p-1)$.
(iii) Compute x, the antilogarithm of y. Then $x=x_1 x_2 \cdots x_n \bmod p$.

This completes our description of a polynomial-size, logarithmic-depth bounded fan-in circuit for division. Because of the precomputed constants $v_1, v_2, \ldots, v_m$ and $\Pi_{j=1}^m c_j$, the circuit appears not to be logspace-uniform, and it remains an open question whether DIVISION is in NC^1. Reif [197] has shown that division is "almost" in NC^1 by giving a logspace-uniform family of division circuits of polynomial size and depth $O(\log n \log\log n)$. Shankar & Ramachandran [211] refined the constructions of Beame, Cook & Hoover and Reif [28, 197] by showing that in each case the size can be reduced to $O(n^{1+\delta})$, where δ is an arbitrarily small positive constant. Reif & Tate [199] improve upon the result of Shankar & Ramachandran by giving logspace-uniform division circuits having size $O(n \log n \log\log n)$ and depth $O(\log n \log\log n)$.

4.3. *Circuits for expression evaluation*

In Section 2.2.3 we described a tree contraction algorithm and used it to derive a simple, optimal $O(\log n)$ time EREW PRAM algorithm for the expression evaluation problem. Since the expression evaluation problem is an important one, considerable work has been done on solving the problem efficiently for a different model of parallel computation, the arithmetic network.

The problem has both a static version and a dynamic version. In the static version we are given an n-variable expression over an algebraic structure, and our task is to construct an arithmetic circuit of small size and depth for that particular expression. The inputs to the circuit will be the values of the n variables in the expression, and the output will be the value of the expression. In the more general dynamic version we wish to construct an arithmetic Boolean circuit that takes as inputs an n-variable arithmetic expression, presented as a well-formed string of operators, operands and left and right parentheses, together with the values of the variables, and produces the value of the expression.

The early circuits for static expression evaluation are based on methods of decomposing a binary tree into two subtrees of approximately equal size by deleting a single edge. Let T be a tree representing an expression on n variables over a ring. Then T has $m = 2n-1$ vertices. It is easy to see that T has an edge e whose removal breaks it into two subtrees, T_1 and T_2, each of which has no more than $2n/3$ vertices. Let e be directed from node u to its parent v. Then the subtree rooted at u is one of the two operands for the operator at v. Let T_1 be the subtree containing v and T_2, the subtree containing u. Then we can write the value of the expression computed by T_1 as $Ax+B$, where x is the value of u, which is also the value of the expression computed by T_2. The method of Brent [42] recursively computes the coefficients A and B for T_1 and the value x for T_2, and, using a constant-size circuit, combines the outputs of the two subtrees to obtain $Ax+B$, the value of the expression. This leads to a log-depth linear-size arithmetic circuit for the static expression evaluation problem. The tree contraction algorithm of Section 2.2.3 gives another optimal $O(\log n)$ depth circuit for the same problem. Both of these methods are easily extended to handle division.

Brent's method leads to an NC^2 arithmetic Boolean circuit for the dynamic expression evaluation problem since, at each recursive stage, the separating edge for each subtree at that level of recursion can be found in log-depth by a fairly easy construction. Similarly, since $EREW^1 \subseteq NC^2$, the tree contraction algorithm gives another NC^2 arithmetic Boolean circuit for the dynamic expression evaluation problem.

In [43, 44] an NC^1 arithmetic Boolean circuit is given for the problem. As in Brent [42], the method is based on recursively computing the value of the expression by decomposing the tree into subtrees; a factor of $\log n$ in depth is saved by computing all the decompositions simultaneously, instead of doing it separately at each of the $\log n$ recursive stages. The algorithm first transforms the expression into a postfix expression with the property that, for each operator, the length of the second operand is no greater than that of the first. This expression E is then recursively segmented into three equal-sized, overlapping strings of half its length, consisting of the first half of E, the middle half of E and the last half of E. Each of these strings represents a collection of subtrees of E, and the algorithm chooses the roots of the first two subtrees in each string to be the positions at which the expression is to be decomposed. It can be shown that this method of decomposition leads to a log-depth recursive algorithm for evaluating E. The method has the advantage that the positions where decomposition takes place at all levels of recursion can be determined in logarithmic depth by suitably parsing the string representing the expression E, and this leads to an NC^1 arithmetic Boolean circuit for the dynamic expression evaluation problem.

The NC^1 circuits for static and dynamic expression evaluation work for expressions over a field, ring or semiring. In particular, the Boolean expression evaluation problem, in which each input value is 0 or 1, is complete for NC^1 under deterministic log-time reductions, and can be considered, in some sense, to be the canonical problem for NC^1 [43].

4.4. *Boolean matrix multiplication and transitive closure*

We show that the Boolean matrix multiplication problem is in AC^0 and the transitive closure problem is in AC^1. Let $A=(a_{ij})$ and $B=(b_{ij})$ be $n\times n$ matrices of zeros and ones. Then the (Boolean) product of A and B is the $n\times n$ matrix $C=(c_{ij})$, where $c_{ij}=\bigvee_{k=1}^{n} a_{ik}\cdot b_{kj}$. The Boolean product of A and B can be computed by a two-level unbounded fan-in circuit. At the first level there are n^3 AND-gates, each of which computes one of the products $a_{ik}\cdot b_{kj}$. At the second level there are n^2 n-input OR-gates, each of which computes $\bigvee_{k=1}^{n} a_{ik}\cdot b_{kj}$ for a specific i and j. The specification of these circuits is clearly logspace-uniform. Thus Boolean matrix multiplication is in AC^0, and hence in NC^1. By imbedding the problem in an appropriate ring and applying fast matrix multiplication, the number of gates in the bounded fan-in circuit can be reduced asymptotically to $\mathrm{O}(M(n))$, where $M(n)$ is the number of processors required by an arithmetic PRAM to perform $n\times n$ matrix multiplication over a general ring in time $\mathrm{O}(\log n)$. It is known that $M(n)=\mathrm{O}(n^{2.376})$ (Coppersmith & Winograd [66]).

Let $A=(a_{ij})$ be an $n\times n$ matrix of zeros and ones. Let I denote the $n\times n$ identity matrix. For $k=0, 1, 2, \wedge$ let A^k, the kth power of A, be defined inductively as follows: $A^0=I$; for $k=1, 2, \ldots, A^k=A^{k-1}\cdot A$. Define A^*, the reflexive transitive closure of A as follows: $A^*=\bigvee_{k=0}^{\infty} A^k$. It is not difficult to show that $A^*=(I\vee A)^{2^{\lceil\log_2 n\rceil}}$. Hence A^* can be computed by initializing the matrix B to the value $I\vee A$, and successively squaring the matrix B $\lceil\log_2 n\rceil$ times. Since matrix multiplication is in AC^0, it follows that transitive closure lies in AC^1, and hence in NC^2.

4.5. *Matrix computations*

In discussing matrix computations, we work with arithmetic circuits as defined in Section 3: acyclic circuits in which the inputs are elements of a field **F** and the gates perform the four field operations $+$, $-$, $*$ and $/$. It is clear from its defining formula that matrix multiplication is in arithmetic NC^1. Similarly, the problem of computing A^n is in arithmetic NC^2, since it is possible to compute A^n in at most $2\lceil\log_2 n\rceil$ matrix multiplications by first computing $A^2, A^4, A^8, \ldots$ via repeated squaring, and then multiplying together appropriate matrices of the form A^{2^k}.

The problem of computing the inverse of a lower triangular matrix illustrates a divide-and-conquer technique that is often used for the construction of parallel algorithms. Let A be a square nonsingular lower triangular matrix whose number of rows is a power of 2 (square matrices of other dimensions can easily be handled by adding additional rows and columns, whose entries are 0 except for 1s on the diagonal, to pad the dimension out to a power of 2). Then we can block-decompose A into four matrices of half its dimension, as follows:

$$A = \left[\begin{array}{c|c} A_{11} & 0 \\ \hline A_{21} & A_{22} \end{array}\right]$$

Because A is lower triangular, it follows that the upper right-hand submatrix is the zero matrix, and that A_{11} and A_{22} are lower triangular. The inverse of A is given by

$$A^{-1} = \left(\begin{array}{c|c} A_{11}^{-1} & 0 \\ \hline -A_{22}^{-1}A_{21}A_{11}^{-1} & A_{22}^{-1} \end{array}\right)$$

This formula for the inverse suggests a recursive parallel algorithm for computing A^{-1}. First, A_{11}^{-1} and A_{22}^{-1} are computed recursively in parallel. Then $-A_{22}^{-1}A_{21}A_{11}^{-1}$ is computed via two matrix multiplications. This recursive algorithm leads to a uniform family of arithmetic circuits with $O(M(n))$ gates and depth $O(\log^2 n)$; hence the problem of inverting a lower triangular matrix lies in arithmetic NC^2.

4.5.1. *Computing the determinant*

We next turn to the problem of computing the determinant of an $n \times n$ matrix. This problem presents an interesting challenge because Gaussian elimination, the standard sequential algorithm for this problem, does not lead to an NC-algorithm. Gaussian elimination computes the determinant in a series of n stages, each of which transforms the given matrix without changing its determinant. Each stage requires $O(n^2)$ operations which can be performed in parallel, but it appears that the stages must be performed in sequence, so that the running time cannot be reduced below $\Theta(n)$.

The characteristic polynomial of a square matrix A is the nth-degree polynomial $\det(A - xI)$. In 1976 Csanky [67] showed that, for matrices over a field of characteristic zero, the problem of computing the characteristic polynomial lies in arithmetic NC^2. The result was extended to matrices over an arbitrary ring in [40] (see also [34]) and in [49]. Let the characteristic polynomial be $\Sigma_{i=0}^{n} c_i x^i$. Then $\det A = c_0$; and since, by the

Cayley–Hamilton Theorem, a matrix satisfies its own characteristic equation,

$$A^{-1}=\frac{-(c_1+c_2A+\cdots+c_nA^{n-1})}{c_0}.$$

This equation yields an arithmetic NC^2-algorithm for computing A^{-1} and det A from the characteristic polynomial.

We present Chistov's algorithm [49] for computing the characteristic polynomial. The following observation will be required. Let B be an $n\times n$ matrix. Let B_k be the $k\times k$ matrix in the lower right-hand corner of B; i.e., $(B_k)_{ij}=(B)_{n-k+i,n-k+j}$. Assume that, for $k=1,2,\ldots,n$, B_k is nonsingular. Then

$$\frac{1}{\det B}=\prod_{k=1}^{n}(B_k^{-1})_{11}.$$

Chistov's algorithm computes the polynomial $\det(I-xA)$ in time $O(\log^2 n)$ using $n^2M(n)$ processors. The coefficients of the polynomial $\det(I-xA)$ are those of the characteristic polynomial, but in reverse order and, if n is odd, with their signs reversed. The algorithm performs computations on power series in x; however, since the final result of the computation is a polynomial of degree n, these power series can be computed modulo n^{n+1}; i.e., they can be truncated to polynomials of degree n. We will use the fact that if $A(x)$ is a square matrix in which each element is a power series in x with constant term zero, then

$$(I-A(x))^{-1}\bmod x^{n+1}=\sum_{i=0}^{n}A(x)^i \bmod x^{n+1}.$$

By using repeated squaring, the powers of $A(x)$ up to the nth can be computed in time $O(\log^2 n)$ using $O(nM(n))$ processors, and hence $(I-A(x))^{-1}\bmod x^{n+1}$ can be computed in time $O(\log^2 n)$ using $O(nM(n))$ processors.

Our goal is to compute det B, where $B=I-xA$. Henceforth, all the objects computed are power series modulo x^{n+1} or matrices whose elements are such power series. For $k=1,2,\ldots,n$, B_k is a nonsingular matrix of the form I_k-xA_k, where I_k is the $k\times k$ identity matrix. Then $B_k^{-1}=(I_k-xA_k)^{-1}$; and, modulo x^{n+1}, this quantity is equal to $\Sigma_{i=0}^{n}(xA_k)^i$.

Chistov's algorithm proceeds as follows:

(i) for $k=1,2,\ldots,n$ compute B_k^{-1} and extract the element $(B_k^{-1})_{11}$;

(ii) multiply together the n elements extracted in step (i); the resulting product is $1/\det B$;

(iii) compute det B by taking the inverse of the power series obtained in step (ii).

Chistov's algorithm requires $O(n^2M(n))$ processors. For fields of characteristic zero, a variant of Csanky's algorithm due to Preparata & Sarwate [188] computes the characteristic polynomial in time $O(\log^2 n)$ using $O(n^{3.5})$ processors. Even this algorithm is not efficient, since, over a field of characteristic zero, the characteristic polynomial can be calculated sequentially in time $O(n^{2.376})$, where the constant implied by the "big O" is enormous, or, more practically, in $O(n^3)$ steps, where the implied constant is small.

Borodin, Cook & Pippenger [39] have shown that, for each fixed k, the following problem is in NC^2: compute the determinant of a matrix A whose entries are rational functions of the k variables $x_1, x_2, \ldots, x_k$, where each entry is presented as a pair of polynomials of degree at most n in which each coefficient is an n-bit integer. The same problem, but with the coefficients taken from an arbitrary field, is in arithmetic NC^2.

4.5.2. Computing the rank

The problem of computing the rank of a matrix A with elements in a field $\mathbf{F}$ has been treated in [121, 40, 49, 174]. If A is diagonalizable then its rank can be read off from the characteristic polynomial: it is simply $n-m$, where x^m is the highest power of x that divides the characteristic polynomial. If A is not diagonalizable, however, then this rule does not correctly give the rank. For example, the matrix $\left[\begin{smallmatrix}0&1\\0&0\end{smallmatrix}\right]$ has rank 1, but its characteristic polynomial is x^2. In [121], it is shown that, if $\mathbf{F}$ is a subfield of the reals, then $\operatorname{rank}(AA^{\mathrm{T}}) = \operatorname{rank}(A)$ and the matrix AA^{T} is diagonalizable. Hence, in this case, the rank may be read off from the characteristic polynomial.

In [174] it is shown that the problem of computing the rank of matrix A over an arbitrary field $\mathbf{F}$ is in arithmetic NC^2. Mulmuley's algorithm is remarkably simple. Assume that A is square and symmetric; for, if not, one may work instead with the square, symmetric matrix

$$\begin{bmatrix} 0 & A \\ A^{\mathrm{T}} & 0 \end{bmatrix}$$

whose rank is twice the rank of A. Let Z be a diagonal matrix in an indeterminate z such that $Z_{ii} = z^{i-1}$. Using the algorithm of Borodin, Cook & Pippenger [39], compute $Q(x) = \det(xI - ZA)$, the characteristic polynomial of the matrix ZA. Then the rank of A is $n-m$, where x^m is the highest power of x that divides the characteristic polynomial $Q(x)$.

4.6. Dynamic evaluation of straight-line code

A commutative semiring $(R, +, *, 0, 1)$ is an arithmetic structure with domain R and two commutative, associative binary operations $+$ and $*$, such that 0 is an additive identity, 1 is a multiplicative identity, and the distributive law $a*(b+c) = a*b + a*c$ holds. In conformity with the definition of Section 3.8, an arithmetic circuit over this semiring is an acyclic connection of input nodes of in-degree 0 and addition and multiplication nodes of in-degree 2, together with an assignment to each input node of a value from the domain R. The execution of the indicated multiplication and addition operations assigns a value from R to each node of the circuit, and the problem of computing these values is known as the evaluation problem. It is equivalent to the problem of evaluating straight-line programs with operations from a commutative semiring.

The performance of algorithms for this problem is stated in terms of two parameters: n, the size of the circuit C, and d, the (formal algebraic) degree of C. Each node in C can be viewed as computing a polynomial expression in the inputs, and the degree of C is the maximum of the formal degrees of these expressions. More precisely, the degrees of the nodes are defined inductively: the degree of an input node is 1, the degree of an addition node is the larger of the degrees of its two inputs, and the degree of a multiplication node is the sum of the degrees of its two inputs. The paper by Valiant, Skyum, Berkowitz & Rackoff [231] gives a method of converting any arithmetic circuit of size n and degree d into one of size $poly(n)$ and depth $O(\log n \cdot \log d)$. The paper by Miller, Ramachandran & Kaltofen [170] considers the more general dynamic version of the problem, in which the input to the evaluation algorithm consists of the arithmetic circuit C and the values of its inputs; thus no preprocessing based on C alone is allowed. The algorithm of [170] runs in time $O(\log n \log(nd))$ using $M'(n)$ processors, where $M'(n)$ is the number of processors required by an arithmetic PRAM to perform $n \times n$ matrix multiplication over a general semiring in time $O(\log n)$. It is evident that $M'(n) = O(n^3)$.

In order to describe the algorithm of Miller, Ramachandran & Kaltofen, we require the concept of a weighted arithmetic circuit over the semiring R. Such a circuit is a directed acyclic graph in which each edge (u, v) has a weight $W(u, v)$ which is an element of R. The nodes are of three types: leaves, multiplication nodes and addition nodes. A leaf has in-degree 0, a multiplication node has in-degree 2, and an addition node has in-degree greater than 0. Associated with each leaf is a value in R. It is required that no edge be directed from a multiplication node to a multiplication node.

Associated with each node v in a weighted arithmetic circuit is a value $VAL(v)$. The values of the leaves are given as part of the specification. If an addition node v has edges directed into it from nodes $v_1, v_2, \ldots, v_h$, then

$$VAL(v) = \sum_{i=1}^{h} VAL(v_i) * W(v_i, v).$$

If a multiplication node has edges directed into it from nodes v_1 and v_2 then

$$VAL(v) = VAL(v_1) * W(v_1, v) * VAL(v_2) * W(v_2, v).$$

We see that an ordinary arithmetic circuit over R can be viewed as a weighted arithmetic circuit in which every weight is equal to 1. The requirement that no edge runs from one multiplication node to another is easily met by inserting extra addition nodes of in-degree 1. Starting with the given arithmetic circuit, the algorithm constructs a sequence of weighted arithmetic circuits, all of which have the same set of nodes. Moreover, the iteration that produces each weighted arithmetic circuit from its predecessor in the sequence preserves the values of all nodes.

The iteration is accomplished in three steps:

(1) (*MM*): compress subcircuits consisting of additions only; the nature of this operation is indicated in Fig. 4;

(2) (*Rake*): simultaneously evaluate every node for which all direct predecessors are leaves, and delete the edges directed into those nodes;

(3) (*Shunt*): simultaneously bypass every multiplication node having a leaf as a direct predecessor; the nature of this operation is indicated in Fig. 5.

The cost of executing this iteration is dominated by the first step, which can be

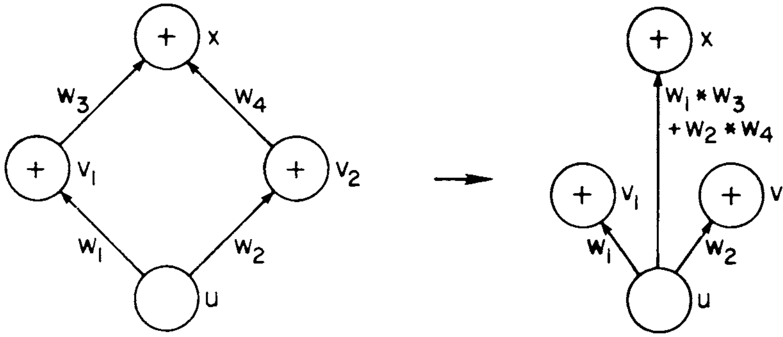

Fig. 4. The MM operation.

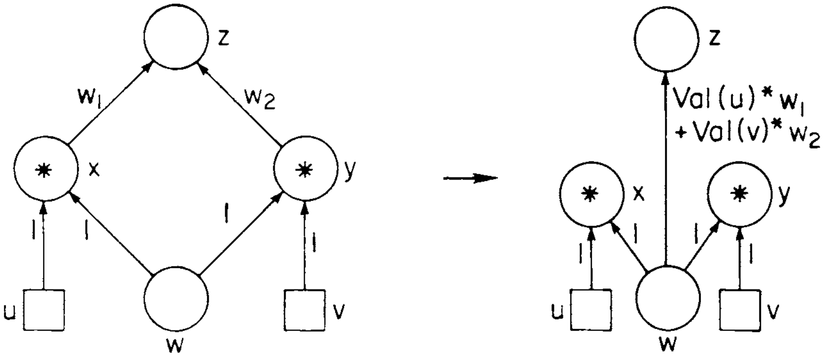

Fig. 5. The Shunt operation.

organized as a matrix multiplication. Thus each iteration can be performed in time O(log n) using $M'(n)$ processors. It can be shown that all nodes get evaluated within O(log(nd)) iterations of this transformation.

The paper by Miller & Teng [172] extends the result of [170] to a wider class of algebraic structures, and Kaltofen [126] gives a randomized algorithm for the static problem in the case where division is allowed.

4.7. *The maximal independent set problem*

A set of vertices S in a graph G is called *independent* if no vertices in S are adjacent, and *maximal independent* if it is independent and not properly contained in any independent set. Noting that the most obvious methods of creating a maximal independent set do not parallelize, the paper [230] suggested the possibility that the problem of constructing such a set might be inherently resistant to solution in parallel. This was shown to be false in 1985, when Karp & Wigderson [140] showed that the problem of constructing a maximal independent set of vertices in a graph is in NC^4.

Soon thereafter, effecient randomized parallel algorithms for the problem were presented in [157] and [8]. On a graph with n vertices and m edges, these algorithms run in time $O(\log^2 n)$ on an EREW PRAM, and require $O(m+n)$ processors. Goldberg & Spencer [104] give a deterministic algorithm for constructing a maximal independent set that runs on an EREW PRAM in time $O(\log^4 n)$ using $O(m+n)$ processors.

All of these algorithms for constructing a maximal independent set in a graph $G'=(V',E')$ have the following overall structure.

```
begin
  $I \leftarrow \emptyset$: $V \rightarrow V'$;
  while $V \neq \emptyset$ do
  begin
    $G \leftarrow$ the subgraph of $G'$ induced by the vertex set $V$;
    $S \leftarrow IN(G)$;
    $V \leftarrow V-(S \cup N(S))$
    $I \leftarrow I \cup S$
  end
end.
```

Here $IN(G)$ is an independent set in the graph G, and $N(S)$ denotes the set of vertices in V' that are adjacent to one or more vertices in S. It is easy to check that, upon termination of this algorithm, S is a maximal independent set in G.

The algorithms differ in the way they construct the independent set $IN(G)$. Luby's randomized method [157] is as follows. Let $d_G(v)$ denote the degree of vertex v in G. Then $IN(G)$ is constructed as follows:

```
$X \leftarrow \emptyset$
in parallel for all $v \in V$ do
  insert $v$ into $X$ with probability $1/(2d_G(v))$
in parallel for all 2-element sets $\{u,v\} \subseteq X$ such that $\{u,v\}$ is an edge of $G$ do
  if $d_G(u) < d_G(v)$ then delete $u$ from $X$;
  if $d_G(v) < d_G(u)$ then delete $v$ from $X$;
  if $d_G(u) = d_G(v)$ then randomly choose either $u$ or $v$ and delete it from $X$.
$IN(G) \leftarrow X$.
```

The crucial lemma in Luby's analysis of his algorithm is as follows.

LEMMA. *Let E be the edge set of G. Then the expected number of edges of G incident with vertices in $IN(G) \cup N(IN(G))$ is at least $\frac{1}{8}|E|$.*

From the lemma, it easily follows that the expected number of calls on procedure $IN(G)$ required to construct a maximal independent set in the original graph G' is $O(\log n)$, where n is the number of vertices in G'. As pointed out in [157], it is sufficient for the random trials in the algorithm to be pairwise independent rather than mutually independent. This observation permits the trials to be taken from a probability space small enough to be searched exhaustively, yielding a deterministic NC-algorithm for the MIS problem. Luby [158] gives a more refined method of searching through this probability space, leading to a deterministic NC-algorithm with a linear number of processors.

The algorithm of Goldberg & Spencer [104] is based on a direct deterministic method of constructing an independent set S in a graph $G=(V,E)$. Let $|V|=p$ and $|E|=q$. Then, on an EREW PRAM their method runs in time $O(\log^2 p)$ using $O(p+q)$ processors, and guarantees that $|S \cup N(S)| \geqslant p/3 \log p$. If procedure IN is implemented using their method then the total number of calls on the procedure will be $O(\log^2 n)$, and the overall algorithm will require $n+m$ processors and run in time $O(\log^4 n)$ on an EREW PRAM.

Thus we see that there is a polylog-time deterministic algorithm to construct a maximal independent set using a linear number of processors; perhaps, in future work, the power of $\log n$ in the time bound will be reduced below 4.

4.8. Applications

The algorithms given above for computing the transitive closure of a Boolean matrix, for computing the characteristic polynomial and the rank of a matrix with entries drawn from a field, and for the fast parallel evaluation of functions specified by straight-line programs, provide fundamental tools for placing problems in NC. In this section we describe several of these applications.

4.8.1. Applications of transitive closure techniques

The following is a list of six basic computational problems regarding digraphs:

(i) computing the strong components of a digraph G;

(ii) determining whether G is acyclic;

(iii) constructing a tree rooted at a given vertex v of G and containing all vertices reachable from v;

(iv) constructing a breadth-first search tree rooted at a given vertex v;

(v) constructing a topological ordering of the vertices of an acyclic digraph G; i.e., a bijection h from the set of n vertices onto the integers $1, 2, \ldots, n$ such that, for every directed edge (u, v), $h(u) < h(v)$;

(vi) computing shortest paths from a given root vertex v to all other vertices, in a digraph G whose edges have nonnegative weights.

All of these problems can be placed in NC^2 using techniques related to transitive closure. The solutions of (i) and (ii) can be read off directly from the transitive closure. An NC^2-algorithm for problem (vi) is easily constructed, based on the well-known technique of iterated min/+ matrix multiplication. Problem (iv) is a special case of problem (vi), and a solution to (iv) also solves (iii). Problem (v) can be solved by computing the length of a longest path to each vertex by iterated max/+ matrix multiplication, sorting the vertices in increasing order of their longest path lengths (ties being broken arbitrarily), and then assigning each vertex a number equal to its rank in this sorted order.

Using techniques related to transitive closure, we have exhibited NC^2-algorithms for six elementary problems related to digraphs. Unfortunately, none of these algorithms are efficient. On a graph with n vertices and m edges, problem (vi) can be solved sequentially in time $O(n \log n + m)$, and each of the other problems can be solved sequentially in time $O(n+m)$. By contrast, our parallel algorithms run in time $O(\log^2 n)$ and require $M(n)$ processors in the case of problems (i)–(v), and $M'(n)$ processors in case (vi) (see [96]). Thus, we see that the processor–time product for each of our algorithms is far in excess of the time required to solve the same problem sequentially. In order to construct efficient parallel algorithms for these problems, it will be necessary to avoid the use of matrix powering or transitive closure as a subroutine; our inability to do so is sometimes called the *transitive closure bottleneck.*

4.8.2. Problems reducible to the solution of linear equations

We have shown that the problems of inverting a matrix and computing the rank of a matrix are in arithmetic NC^2. It follows easily that the problem of solving the linear system $Ax=b$ (where A is not necessarily of full rank) is also in arithmetic NC^2.

Many problems can be reduced to the solution of a system of linear equations. As an example, we consider the problem of computing the greatest common divisor (gcd) of two univariate polynomials f and g of degree n [40]. It can be shown that gcd(f, g) is of degree $\leqslant i$ if and only if there exist polynomials s and t of degree less than $n-i$ such that $sf+tg$ is of degree i. The condition that such polynomials exist is expressed by a nonsingular system of $2n-2i$ linear equations in which the unknowns are the $2n-2i$ coefficients of s and t. Each entry in the matrix of this system is either 0, a coefficient of f, or a coefficient of g.

As a second example, consider the problem of drawing a 3-connected planar graph in the plane without crossing edges, in such a way that all the edges are line segments. Call a cycle C of G a bounding cycle if there exists a plane imbedding of G in which the cycle C bounds a face. If T is any spanning tree of G then each edge e not in T forms a unique "fundamental cycle" when added to T; at least one of these fundamental cycles is a bounding cycle. In an interesting paper entitled "How to Draw a Graph", Tutte [225] gives the following result: Let G be a 3-connected planar graph, and let C be a bounding cycle. Let the successive vertices of C in cyclic order be $v_1, v_2, \ldots, v_k$. Let $p_1, p_2 \ldots, p_k$ be points in the plane such that the line segments $\overline{p_1 p_2}, \overline{p_2 p_3}, \ldots, \overline{p_k p_1}$ determine a convex polygon. Let the vertices of G be placed in the plane so that v_i is placed at p_i, $i=1,2,\ldots,k$, and each vertex not on C is located at the center of gravity of the vertices adjacent to it in G. For each edge $\{u, v\}$, let the points corresponding to u and v be joined by a line segment. Then no two of these lines segments will intersect except at a common endpoint; i.e., the process produces a straight-line imbedding of G. Thus, as Ja'Ja' & Simon [123] have observed, a straight-line imbedding of G can be constructed by identifying a bounding cycle C, and then constructing the associated placement. Once the vertices of C have been placed at the vertices of a convex polygon, the placement of the remaining points can be obtained by solving a nonsingular system of linear equations. The unknowns are the x- and y-coordinates of the vertices not on C, and the linear equations express the condition that each vertex lies at the center of gravity of its neighbors. This approach leads to an NC-algorithm for constructing a straight-line imbedding.

4.8.3. *Applications of Mulmuley's rank algorithm*

Mulmuley's algorithm [174] for computing the rank of a matrix over an arbitrary field is also a powerful tool for placing problems in NC. Among these are the problem of solving a (possibly singular) system of linear equations and the problem of factoring polynomials over finite fields. Mulmuley's algorithm also enables a number of problems about permutation groups presented by generators to be placed in NC [23]. These include the problems of determining the order, finding the derived series or a composition series, testing membership, testing if the permutation group is solvable, finding a central composition series, and finding pointwise stabilizers of sets when the permutation group is nilpotent. In each case Mulmuley's algorithm permits an earlier randomized algorithm to be made deterministic.

4.8.4. *Context-free recognition—an application of the technique of Miller, Ramachandran and Kaltofen*

The technique of Miller, Ramachandran & Kaltofen [170] for parallel evaluation of straight-line code enables several particular problems to be placed in NC. It is especially applicable to problems that can be solved in polynomial sequential time using dynamic programming. We shall illustrate the approach by giving an AC^1-algorithm for the problem of deciding whether a given string is in the language generated by a given context-free grammar in Chomsky Normal Form.

Our starting point is the well-known Cocke–Kasami–Younger dynamic programming algorithm for context-free recognition [141, 244]. Let (V, Σ, P, S) be a context-free grammar, where V denotes the set of symbols, Σ denotes the set of terminal symbols, P denotes the set of productions, and S denotes the initial symbol. Since the grammar is in Chomsky Normal Form, each production is of the form $A \to BC$ or $A \to a$, where A, B and C denote nonterminal symbols and a denotes a terminal symbol. For $A \in V - \Sigma$ and $w \in \Sigma^*$, we say that $A \overset{*}{\to} w$ (*A derives w*) if the nonterminal symbol A can generate the string w.

Let $x = a_1 a_2 \dots a_n$ be the input string, and let $x(i,j)$ be the substring $a_1 a_{i+1} \dots a_j$. For $1 \leqslant i \leqslant j \leqslant n$, and for every nonterminal symbol A, let the predicate $A(i,j)$ be true if $A \overset{*}{\to} x(i,j)$. The Cocke–Kasami–Younger algorithm evaluates all such predicates; the string x is accepted if and only if the predicate $S(1,n)$ is true.

The evaluation of these predicates is based on the following rules:

(i) for $i = 1, 2, \dots, n$, $A(i,i)$ is true if and only if there is a production of the form $A \to a_i$;

(ii) for $1 \leqslant i < j \leqslant n$, $A(i,j)$ is true if and only if there exists a k, $i \leqslant k < j$, and a production $A \to BC$, such that $B(i, k)$ and $C(k+1, j)$ are true.

The resulting algorithm can be represented as an arithmetic circuit over the semiring $(\{0,1\}, +, *, 0, 1)$, where $+$ is Boolean OR and $*$ is Boolean AND. The circuit contains $O(n^3)$ nodes, and the node that computes the predicate $A(i,j)$ is of degree $j - i + 1$. Thus d, the maximum degree of any node, is $n+1$, and it follows that the number of iterations required by the parallel evaluation algorithm of [170] is $O(\log n)$. The time for each iteration is dominated by the matrix multiplication required in the MM step of that algorithm. Since the semiring in this case is Boolean, matrix multiplication is in AC^0, and thus the time for the entire algorithm is O(log n), using a polynomial-bounded number of processors. It follows that the problem of deciding whether a given string is in the language generated by a given context-free grammar in Chomsky Normal Form is in AC^1. In [202], Ruzzo proved the static version of this result in a different way by showing that every context-free language can be recognized by an alternating Turing machine operating within space log n and using log n alternations. It follows that every context-free language is in AC^1.

4.9. Randomized NC-algorithms

4.9.1. Testing whether a symbolic determinant is nonzero

The following lemma is an important tool for the design of randomized parallel algorithms.

LEMMA. *Let $F(x_1, x_2, \ldots, x_n)$ be a polynomial of degree d. If F is not identically zero then, for at least half of the $(2d+1)^n$ n-tuples in which each component is an integer between $-d$ and d, $F(a_1, a_2, \ldots, a_n) \neq 0$.*

We omit the proof, which goes by induction on n, with the case $n=1$ corresponding to the Fundamental Theorem of Algebra (see, e.g., [209]).

The lemma suggests a randomized algorithm for testing whether a polynomial of degree d is not identically zero: simply substitute independent random integers between d and $-d$ for the variables. If the polynomial is not identically zero then, with probability greater than $\frac{1}{2}$, a non-zero value will result. For any class of polynomial expressions that can be evaluated by an NC-algorithm, we obtain in this way an RNC-algorithm for testing whether a polynomial in that class is not identically zero.

4.9.2. The matching problem

We give an RNC^2-algorithm for recognizing the graphs that possess perfect matchings (recall that a perfect matching in a graph G is a set of edges M, such that each vertex of G is incident with exactly one vertex of M). The algorithm is based on the following theorem attributed to Tutte.

THEOREM. *Let G be a simple graph (no loops or multiple edges) with vertex set $\{1, 2, \ldots, n\}$ and edge set E. Let $A = (a_{ij})$ be the following $n \times n$ matrix, in which the variables x_{ij} are indeterminates:*

$$a_{ij} = \begin{cases} x_{ij} & \text{if } \{i,j\} \in E \text{ and } i<j, \\ -x_{ji} & \text{if } \{i,j\} \in E \text{ and } i>j, \\ 0 & \text{if } \{i,j\} \notin E. \end{cases}$$

Then G has a perfect matching if and only if the determinant of A is not identically 0.

The matrix A is called the Tutte matrix of the graph G. The determinant of A is a polynomial of degree n in the indeterminates x_{ij}. Combining Tutte's Theorem, the lemma, and the existence of an NC^2-algorithm for computing the determinant of an $n \times n$ matrix with integer entries in $[-n, n]$, we obtain the desired one-sided-error algorithm for deciding whether a graph has a perfect matching.

This result does not directly yield an RNC^2-algorithm for constructing a perfect matching when one exists. Such an algorithm was first provided in [138]. We present here a particularly elegant RNC^2-algorithm for the problem, due to Mulmuley, Vazirani & Vazirani [175]. Their algorithm is based on the following probabilistic lemma.

LEMMA. *Let C be any nonempty collection of subsets of $\{1, 2, \ldots, N\}$. Let $w(1), w(2), \ldots, w(N)$ be independent random variables, each with the uniform distribution over $\{0, 1, 2, \ldots, 2N\}$. Associate with each set $S \subseteq \{1, 2, \ldots, N\}$ a weight $w(S) = \Sigma_{i \in S} w(i)$. Then, with probability greater than $\frac{1}{2}$, the family C contains a unique set of minimum weight.*

The lemma can be applied to the matching problem by taking C to be the set of perfect matchings in a graph with N edges. If each edge is given a weight drawn from the uniform distribution over $\{0, 1, \ldots, 2N\}$, and the weight of a matching is the sum of the weights of its edges, then, with probability $> \frac{1}{2}$, there will be a unique perfect matching of minimum weight.

Mulmuley, Vazirani & Vazirani go on to show that when there is a unique perfect matching of minimum weight, it can be constructed at the cost of a single matrix inversion by the following algorithm.

(1) For each edge $\{i, j\}$, draw a weight w_{ij} from the uniform distribution over $\{0, 1, \ldots, 2|E|\}$.
(2) Form the Tutte matrix of G and, for each indeterminate x_{ij} occurring in the Tutte matrix, substitute the constant $2^{w_{ij}}$; let the resulting matrix be B.
(3) Using a parallel matrix inversion algorithm that yields the determinant and the adjoint, such as the one in [180], compute $|B|$ and adj(B). The (i, j) entry of adj(B) is the minor $|B_{ij}|$.
(4) In parallel, for all edges $\{i, j\}$, compute $|B_{ij}|\, 2^{w_{ij}}/2^{2w}$. Let M be the set of edges for which this quantity is odd.
(5) If M is a perfect matching then output M.

Whenever the weights w_{ij} are such that there is a unique perfect matching of minimum weight, the algorithm will produce this matching. Thus each execution of the algorithm produces a perfect matching with probability $> \frac{1}{2}$, provided that a perfect matching exists. The algorithm runs in $O(\log^2 n)$ time using a polynomial-bounded number of processors. Thus we can conclude that the problem of constructing a perfect matching is in RNC^2. Further results by Karloff [134] establish that the problem lies in ZNC^2.

We have discussed two problems related to perfect matchings: the *decision problem*, in which the task is to decide whether a perfect matching exists, and the *search problem*, in which the task is to construct a perfect matching when one exists. Karp, Upfal & Wigderson [139] have studied the general question of how to construct a parallel algorithm for a search problem, given a subroutine for the corresponding decision problem.

4.9.3. Applications of matching

Karp, Upfal & Wigderson [138] have shown that the following problems related to matching and network flows are in RNC:

(i) constructing a perfect matching of maximum weight in a graph whose edge weights are given in unary notation;

(ii) constructing a matching of maximum cardinality;

(iii) constructing a matching that covers a set of vertices of maximum weight in a graph whose vertex weights are given in binary;

(iv) constructing a maximum s-t flow in a directed or undirected network whose edge weights are given in unary notation.

Each of these results is obtained by a reduction of the given problem to the problem of constructing a perfect matching, or by a closely related algorithmic technique. In view of the above result of Mulmuley, Vazirani & Vazirani [175], these four problems lie in RNC^2.

4.9.4. Depth-first search

Let $G=(V, E)$ be a connected graph. Let T be a spanning tree of G rooted at r. Then T is called a *depth-first-search tree* if, for every edge $e \in E$, one of the two endpoints of e is an ancestor of the other.

There is a sequential algorithm running in time $O(|E|)$ for the construction of a depth-first search tree rooted at a given vertex. This algorithm is the backbone of linear-time sequential algorithms for testing whether a graph is planar, computing the biconnected components of a graph, and many other important problems (see, e.g., [79].

Although the goal of finding an efficient parallel algorithm for depth-first search has not been reached, some progress has been made. Define a *splitting path* as a simple path having the root as an endpoint, with the property that its deletion breaks the remaining vertices into connected components of size less than or equal to $c|V|$, for a suitable constant c. Once such a splitting path is found, a depth-first search tree for the overall graph can be constructed recursively out of the splitting path together with depth-first search trees for these connected components. This approach is used in [215] to solve the depth-first search problem in the case of planar graphs (see also [102, 116, 122] for efficient implementations of this algorithm), and by Ramachandran [189] in the case of directed acyclic graphs and reducible flow graphs.

It remained an open question whether the depth-first search problem for general graphs was in ZNC. Aggarwal & Anderson [2] settled this question by giving a ZNC^5-algorithm that constructs a depth-first search tree in an n-vertex undirected graph. The algorithm has $O(\log n)$ levels of recursion, in each of which it constructs a splitting path; in order to construct such a path, it makes $O(\log^2 n)$ successive calls to a subroutine for the following problem: given a bipartite graph in which each edge has weight 0 or 1, find a perfect matching of minimum weight. Each of the bipartite graphs presented to the subroutine has at most n vertices. Since this matching problem is in

ZNC^2, it follows that the depth-first search problem is in ZNC^5. The algorithm of Aggarwal & Anderson is quite intricate, and we shall not attempt to describe it here. Building on results of Kao [131] for planar directed graphs, this result is extended to yield a ZNC algorithm for depth-first search in directed graphs in [3].

4.10. *Further results*

Our treatment of NC-algorithms has focused on methods and results that appear to be of general utility for placing problems in the classes NC, RNC or ZNC, or for locating them in the hierarchies such as $\{AC^k\}$ and $\{NC^k\}$. Many further problems have been analyzed from this point of view. The following references give a representative sample of such results.

Graph theory: Babai [22]; Coppersmith, Raghavan & Tompa [65]; Dahlhaus & Karpinski [68]; Galil & Pan [89]; Gazit & Miller [95]; Gibbons, Karp, Miller & Soroker [101]; Grigoriev & Karpinski [109]; Johnson [124]; Karloff [134]; Karloff & Shmoys [136]; Klein & Reif [143]; Kozen, Vazirani & Vazirani [146]; Lev, Pippenger & Valiant [152]; Lingas & Karpinski [154]; Lovász [156]; Nisan & Soroker [178]; Reif [193, 194]; Savage & Ja'Ja' [205]; Soroker [218, 219], Tsin & Chin [224]; Vazirani [232].

Scheduling theory: Hembold & Mayr [117].

Algebra: Babai, Luks & Seress [23]; Ben-Or & Tiwari [32]; Borodin, von zur Gathen & Hopcroft [40]; Eberly [75]; Fich & Tompa [84]; Galil & Pan [89]; von zur Gathen [90, 91, 92]; Ibarra, Moran & Rosier [121]; Kaltofen, Krishnamoorthy & Saunders [127]; Kaltofen & Saunders [128]; Kannan, Miller & Rudolph [130]; Litow & Davida [155]; Lueker, Megiddo & Ramachandran [159]; Luks [160]; Luks & McKenzie [161]; McKenzie & Cook [165]; Pan [180]; Pan & Reif [181]; Reif [197].

Language theory: Ibarra, Jiang, Ravikumar & Chang [120]; Klein & Reif [144].

Logic programming: Mayr [163]; Ullman & van Gelder [226].

Analysis: Ben-Or, Feig, Kozen & Tiwari [30]; Ben-Or, Kozen & Reif [31]; Kozen & Yap [147].

In addition to the work on NC, there is an important body of research concerned with algorithms that achieve significant speed-up, even if they do not run in polylog time. A prominent example is Gaussian elimination, which solves a system of n linear equations in time $O(n)$ using n^2 processors. Further examples include the algorithm due to Eckstein [76] for performing depth-first search in an n-node, m-edge graph in time $O(m/p+n \log p)$, using p processors; the algorithm of Gabow & Tarjan [87] for the assignment problem on a weighted bipartite graph with n nodes, m edges and maximum edge weight N that runs in

$$O(\sqrt{n}\, m \log(nN)\ (\log 2p)/p)$$

time with $p \leqslant m/(\sqrt{n} \log^2 n)$ processors; and the algorithms of Goldberg, Plotkin & Vaidya [103] for performing depth-first search on a graph with n vertices and m edges in time $O(\sqrt{n}\, polylog(n))$ using $n+m$ processors, and for computing a maximum matching in an n-node bipartite graph in time $O(n^{2/3} polylog(n))$ using $M(n)$

processors. In the terminology of Kruskal, Rudolph & Snir (cf. Section 2.1), Eckstein's algorithm achieves polynomial speed-up for dense graphs with constant inefficiency, the algorithms of Gabow & Tarjan achieve polynomial speed-up with logarithmic inefficiency, and the algorithms of Goldberg, Plotkin & Vaidya achieve polynomial speed-up with polynomial inefficiency.

4.11. P-*complete problems*

Let P denote the set of decision problems solvable by deterministic Turing machines in polynomial time. Every decision problem in NC lies in P. A fundamental open question is whether every problem in P lies in NC. If this were so, it would mean, roughly speaking, that every problem that is efficiently solvable in a sequential model of computation can be solved very fast in parallel, using a polynomial-bounded number of processors (this interpretation is disputed in the interesting paper by Vitter & Simons [239]). Using a reducibility technique, we shall identify a set of problems within P called the P-complete problems; a P-complete problem lies in NC if and only if P = NC. Thus the P-complete problems can be viewed as the problems in P most resistant to parallelization.

We adopt the usual convention of representing a decision problem as a subset of $\{0,1\}^*$. Decision problem A is said to be *logspace-reducible* to decision problem B if there is a function $f:\{0,1\}^*\rightarrow\{0,1\}^*$ such that f is computable by a logspace Turing machine and, for all $x\in\{0,1\}^*$, $x\in A$ if and only if $f(x)\in B$. A decision problem in P is called *P-complete* if every problem in P is logspace-reducible to it. The relation of logspace-reducibility is transitive and has the following further property: if A is logspace-reducible to B and B is in NC^k, where $k\geqslant 2$, then A is in NC^k. Our interest in P-completeness stems from the following consequences of this observation: let A be a P-complete problem; then $\mathrm{P}=\mathrm{NC}$ if and only if A lies in NC, and, for $k\geqslant 2$, $\mathrm{P}\subseteq\mathrm{NC}^k$ if and only if A lies in NC^k.

Our official definition of a P-complete problem requires that it be specified as a function from strings over one alphabet to strings over another. In practice, the inputs and outputs to a problem are often other kinds of symbolic objects: graphs, formulas, circuits, grammars, families of sets and the like. In describing such problems and proving them P-complete, we will often not specify how their inputs and outputs are encoded as strings, since the details of that encoding are seldom of interest.

The usual method of proving a problem P-complete is to show that it lies in P and that some standard P-complete problem is logspace-reducible to it. The standard problem most often used for this purpose is the monotone circuit value problem (MCVP) (Goldschlager [105]). Informally, the input to this problem is a single-output fan-in-2 Boolean circuit without NOT-gates, together with an assignment of a constant value (0 or 1) to each input line. More precisely, the input is given as a sequence of equations, specifying the outputs of the gates in a monotone circuit. The first two equations are $g_0=0$ and $g_1=1$, and for each i, $i=2,3,\ldots,n$, there is either an equation of the form $g_i=g_j\vee g_k$ or an equation of the form $g_i=g_j\wedge g_k$, where j and k are nonnegative indices less than i. The first of these two equations corresponds to the case where gate i is an OR-gate with inputs from gates j and k; similarly, the second equation corresponds to the case where gate i is an AND-gate. The output is the value of g_n in the unique solution of this system of equations.

We sketch the proof that MCVP is P-complete. Let A be a decision problem in P. Then A is accepted by a deterministic one-tape Turing machine M which has a unique accepting state q^* and operates within the polynomial time bound $T(n)$. For a given input string x, let $T=T(|x|)$. Then T is an upper bound on the execution time of M on x, and the only tape squares whose contents can change during the computation are within distance T of the home square. Introduce the following Boolean variables to represent the computation of M on input x:

- $h(i,t)$, meaning "the head is on tape square i at time t";
- $a(i,t)$, meaning "the symbol on square i at time t is a";
- $q(t)$, meaning "the state at time t is q".

Here a ranges over the tape alphabet of M, q ranges over the state set of M, $-T \leqslant i \leqslant T$ and $0 \leqslant t \leqslant T$. The input x is accepted if and only if $q^*(T)=1$.

Given input x and the description of the transition function of M, there is a logspace algorithm to generate a system of monotone Boolean equations, in the format of the MCVP problem, specifying the values of these variables. This is the required logspace reduction from problem A to the MCVP. Since A is an arbitrary problem in P, it follows that MCVP is P-complete.

We shall give two further examples of P-completeness proofs, each involving a reduction from MCVP. The first example involves a certain "greedy" sequential algorithm for constructing a maximal independent set of vertices in a graph. Let G be a graph with vertex set V, and let a linear ordering of V be given. The following algorithm constructs a maximal independent set in G.

```
GREEDY INDEPENDENT SET ALGORITHM
$S \leftarrow \emptyset$;
for $i=1,2,\ldots,|V|$ do
begin
    let $v$ be the $i$th element in the linear ordering of $V$;
    if $v$ is adjacent to no vertex in $S$
    then $S \leftarrow S \cup \{v\}$
end.
```

The following decision problem related to this algorithm is P-complete (Cook [61]).

GREEDY INDEPENDENT SET

Input: Graph $G=(V,E)$ where V is linearly ordered,

Property: The last vertex in the linear ordering of V is in the independent set constructed by the greedy independent set algorithm.

We give a reduction from MCVP. Let an instance of MCVP be specified, as described above, by equations for the gate outputs $g_0, g_1, \ldots, g_n$. The reduction will produce a graph G with vertex set $V=\{v_0, v_1, \ldots, v_n\} \cup \{w_0, w_1, \ldots, w_n)$, together with a linear ordering of V. The linear ordering is such that, whenever $i<j$, v_i and w_i precede

v_j and w_j. The relative ordering of v_i and w_i and the specification of the edges incident with v_i and w_i depend on the nature of the equation for g_i as follows: w_0 precedes v_0; v_1 precedes w_1; if the equation for g_i is $g_i = g_j \vee g_k$ then w_i precedes v_i and the graph contains the edges $\{v_j, w_i\}$ and $\{v_k, w_i\}$; if the equation is $g_i = g_j \wedge g_k$ then v_i precedes w_i and the graph contains the edges $\{w_j, v_i\}$ and $\{w_k, v_i\}$. In addition, all edges $\{v_i, w_i\}$ are present.

The construction of the graph from the circuit can be performed by a logspace algorithm. It is easy to prove by induction on n that v_i lies in the greedy independent set if and only if the output of g_i is 1, and w_i lies in the greedy independent set if and only if the output of g_i is 0. Thus we have a logspace-reduction from the monotone circuit value problem to the lexicographically-first-independent-set problem, showing that the latter problem is P-complete. The P-completeness of this problem stands in contrast to the result that the problem of constructing some maximal independent set, not necessarily the one produced by the greedy algorithm, is in EREW^4.

As a final example, we show that the following problem is P-complete [107].

MAX-FLOW

Input: A directed graph $G = (V, E)$, a pair of distinct vertices s and t called the source and sink respectively, and a function c from E into the nonnegative integers, assigning to each edge e a capacity $c(e)$.

Property: The value of the maximum flow from s to t is an odd integer.

We give a reduction from the MCVP to MAX-FLOW. Let C be a monotone circuit. We may assume without loss of generality that the following properties hold:

(i) each gate has fan-out at most 2 (i.e., each variable g_i occurs on the right-hand side of at most two equations);

(ii) g_n is the output of an OR-gate.

We refer to the edges of the circuit as wires in order to distinguish them from the edges of the network produced by the reduction. The network has vertex set $\{s, t, v_0, v_1, \ldots, v_n\}$, and its edges and capacities are specified as follows:

(i) for each wire of C connecting the output of g_j to the input of g_i, there is an edge of capacity 2^{n-j} from v_j to v_i;

(ii) there is an edge of capacity 2^{n+1} from s to v_n;

(iii) there is an edge of capacity 1 from v_n to t;

(iv) if g_i is an OR-gate then there is an edge from v_i to s;

(v) if g_i is an AND-gate, there is an edge from v_i to t and

(vi) the capacities of the edges specified in (iv) and (v) are such that the sum of the capacities of the edges directed into any vertex v_i is equal to the sum of the capacities of the edges directed out of that vertex.

It is easy to prove by induction on n that there is a maximum flow in G having the following properties:

(i) if a wire in C carries the signal 0 then the flow in the corresponding edge of G is 0;

(ii) if a wire in C carries the signal 1 then the flow in the corresponding edge of G is equal to the capacity of that edge;

(iii) with the exception of the edges from v_n to s and t, the flow in every edge of G is an even integer;

(iv) The flow in the edge from v_n to t is 0 if the output of C is 0, and 1 if the output of C is 1.

It follows that the value of a maximum flow in G is odd if and only if the output of C is 1. This establishes that the reduction is correct.

Thus we see that the max-flow problem is P-complete. In contrast to this is the result given in Section 4.9.3 that when the capacities of the edges are bounded by a polynomial in the number of vertices, the max-flow problem is in ZNC^2.

The seminal results on P-complete problems are given in [60, 125, 149]. More recent P-completeness results can be found in [11, 19, 69, 71, 135, 195].

4.12. Open problems

The following is a list of problems, solvable in sequential polynomial time, whose parallel complexity remains unknown despite the efforts of many researchers. It is not known whether these problems lie in ZNC, and they have not been proven to be P-complete. One resolution of these questions would be a proof that P = NC, but this seems unlikely to be true and, in any case, unlikely to be settled by known proof methods.

1. EXISTENCE OF A PERFECT MATCHING
Input: A graph G.
Question: Does G have a perfect matching?

This problem is in ZNC, as is the related problem of constructing a perfect matching when one exists (see Section 4.9.2). In the bipartite case, an algorithm running in $O(n^{2/3}polylog(n))$ time using a polynomial-bounded number of processors is known [103].

2. UNDIRECTED DEPTH-FIRST SEARCH
Input: A connected graph G and a vertex v.
Output: A spanning tree T of G, rooted at v and having the following property: for each non-tree edge $\{u, w\}$, u is either an ancestor or a descendant of w in T.

This problem is in RNC (see Section 4.9.4). A $\sqrt{n}\, polylog(n)$ time algorithm using $poly(n)$ processors is known [103].

3. DIRECTED DEPTH-FIRST SEARCH
Input: A digraph G and a vertex v from which all vertices of G are reachable.
Output: An oriented tree T, rooted at v and containing a directed path from v to each vertex of G, such that T is a subgraph of G and has the following directed depth-first search property: there is a preorder numbering of G such that, if (u, w) is an edge of $G - T$ then either w precedes u in the preorder numbering, or w is a descendant of u in T.

The problem is in RNC [3].

4. Weighted Matching

Input: A graph G with n vertices and, for each edge e, a positive integer weight $w(e)$.

Output: A matching of maximum total weight.

When the weights are polynomial in n this problem is in ZNC [138].

5. Maximal Independent Set in a Hypergraph

Input: A collection C of subsets of a finite set X.

Output: A set $X' \subseteq X$ which is maximal with respect to the property that it does not contain any set in the collection C.

The case where C is a collection of 2-element sets is the maximal independent set problem for graphs, which is in NC (see Section 4.7).

6. Two-Variable Linear Programming

Input: A linear system of inequalities $Ax \leqslant b$ over the rationals, such that each row of A has at most two non-zero elements.

Output: A feasible solution, if one exists.

This problem has a polylog-time parallel algorithm with $n^{polylog(n)}$ processors [159].

7. Integer GCD

Input: Integers a and b.

Output: The greatest common divisor of a and b.

A sublinear-time algorithm using a polynomial-bounded number of processors is given in [130], and subsequently improved in [50]. The problem of computing the greatest common divisor of two polynomials is in NC (see Section 4.8.2).

8. Modular Integer Exponentiation

Input: n-bit integers a, b and m.

Output: $a^b \bmod m$.

9. Modular Polynomial Exponentiation

Input: Polynomials $a(x)$ and $m(x)$ with coefficient from a ring, and a positive integer e.

Output: $a(x)^e \bmod m(x)$.

Problem 8 is a special case. Over fields of some fixed characteristic the problem is in arithmetic NC^2, provided that unit-time operations over the base field are assumed [84].

10. Polynomial Root Approximation

Input: An nth-degree polynomial $P(x)$ with integer coefficients, and a positive error bound ε.

Output: For each (real or complex) root ξ of P, an approximation ξ' such that $|\xi-\xi'|<\varepsilon$.

When all roots are real the problem is in NC [30].

11. COMPARATOR CIRCUIT VALUE
Input: A comparator network (see Section 2.4) together with values for its inputs.
Output: The value of a specified output.

This problem is complete in the complexity class CC defined in [164].

5. Conclusion

The examples we have given in this survey of parallel algorithms are of interest not only for their theoretical significance, but also as illustrations of typical methods of exploiting the parallelism inherent in problems. Basic algorithms for such problems as prefix sums, list ranking, sorting, and graph searching can serve as fundamental building blocks for further algorithm construction. As parallel computation grows in importance, such algorithms will find their way into the undergraduate textbooks, and will become part of the general lore of computer science.

We have seen that, within many different abstract models, the class NC represents the collection of problems solvable in polylog time with a polynomial-bounded number of computing elements, and we have identified many basic problems as lying in NC. The robustness of this class suggests that it is of fundamental importance, and lends interest to the question of whether NC coincides with the familiar complexity class P. We have also seen the usefulness of the concept of P-completeness in identifying the problems in P least likely to lie in NC.

So far, the studies of complexity within the PRAM model have been somewhat unrealistic because of the assumption that the number of processors is allowed to grow as a function of the size of the input. It would be useful to complement this line of research with studies in which the number of processors remains fixed as the input size grows; this assumption is closer to the typical situation in practice.

Although the PRAM model neglects communication issues, it is a very convenient vehicle for the logical design of parallel algorithms. For this reason, there has been intense interest in the simulation of PRAMs on more feasible models of parallel computation. The chief components of such a simulation are the choice of a processor interconnection pattern; the mapping of the address space of the PRAM onto the set of memory cells of the simulating machine; and the algorithm for routing read and write requests, and the replies to read requests, through the network of processors.

A series of more and more refined PRAM simulations [166, 227, 228, 133, 10] has culminated in Ranade's efficient randomized simulation of an n-processor CRCW PRAM on an n-processor butterfly network [191]. The simulation time per PRAM step is only logarithmic in the number of processors. This simulation opens the way for a programming environment in which algorithms are designed within the convenient

PRAM model and then simulated efficiently on a network of processors, and thus underscores the value of the PRAM model, and the applicability of algorithms designed within it.

Acknowledgment

Many colleagues have provided valuable comments on earlier drafts of this chapter. We would like to express our thanks to Richard Anderson, Laszlo Babai, David Barrington, Robert Boyer, Joachim von zur Gathen, Hillel Gazit, Mark Goldberg, Michael Goodrich, Erich Kaltofen, János Komlós, Jan van Leeuwen, Michael Loui, Eugene Luks, Jay Misra, Ketan Mulmuley, Steve Omohundro, Victor Pan, James Renegar, John Reif, Wojciech Rytter, John Savage, Marc Snir, Robert Tarjan, Prasoon Tiwari, Martin Tompa, Stephen Vavasis, Jeffrey Vitter, Yaacov Yesha and, especially, Torben Hagerup and Larry Stockmeyer. Finally, we owe an enormous debt of gratitude to Ann DiFruscia, who assisted us with great dedication and skill in the preparation of this manuscript.

The research of the first author was supported by the International Computer Science Institute, Berkeley, CA, and NSF Grant Nos. DCR-8411954 and CCR-8612563. The research of the second author was supported by the International Computer Science Institute, Berkeley, CA, and by Joint Services Electronics Program NOOO14-84-C-0149 while she was with the University of Illinois, Urbana, IL.

References

[1] Abrahamson, K., N. Dadoun, D.A. Kirkpatrick and T. Przytycka, A simple parallel tree contraction algorithm, in: *Proc. 25th Ann. Allerton Conf. on Communication, Control and Computing* (1987) 624–633.

[2] Aggarwal, A. and R.J. Anderson, A random NC algorithm for depth first search, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 325–334; revised version, *Combinatorica* **8** (1988) 1–12.

[3] Aggarwal, A., R.J. Anderson and M. Kao, Parallel depth-first search in general directed graphs, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989) 297–308.

[4] Aggarwal, A., B. Chazelle, L. Guibas, C. O'Dunlaing and C. Yap, Parallel computational geometry, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 468–477.

[5] Ajtai, M., Σ_1^1-formulae on finite structures, *Ann. Pure Appl. Logic* **24** (1983) 1–48.

[6] Ajtai, M., J. Komlós, W.L. Steiger and E. Szemerédi, Deterministic selection in $O(\log\log n)$ parallel time, in: *Proc. 18th Ann. ACM Symp. on Theory of Computing* (1986) 188–195.

[7] Ajtai, M., J. Komlós and E. Szemerédi, Sorting in $c \log n$ parallel steps, *Combinatorica* **3** (1983) 1–19.

[8] Alon, N., L. Babai and A. Itai, A fast and simple randomized parallel algorithm for the maximal independent set problem, *J. Algorithms* **7** (1986) 567–583.

[9] Alt, H., Comparison of arithmetic functions with respect to Boolean circuit depth, in: *Proc. 16th Ann. ACM Symp. on Theory of Computing* (1984) 466–470.

[10] Alt, H., T. Hagerup, K. Mehlhorn and F.P. Preparata, Deterministic simulation of idealized parallel computers on more realistic ones, *SIAM J. Comput* **16** (1987) 808–835.

[11] Anderson, R.J. and E.W. Mayr, Parallelism and greedy algorithms, in: Advances in Computing Research, Vol. 24 (JAI Press, Greenwich, CT, 1987) 17–38.

[12] Anderson, R.J. and G.L. Miller, Deterministic parallel list ranking, in: *VLSI Algorithms and*

Architectures, Proc. 3rd Aegean Workshop on Computing, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 81–90.

[13] ATALLAH, M.J., Parallel strong orientation of an undirected graph, *Inf. Proc. Letters* **18** (1984) 37–39.

[14] ATALLAH, M.J., R. COLE and M.T. GOODRICH, Cascading divide-and-conquer: a technique for designing parallel algorithms, in: *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 151–160.

[15] ATALLAH, M.J. and M.T. GOODRICH, Efficient parallel solutions to some geometric problems, *J. Parallel Distr. Comput.* **3** (1986) 492–507.

[16] ATALLAH, M.J. and M.T. GOODRICH, Parallel algorithms for some functions of two convex polygons, *Algorithmica* **3** (1988) 535–548.

[17] ATALLAH, M.J., M.T. GOODRICH and S.R. KOSARAJU, Parallel algorithms for evaluating sequences of set manipulation operations, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 1–10.

[18] ATALLAH, M. and U. VISHKIN, Finding Euler tours in parallel, *J. Comput. System Sci.* **29** (1984) 330–337.

[19] AVENHAUS J. and K. MADLENER, The Nielsen reduction and P-complete problems in free groups, *Theoret. Comput. Sci.* **32** (1984) 61–76.

[20] AWERBUCH, B., A. ISRAELI and Y. SHILOACH, Finding Euler circuits in logarithmic parallel time, in: *Proc. 16th Ann. ACM Symp. on Theory of Computing* (1984) 249–257.

[21] AWERBUCH, B. and Y. SHILOACH, New connectivity and MSF algorithms for shuffle-exchange network and PRAM, *IEEE Trans. Comput.* **36** (1987) 1258–1263.

[22] BABAI, L., A Las-Vegas NC algorithm for isomorphism of graphs with bounded multiplicity of eigenvalues, in: *Proc. 27th Ann. IEEE Symp. on Foundations of Computer Science* (1986) 303–312.

[23] BABAI, L., E.M. LUKS and A. SERESS, Permutation groups in NC, in: *Proc.19th Ann. ACM Symp. on Theory of Computing* (1987) 409–420.

[24] BAR-ON, I. and U. VISHKIN, Optimal parallel generation of a computation tree form, *ACM Trans. Programming Languages and Systems* **7** (1985) 348–357.

[25] BARRINGTON, D.A., Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1, in: *Proc 18th Ann. ACM Symp. on Theory of Computing* (1986) 1–5.

[26] BATCHER, K.E., Sorting networks and their applications, in: *Proc. AFIPS Spring Joint Summer Computer Conf., Vol. 32* (1968) 307–314.

[27] BEAME, P.W., Limits on the power of concurrent-write parallel machines, *Inform. and Comput.* **76** (1988) 13–28.

[28] BEAME, P.W., S.A. COOK and H.J. HOOVER, Log depth circuits for division and related problems, *SIAM J. Comput.* **15** (1986) 994–1003.

[29] BEAME, P. and J. HASTAD, Optimal bounds for decision problems on the CRCW PRAM, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 83–93.

[30] BEN-OR, M., E. FEIG, D. KOZEN and P. TIWARI, A fast parallel algorithm for determining all roots of a polynomial with real roots, *SIAM J. Comput.* **17** (1988) 1081–1092.

[31] BEN-OR, M., D. KOZEN and J. REIF, The complexity of elementary algebra and geometry, in: *Proc. 16th Ann. ACM Symp. on Theory of Computing* (1984) 457–464.

[32] BEN-OR, M. and P. TIWARI, A deterministic algorithm for sparse multivariate polynomial interpolation, in: *Proc. 20th Ann. ACM Symp. on Theory of Computing* (1988) 301–309.

[33] BERKMAN, O., D. BRESLAUER, Z. GALIL, B. SCHIEBER and U. VISHKIN, Highly parallelizable problems, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989).

[34] BERKOWITZ, S.J., On computing the determinant in small parallel time using a small number of processors, *Inf. Proc. Letters* **18** (1984) 147–150.

[35] BILARDI, G. and A. NICOLAU, Bitonic sorting with $O(N \log N)$ comparisons, in: *Proc. 20th Ann. Conf. on Information Science and Systems*, Princeton Univ., Princeton, NJ (1986) 309–319.

[36] BOPPANA, R., Optimal separations between concurrent-write parallel machines, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989) 320–326.

[37] BOPPANA, R. and M. SIPSER, The complexity of finite functions, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A* (North-Holland, Amsterdam, 1990) 757–804.

[38] BORODIN, A., On relating time and space to size and depth, *SIAM J. Comput.* **6** (1977) 733–744.

[39] BORODIN, A., S.A. COOK and N. PIPPENGER, Parallel computation for well-endowed rings and space-bounded probabilistic machines, *Inform. and Control* **58** (1983) 113–136.

[40] BORODIN, A., J. VON ZUR GATHEN and J.E. HOPCROFT, Fast parallel matrix and GCD computations, in: *Proc. 23rd Ann. IEEE Symp. on Foundations of Computer Science* (1982) 65–71.

[41] BORODIN, A. and J.E. HOPCROFT, Routing, merging, and sorting on parallel models of computation, *J. Comput. System Sci.* **30** (1985) 130–145.

[42] BRENT, R.P., The parallel evaluation of general arithmetic expressions, *J. ACM* **21** (1974) 201–206.

[43] BUSS, S.R., The Boolean formula value problem is in ALOGTIME, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 123–131.

[44] BUSS, S.R., S.A. COOK, A. GUPTA and V. RAMACHANDRAN, An optimal parallel algorithm for fomula evaluation, Manuscript, Toronto, Canada, 1989.

[45] CHANDRA, A.K., S. FORTUNE, R.J. LIPTON, Unbounded fan-in circuits and associative functions, *J. Comput. System Sci.* **30** (1985) 222–234.

[46] CHANDRA, A.K, D.C. KOZEN and L.J. STOCKMEYER, Alternation, *J. ACM* **28** (1981) 114–133.

[47] CHANDRA, A.K., L. STOCKMEYER and U. VISHKIN, Constant depth reducibility, *SIAM J. Comput.* **13** (1984) 423–439.

[48] CHIN, F.Y., J. LAM and I. CHEN, Efficient parallel algorithms for some graph problems, *Comm. ACM* **25** (1982) 659–665.

[49] CHISTOV, A.L., Fast parallel calculation of the rank of matrices over a field of arbitrary characteristic, in: *Fundamentals of Computation Theory, FCT'85*, Lecture Notes in Computer Science, Vol. 199 (Springer, Berlin, 1985) 63–79.

[50] CHOR, B. and O. GOLDREICH, An improved parallel algorithm for integer gcd, Manuscript, 1985.

[51] CHOW, A., Parallel algorithms for geometric problems, Dissertation, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 1980.

[52] CHROBAK, M. and M. YUNG, Fast parallel and sequential algorithms for edge -coloring planar graphs, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 11–23.

[53] COLE, R., Parallel merge sort, *SIAM J. Comput.* **17** (1988) 770–785.

[54] COLE, R. and M.T. GOODRICH, Optimal parallel algorithms for polygon and point-set problems, in: *Proc. 4th Ann. ACM Conf. on Computational Geometry* (1988) 201–210.

[55] COLE, R. and U. VISHKIN, Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70** (1986) 32–53.

[56] COLE, R. and U. VISHKIN, Approximate and exact parallel scheduling with applications to list, tree and graph problems, in: *Proc. 27th Ann. IEEE Symp. on Foundations of Computer Science* (1986) 478–491.

[57] COLE, R. and U. VISHKIN, The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, *Algorithmica* **3** (1988) 329–346.

[58] COLE, R. and U. VISHKIN, Approximate parallel scheduling, part I: the basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput* **17** (1988) 128–142.

[59] COLE, R. and C.K. YAP, A parallel median algorithm, *Inf. Proc. Letters* **20** (1985) 137–139.

[60] COOK, S.A., An observation on time-storage trade-off, *J. Comput. System Sci.* **9** (1974) 308–316.

[61] COOK, S.A., Towards a complexity theory of synchronous parallel computation, *Enseign. Math.* **27** (1981) 99–124.

[62] COOK, S.A., A taxonomy of problems with fast parallel algorithms, *Inform. and Control* **64** (1985) 2–22.

[63] COOK, S.A., C. DWORK and R. REISCHUK, Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM J. Comput.* **15** (1986) 87–97.

[64] COOK, S.A. and R.A. RECKHOW, Time-bounded random access machines, *J. Comput. System Sci.* **7** (1973) 354–375.

[65] COPPERSMITH, D., P. RAGHAVAN and M. TOMPA, Parallel graph algorithms that are efficient on the average, in: *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 260–270.

[66] COPPERSMITH, D. and S. WINOGRAD, Matrix multiplication via arithmetic progressions, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 1–6.

[67] CSANKY, L., Fast parallel matrix inversion algorithms, *SIAM J. Comput.* **5** (1976) 618–623.

[68] DAHLHAUS, E. and M. KARPINSKI, The matching problem for strongly chordal graphs is in NC, Tech. Report 855-CS, Institut für Informatik, Universität Bonn, 1986.

[69] DOBKIN, D., R.J. LIPTON and S. REISS, Linear programming is log-space hard for P, *Inf. Proc. Letters* **8** (1979) 96–97.

[70] DOLEV, D., C. DWORK, N. PIPPENGER and A. WIGDERSON, Superconcentrators, generalizers and generalized connectors with limited depth, in: *Proc: 15th Ann. ACM Symp. on Theory of Computing* (1983) 42–51.

[71] DWORK, C., P.C. KANELLAKIS and J.C. MITCHELL, On the sequential nature of unification, *J. Logic Programming* **1** (1984) 35–50.

[72] DWORK, C., P.C. KANELLAKIS and L. STOCKMEYER, Parallel algorithms for term matching, *SIAM J. Comput* **17** (1988) 711–731.

[73] DYMOND, P.W. and S.A. COOK, Hardware complexity and parallel complexity, in: *Proc. 21st Ann. IEEE Symp. on Foundations of Computer Science* (1980) 360–372.

[74] DYMOND, P.W. and W.R. RUZZO, Parallel RAMs with owned global memory and deterministic context-free language recognition, in: *Proc. 13th Internat. Coll. on Automata Languages and Programming* (1986) 95–104.

[75] EBERLY W., Very fast parallel matrix and polynomial arithmetic, in: *Proc. 25th Ann. IEEE Symp. on Foundations of Computer Science* (1984) 21–30.

[76] ECKSTEIN, D.M., Parallel processing using depth-first search and breadth-first search, Ph.D. Thesis, Dept. of Computer Science, Univ. of Iowa, Iowa City, Iowa, 1977.

[77] ECKSTEIN, D.M., Simultaneous memory access. Tech. Report TR-79-6, Computer Science Dept., Iowa State Univ., Ames, IA, 1979.

[78] EPPSTEIN, D. and Z. GALIL, Parallel algorithmic techniques for combinatorial computation, Manuscript, Dept. of Computer Science, Columbia Univ., New York, 1988.

[79] EVEN, S., *Graph Algorithms* (Computer Science Press, Potomac, MD, 1979).

[80] FICH, F.E., New bounds for parallel prefix circuits, in: *Proc. 15th Ann. ACM Symp. on Theory of Computing* (1983) 27–36.

[81] FICH, F.E., F. MEYER AUF DER HEIDE and A. WIGDERSON, Lower bounds for parallel random-access machines with unbounded shared memory, Advances in Computing Research, Vol. 4 (JAI Press, Greenwich, CT, 1987) 1–15.

[82] FICH, F.E., P. RAGDE and A. WIGDERSON, Simulations among concurrent-write models of parallel computation, *Algorithmica* **3** (1988) 43–51.

[83] FICH, F.E., P. RAGDE and A. WIGDERSON, Relations between concurrent-write models of parallel computation, *SIAM J. Comput.* **17** (1988) 606–627.

[84] FICH, F.E. and M. TOMPA, The parallel complexity of exponentiating polynomials over finite fields, *J. ACM* **35** (1988) 651–667.

[85] FORTUNE, S. and J. WYLLIE, Parallelism in random access machines, in: *Proc. 10th Ann. ACM Symp. on Theory of Computing* (1978) 114–118.

[86] FURST, M., J.B. SAXE and M. SIPSER, Parity, circuits and the polynomial time hierarchy, *Math. Systems Theory* **17** (1984) 13–28.

[87] GABOW, H.N. and R.E. TARJAN, Almost-optimum speed-ups of algorithms for bipartite matching and related problems, in: *Proc. 20th Ann. ACM Symp. on Theory of Computing* (1988) 514–527.

[88] GALIL, Z., Optimal parallel algorithms for string matching, *Inform. and Control* **67** (1985) 144–157.

[89] GALIL, Z. and V. PAN, Improved processor bounds for algebraic and combinatorial problems in RNC, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 490–495.

[90] VON ZUR GATHEN, J., Parallel algorithms for algebraic problems, *SIAM J. Comput.* **13** (1984) 802–824.

[91] VON ZUR GATHEN, J., Parallel arithmetic computations: a survey, in: *Proc. 12th. Internat. Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, Vol. 233 (Springer, Berlin, 1986) 93–113.

[92] VON ZUR GATHEN, J., Computing powers in parallel, *SIAM J. Comput.* **16** (1987) 930–945.

[93] VON ZUR GATHEN, J., Algebraic complexity theory, *Ann. Rev. Comp. Sci.* **3** (1988) 317–347.

[94] GAZIT, H., An optimal randomized parallel algorithm for finding connected components in a graph, in: *Proc. 27th Ann. IEEE Symp. on Foundations of Computer Science* (1986) 492–501.

[95] GAZIT, H. and G.L. MILLER, A parallel algorithm for finding a separator in planar graphs, in: *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 238–248.

[96] GAZIT, H. and G.L. MILLER, An improved parallel algorithm that computes the bfs numbering of a directed graph, *Inf. Proc. Letters* **28** (1988) 61–65.

[97] Gazit, H., G.L. Miller and S.H. Teng, Optimal tree contraction in the EREW model, in: S.K. Tewksbury, B.W. Dickinson and S.C. Schwartz, eds., *Concurrent Computations: Algorithms, Architecture, and Technology* (Plenum, New York, 1988) 139–156.

[98] Gibbons, A.M. and W. Rytter, An optimal parallel algorithm for dynamic expression evaluation and its applications, in: *Proc. 6th Conf. on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, Vol. 241 (Springer, Berlin, 1986) 453–469.

[99] Gibbons, A.M. and W. Rytter, *Efficient Parallel Algorithms* (Cambridge Univ. Press, Cambridge, UK, 1988).

[100] Gibbons, P., Personal communication, 1987.

[101] Gibbons, P., R.M. Karp, G. Miller and D. Soroker, Subtree isomorphism is in Random NC, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 43–52.

[102] Goldberg, A.V., S.A. Plotkin and G.E. Shannon, Parallel symmetry-breaking in sparse graphs, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 315–324.

[103] Goldberg, A.V., S.A. Plotkin and P.M. Vaidya, Sublinear-time parallel algorithms for matching and related problems, in: *Proc. 29th Ann. IEEE Symp. on Foundations of Computer Science* (1988) 174–185.

[104] Goldberg, M. and T. Spencer, A new parallel algorithm for the maximal independent set problem, in: *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 161–165.

[105] Goldschlager, L.M., The monotone and planar circuit value problems are log space complete for P, *SIGACT News* **9** (1977) 25–29.

[106] Goldschlager, L.M., A unified approach to models of synchronous parallel machines, *J. ACM* **29** (1982) 1073–1086.

[107] Goldschlager, L.M., R.A. Shaw and J. Staples, The maximum flow problem is log space complete for P, *Theoret. Comput. Sci.* **21** (1982) 105–111.

[108] Goodrich, M.T., Triangulating a polygon in parallel, *J. Algorithms* **10** (1989) 327–351.

[109] Grigoriev, D.Y. and M. Karpinski, The matching problem for bipartite graphs with polynomially bounded permanents is in NC, in: *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 166–172.

[110] Grolmusz, V. and P. Ragde, Incomparability in parallel computation, in: *Proc: 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 89–98.

[111] Hagerup, T., Towards optimal parallel bucket sorting, *Inform. and Comput.* **75** (1987) 39–51.

[112] Hagerup, T., Optimal parallel algorithms on planar graphs, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 24–32.

[113] Hastad, J., Almost optimal lower bounds for small depth circuits, in: *Proc. 18th Ann. ACM Symp. on Theory of Computing* (1986) 6–20.

[114] Hastad, J., *Computational Limitations for Small Depth Circuits* (MIT Press, Cambridge, MA, 1986).

[115] He, X. and Y. Yesha, Binary tree algebraic computation and parallel algorithms for simple graphs, *J. Algorithms* **9** (1988) 92–113.

[116] He, X. and Y. Yesha, A nearly optimal parallel algorithm for constructing depth-first spanning trees in planar graphs, *SIAM J. Comput.* **17** (1988) 486–492.

[117] Helmbold, D. and E. Mayr, Two-processor scheduling is in NC, in: *VLSI Algorithms and Architectures, Proc. Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 227 (Springer, Berlin, 1986) 12–25.

[118] Hirschberg, D.S., A.K. Chandra and D.V. Sarwate, Computing connected components on parallel computers, *Comm. ACM* **22** (1979) 461–464.

[119] Hoover, H.J., M.M. Klawe and N.J. Pippenger, Bounding fan-out in logical networks, *J. ACM* **31** (1984) 13–18.

[120] Ibarra, O.H., T. Jiang, B. Ravikumar and J.H. Chang, On some languages in NC^1, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 64–73.

[121] Ibarra, O.H., S. Moran and L.E. Rosier, A note on the parallel complexity of computing the rank of order n matrices, *Inf. Proc. Letters* **11** (1980) 162.

[122] JA'JA', J. and S.R. KOSARAJU, Parallel algorithms for planar graph isomorphism and related problems, *IEEE Trans. on Circuits and Systems*, **35** (1988) 304–311.

[123] JA'JA', J. and J. SIMON, Parallel algorithms in graph theory: planarity testing, *SIAM J. Comput.* **11** (1982) 314–328.

[124] JOHNSON, D.B., Parallel algorithms for minimum cuts and maximum flows in planar networks, *J. ACM* **34** (1987) 950–967.

[125] JONES, N.D. and W.T. LAASER, Complete problems for deterministic polynomial time, *Theoret. Comput. Sci.* **3** (1977) 105–117.

[126] KALTOFEN, E., Uniform closure properties of P-computable functions, in: *Proc. 18th Ann. ACM Symp. on Theory of Computing* (1986) 330–337.

[127] KALTOFEN, E., M.S. KRISHNAMOORTHY and B.D. SAUNDERS, Fast parallel computation of Hermite and Smith forms of polynomial matrices, *SIAM J. Alg. Discrete Methods* **8** (1987) 683–690.

[128] KALTOFEN, E. and B.D. SAUNDERS, Parallel algorithms for matrix normal forms, Report No. 88-6, Dept. of Computer Science, RPI, Troy, NY, 1988.

[129] KANEVSKY, A. and V. RAMACHANDRAN, Improved algorithms for graph four-connectivity, in: *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 252–259.

[130] KANNAN, R., G.L. MILLER and L. RUDOLPH, Sublinear parallel algorithm for computing the greatest common division of two integers, in: *Proc. 25th Ann. IEEE Symp. on Foundations of Computer Science* (1984) 7–11.

[131] KAO, M., All graphs have cycle separators and planar directed depth-first search is in DNC, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 53–63.

[132] KAO, M. and G. SHANNON, Local reorientation, global order, and planar topology, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989) 286–296.

[133] KARLIN, A.R. and E. UPFAL, Parallel hashing—an efficient implementation of shared memory in: *Proc. 18th Ann. ACM Symp. on Theory of Computing* (1986) 160–168.

[134] KARLOFF, H.J., A Las-Vegas RNC algorithm for maximum matching, *Combinatorica* **6** (1986) 387–392.

[135] KARLOFF, H.J. and W.L. RUZZO, The iterated mod problem, *Inform. and Comput.* **80** (1989) 193–204.

[136] KARLOFF, H.J. and D.B. SHMOYS, Efficient parallel algorithms for edge coloring problems, *J. Algorithms* **8** (1987) 39–52.

[137] KARP, R.M. and M.O. RABIN, Efficient randomized pattern-matching algorithms, *IBM J. Res. Dev.* **31** (1987) 249–260.

[138] KARP, R.M., E. UPFAL and A. WIGDERSON, Constructing a perfect matching is in random NC, *Combinatorica* **6** (1986) 35–48.

[139] KARP, R.M., E. UPFAL and A. WIGDERSON, The complexity of parallel search, *J. Comput. System. Sci.* **36**(2) (1988) 225–253.

[140] KARP, R.M. and A. WIGDERSON, A fast parallel algorithm for the maximal independent set problem, *J. ACM* **32** (1985) 762–773.

[141] KASAMI, T., An efficient recognition and syntax—analysis algorithm for context-free languages, Science Report AFCRL-65-758, Air Force Cambridge Research Lab., Bedford, MA, 1965.

[142] KLEIN, P.N., Efficient parallel algorithms for chordal graphs, in: *Proc. 29th Ann. IEEE Symp. on Foundations of Computer Science* (1988) 150–161.

[143] KLEIN, P.N. and J.H. REIF, An efficient parallel algorithm for planarity, in: *Proc. 27th Ann. IEEE Symp. on Foundations of Computer Science* (1986) 465–477.

[144] KLEIN, P.N. and J.H. REIF, Parallel time O(log n) acceptance of deterministic CFLs on an exclusive-write P-RAM, *SIAM J. Comput.* **17** (1988) 463–485.

[145] KOSARAJU, S.R. and A.L. DELCHER, Optimal parallel evaluation of tree-structured computations by raking, in: *VLSIAlgorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 101–110.

[146] KOZEN, D., U.V. VAZIRANI and V.V. VAZIRANI, NC algorithms for comparability graphs, interval graphs, and testing for unique perfect matching, in: *Proc. 5th Conf. on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, Vol. 206 (Springer, Berlin, 1985) 496–503.

[147] KOZEN, D. and C.-K. YAP, Algebraic cell decomposition in NC, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 515–521.

[148] KUČERA, L., Parallel computation and conflicts in memory access, *Inf. Proc. Letters* **14** (1982) 93–96.

[149] LADNER, R.E., The circuit value problem is log space complete for P, *SIGACT News* 7(2) (1975) 18–20.

[150] LADNER, R.E. and M.J. FISCHER, Parallel prefix computation, *J. ACM* **27** (1980) 831–838.

[151] LANDAU, G.M. and U. VISHKIN, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in: *Proc. 18th Ann ACM Symp. on Theory of Computing* (1986) 220–230.

[152] LEV, G.F., N. PIPPENGER and L.G. VALIANT, A fast algorithm for routing in permutation networks, *IEEE Trans. Comput.* **30** (1981) 93–100.

[153] LI, M. and Y. YESHA, New lower bounds for parallel computation, *Inform. and Comput.* **73** (1987) 102–128.

[154] LINGAS, A. and M. KARPINSKI, Subtree isomorphism and bipartite perfect matching are mutually NC-reducible, Report No. 856-CS, Institut für Informatik, Universität Bonn, 1986.

[155] LITOW, B.E. and G.I. DAVIDA, O(log(n)) parallel time finite field inversion, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 74–80.

[156] LOVÁSZ, L., Computing ears and branchings in parallel, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 464–467.

[157] LUBY, M., A simple parallel algorithm for the maximal independent set problem, *SIAM J. Comput.* **15** (1986) 1036–1053.

[158] LUBY, M., Removing randomness in parallel computation without a processor penalty, in: *Proc. 29th Ann. IEEE Symp. on Foundations of Computer Science* (1988) 162–173.

[159] LUEKER, G.S., N. MEGIDDO and V. RAMACHANDRAN, Linear programming with two variables per inequality in poly-log time, in: *Proc. 18th Ann. ACM Symp. on Theory of Computing* (1986) 196–205.

[160] LUKS, E.M., Parallel algorithms for permutation groups and graph isomorphism, in: *Proc. 27th Ann. IEEE Symp. on Foundations of Computer Science* (1986) 292–302.

[161] LUKS, E.M. and P. MCKENZIE, Fast parallel computation with permutation groups, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 505–514.

[162] MAON, Y., B. SCHIEBER and U. VISHKIN, Parallel ear decomposition search (EDS) and *st*-numbering in graphs, *Theoret. Comput. Sci.* **47** (1986) 277–298.

[163] MAYR, E.W., The dynamic tree expression problem, in: S.K. Tewksbury, B.W. Dickinson and S.C. Schwartz, eds. *Concurrent Computations: Algorithms, Architecture, and Technology* (Plenum, New York, 1988) 157–180.

[164] MAYR, E.W. and A. SUBRAMANIAN, The complexity of circuit value and network stability, in: *Proc. 4th Ann. Conf. on Structure in Complexity Theory* (1989) 114–123.

[165] MCKENZIE, P. and S.A. COOK, The parallel complexity of the abelian permutation group membership problem, in: *Proc. 24th Ann. IEEE Symp. on Foundations of Computer Science* (1983) 154–161.

[166] MEHLHORN, K. and U. VISHKIN, Randomized and deterministic simulation of PRAMs by parallel machines with restricted granularity of parallel memories, *Acta Inform.* **21** (1984) 339–374.

[167] MEYER AUF DER HEIDE, F. and A. WIGDERSON, The complexity of parallel sorting, *SIAM J. Comput.* **16** (1987) 100–107.

[168] MILLER, G.L. and V. RAMACHANDRAN, Efficient parallel ear decomposition with applications, Manuscript, MSRI, Berkeley, CA, 1986.

[169] MILLER, G.L. and V. RAMACHANDRAN, A new graph triconnectivity algorithm and its parallelization, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 335–344.

[170] MILLER, G.L., V. RAMACHANDRAN and E. KALTOFEN, Efficient parallel evaluation of straight-line code and arithmetic circuits, *SIAM J. Comput.* **17** (1988) 687–695.

[171] MILLER, G.L. and J.H. REIF, Parallel tree contraction and its application, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 478–489.

[172] MILLER, G.L. and S. TENG, Dynamic parallel complexity of computational circuits, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 254–263.

[173] MIYANO, S., The lexicographically first maximal subgraph problems: P-completeness and NC algorithms, Manuscript, Universität Paderborn, Paderborn, Fed. Rep. Germany, 1986.

[174] MULMULEY, K., A fast parallel algorithm to compute the rank of a matrix over an arbitrary field, *Combinatorica* **7** (1987) 101–104.

[175] MULMULEY, K., U.V. VAZIRANI and V.V. VAZIRANI, Matching is as easy as matrix inversion, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 345–354.

[176] NATH, D. and S.N. MAHESHWARI, Parallel algorithms for the connected components and minimal spanning tree problems, *Inf. Proc. Letters* **14** (1982) 7–11.

[177] NISAN, N., CREW PRAMs and decision trees, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989) 327–335.

[178] NISAN, N. and D. SOROKER, Parallel algorithms for zero-one supply-demand problems, *SIAM J. Discrete Math.* **2** (1989) 108–125.

[179] OFMAN, YU., On the algorithmic complexity of discrete functions, English translation in: *Sov. Phys. Dokl.* **7** (1963) 589–591; original in: *Dokl. Akad. Nauk SSSR* **145** (1963) 48–51.

[180] PAN, V., Fast and efficient algorithms for the exact inversion of integer matrices, in: *Proc. 5th Ann. Conf. on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, Vol. 206 (Springer, Berlin, 1985) 504–521.

[181] PAN, V. and J. REIF, Efficient parallel solution of linear systems, in: *Proc. 17th Ann. ACM Symp. on Theory of Computing* (1985) 143–152.

[182] PARBERRY, I., *Parallel Complexity Theory* (Pitman, London, 1987).

[183] PATERSON, M.S., Improved sorting networks with O(log N) depth, Research Report 89, Dept. of Computer Science, Univ. of Warwick, Coventry, 1987.

[184] PIPPENGER, N., On simultaneous resource bounds, in: *Proc. 20th Ann. IEEE Symp. on Foundations of Computer Science* (1979) 307–311.

[185] PIPPENGER, N., The complexity of computations by networks, *IBM J. Res. Dev.* **31** (1987) 235–243.

[186] PIPPENGER, N., Communication networks, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A* (North-Holland, Amsterdam, 1990) 805–833.

[187] PRATT, V.R. and L.J. STOCKMEYER, A characterization of the power of vector machines, *J. Comput. System Sci.* **12** (1976) 198–221.

[188] PREPARATA, F.P. and D.V. SARWATE, An improved parallel processor bound in fast matrix inversion, *Inf. Proc. Letters* **7** (1978) 148–150.

[189] RAMACHANDRAN, V., Fast parallel algorithms for reducible flow graphs, in: S.K. Tewksbury, B.W. Dickinson and S.C. Schwartz, eds., *Concurrent Computations: Algorithms, Architecture, and Technology* (Plenum, New York, 1988) 117–138.

[190] RAMACHANDRAN, V. and U. VISHKIN, Efficient parallel triconnectivity in logarithmic time, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 33–42.

[191] RANADE, A.G., How to emulate shared memory, in: *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science* (1987) 185–194.

[192] REGHBATI (ARJOMANDI), E. and D.G. CORNEIL, Parallel computations in graph theory, *SIAM J. Comput.* **7** (1978) 230–237.

[193] REIF, J.H., Parallel algorithms for graph isomorphism, Tech. Report TR-14-83, Aiken Computation Laboratory, Harvard. Univ. Cambridge, MA, 1983.

[194] REIF, J.H., Symmetric complementation, *J. ACM* **31** (1984) 401–421.

[195] REIF, J.H., Depth-first search is inherently sequential, *Inf. Proc. Letters* **20** (1985) 229–234.

[196] REIF, J.H., An optimal parallel algorithm for integer sorting, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 496–504.

[197] REIF, J.H., Logarithmic depth circuits for algebraic functions, *SIAM J. Comput.* **15** (1986) 231–242.

[198] REIF, J. and S. SEN, Polling: a new randomized sampling technique for computational geometry, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989) 394–404.

[199] REIF, J. and S. TATE, Optimal size integer division circuits, in: *Proc. 21st Ann. ACM Symp. on Theory of Computing* (1989) 264–273.

[200] REIF, J.H. and L.G. VALIANT, A logarithmic time sort for linear size networks, *J. ACM* **34** (1987) 60–76.

[201] REISCHUK, R., A fast probabilistic sorting algorithm, *SIAM J. Comput.* **14** (1985) 396–409.

[202] RUZZO, W.L., Tree-size bounded alternation, *J. Comput. System Sci.* **21** (1980) 218–235.

[203] RUZZO, W.L., On uniform circuit complexity, *J. Comput. System. Sci.* **22** (1981) 365–383.

[204] SAVAGE, J.E., *The Complexity of Computing* (Wiley, New York, 1976).

[205] SAVAGE, C. and J. JA'JA', Fast, efficient parallel algorithms for some graph problems, *SIAM J. Comput.* **10** (1981) 682–691.

[206] SAVITCH, W.J. and M.J. STIMSON, Time bounded random access machines with parallel processing, *J. ACM* **26** (1979) 103–118.

[207] SCHIEBER B. and U. VISHKIN, On finding lowest common ancestors: simplification and parallelization, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 111–123.

[208] SCHÖNHAGE, A. and V. STRASSEN, Schnelle Multiplikation grosser Zahlen, *Comput.* **7** (1971) 281–292.

[209] SCHWARTZ, J.T., Fast probabilistic algorithms for verification of polynomial identities, *J. ACM* **27** (1980) 701–717.

[210] SEIFERAS, J., Machine-independent complexity theory, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A* (North-Holland, Amsterdam, 1990).

[211] SHANKAR, N. and V. RAMACHANDRAN, Efficient parallel circuits and algorithms for division, *Inf. Proc. Letters* **29** (1988) 307–313.

[212] SHILOACH, Y. and U. VISHKIN, Finding the maximum, merging, and sorting in a parallel computation model, *J. Algorithms* **2** (1981) 88–102.

[213] SHILOACH, Y. and U. VISHKIN, An O(log n) parallel connectivity algorithm, *J. Algorithms* **3** (1982) 57–67.

[214] SIMON, J., On feasible numbers, in: *Proc. 9th Ann. ACM Symp. on Theory of Computing* (1977) 195–207.

[215] SMITH, J.R., Parallel algorithms for depth-first searches: I. planar graphs, *SIAM J. Comput.* **15** (1986) 814–830.

[216] SMOLENSKY, R., Algebraic methods in the theory of lower bounds for Boolean circuit complexity, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing* (1987) 77–82.

[217] SNIR, M., On parallel searching, *SIAM J. Comput.* **14** (1985) 688–708.

[218] SOROKER, D., Fast parallel algorithms for finding hamiltonian paths an cycles in tournaments, *J. Algorithms* **9** (1988) 276–286.

[219] SOROKER, D., Fast parallel strong orientation of mixed graphs and related augmentation problems, *J. Algorithms* **9** (1988) 205–223.

[220] STOCKMEYER, L. and U. VISHKIN, Simulation of parallel random access machines by circuits, *SIAM J. Comput.* **13** (1984) 409–422.

[221] TARJAN, R.E. and U. VISHKIN, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14** (1985) 862–874.

[222] TRAHAN, J., Instruction sets for parallel random access machines, Ph. D. Thesis, Coordinated Science Lab., Univ. of Illinois, Urbana, IL, 1988.

[223] TRAHAN, J., V. RAMACHANDRAN and M.C. LOUI, The power of parallel random access machines with augmented instruction sets, in: *Proc. Fourth Ann. Conf. on Structure in Complexity Theory* (1989) 97–103.

[224] TSIN, Y.H. and F.Y. CHIN, Efficient parallel algorithms for a class of graph theoretic problems, *SIAM J. Comput.* **13** (1984) 580–598.

[225] TUTTE, W.T., How to draw a graph, *Proc. London Math. Soc.* **13** (1963) 743–767.

[226] ULLMAN, J.D. and A. VAN GELDER, Parallel complexity of logical query programs, *Algorithmica* **3** (1988) 5–42.

[227] UPFAL, E., A probabilistic relation between desirable and feasible models of parallel computation, in: *Proc. 16th Ann. ACM Symp. on Theory of Computing* (1984) 258–1265.

[228] UPFAL, E. and A. WIGDERSON, How to share memory in a distributed system, *J. ACM* **34** (1987) 116–127.

[229] VALIANT, L.G., Parallelism in comparison problems, *SIAM J. Comput.* **4** (1975) 348–355.

[230] VALIANT, L.G., Parallel computation, in: *Proc. 7th IBM Symp. on Mathematical Foundations of Computer Science* (1982) 173–189.

[231] VALIANT, L.G., S. SKYUM, S. BERKOWITZ and C. RACKOFF, Fast parallel computation of polynomials using few processors, *SIAM J. Comput.* **12** (1983) 641–644.

[232] VAZIRANI, V.V., NC algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems, Computer Science Dept., Cornell Univ., Ithaca, NY, 1987.

[233] VENKATESWARAN, H. and M. TOMPA, A new pebble game that characterizes parallel complexity classes, in: *Proc. 27th Ann. IEEE Symp. on Foundations of Computer Science* (1986) 348–360.

[234] VISHKIN, U., Implementation of simultaneous memory address access in models that forbid it, *J. Algorithms* **4** (1983) 45–50.

[235] VISHKIN, U., Randomized speed-ups in parallel computation, in: *Proc. 16th Ann. ACM Symp. on Theory of Computing* (1984) 230–239.

[236] VISHKIN, U., On efficient parallel strong orientation, *Inf. Proc. Letters* **20** (1985) 235–240.

[237] VISHKIN, U., Optimal parallel pattern matching in strings, *Inform. and Control* **67** (1985) 91–113.

[238] VISHKIN, U. and A. WIGDERSON, Trade-offs between depth and width in parallel computation, *SIAM J. Comput.* **14** (1985) 303–314.

[239] VITTER, J.S. and R.A. SIMONS, New classes for parallel complexity: a study of unification and other complete problems for P, *IEEE Trans. on Computers* **35** (1986) 403–418.

[240] WALLACE, C.S., A suggestion for a fast multiplier, *IEEE Trans. Comput.* **13** (1964) 14–17.

[241] WHITNEY, H., Non-separable and planar graphs, *Trans. Amer. Math. Soc.* **34** (1932) 339–362.

[242] WYLLIE, J.C., The complexity of parallel computations, Ph.D. Dissertation, Computer Science Dept., Cornell Univ., Ithaca, NY, 1981.

[243] YAO, A.C., Separating the polynomial-time hierarchy by oracles; Part I, in: *Proc. 26th Ann. IEEE Symp. on Foundations of Computer Science* (1985) 1–10.

[244] YOUNGER, D.H., Recognition and parsing of context-free languages in time n^3, *Inform. and Control* **10** (1967) 189–208.

[245] KRUSKAL, C.P., L. RUDOLPH and M. SNIR, A complexity theory of efficient parallel algorithms, RC 13572 (#60702), IBM Research Division, 1988.