INFORMATION AND CONTROL **70**, 32–53 (1986)

Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking

RICHARD COLE*

New York University, New York, New York 10012

AND

UZI VISHKIN†

New York University, New York, New York 10012, and Tel Aviv University, Tel Aviv, Israel

The following problem is considered: given a linked list of length n, compute the distance from each element of the linked list to the end of the list. The problem has two standard deterministic algorithms: a linear time serial algorithm, and an $O(\log n)$ time parallel algorithm using n processors. We present new deterministic parallel algorithms for the problem. Our strongest results are (1) $O(\log n \log^* n)$ time using $n/(\log n \log^* n)$ processors (this algorithm achieves optimal speed-up); (2) $O(\log n)$ time using $n \log^{(k)} n/\log n$ processors, for any fixed positive integer k. The algorithms apply a novel "random-like" deterministic technique. This technique provides for a fast and efficient breaking of an apparently symmetric situation in parallel and distributed computation. © 1986 Academic Press, Inc.

1. INTRODUCTION

The model of parallel computation used in this paper is the exclusive-read exclusive-write (EREW) parallel random access machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. An EREW PRAM does not allow simultaneous access by more than one processor to the same memory location for read or write purposes. See Vishkin (1983a) for a survey of results concerning PRAMs.

Let Seq(n) be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem being considered. Obviously, the best upper bound on the parallel time achievable

* This research was partially supported by NSF Grant DCR-84-01633 and by an IBM faculty development award.

† This research was supported by NSF grants NSF-DCR-8318874 and NSF-DCR-8413359, ONR Grant N00014-85-K-0046 and by the Applied Mathematical Sciences subprogram of the office of Energy Research, U.S. Department of Energy, under Contract DE-AC02-76ER03077.

using p processors, without improving the sequential result, is of the form $O(\mathrm{Seq}(n)/p)$. A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal.*

We present a new *deterministic coin tossing* technique for devising parallel algorithms. The technique uses the binary representation of names (numbers) for breaking a symmetric situation in a "random-like" fashion.

Let m be the size of the memory of our computer. Our technique performs well when each variable in the underlying model of computation is represented by a few bits (say $O(\log m)$ bits). Interestingly, the technique performs badly when each variable is represented by many bits (say $f(m)$ bits, where f is the inverse of $\log^*$ and $\log^* n$ is the least i such that $\log^{(i)} n \leqslant 2$, where $\log^{(i)}$ is the ith iterate of the log function). Representing each variable by $O(\log m)$ bits is in line with typical definitions of RAMs (see Aho (1974). The role of PRAMs is to extend the RAM model to express parallelism. This extension should have no effect on the number of values that each variable may assume. A variant of PRAMs (called PRAM–INFINITY) that allows each variable to assume infinitely many values has been proposed recently. The PRAM–INFINITY also allows infinitely large shared memory. This variant (or closely related ones) was used to prove lower bounds for various interesting problems; the proofs apply mathematically appealing "Ramsey-like" theorems (see Fich, Meyer auf der Heide, Ragde and Wigderson (1985); Israeli and Moran (1985); Meyer auf der Heide and Wigderson (1985).

It appears that in the transition from PRAM to PRAM–INFINITY we lose the coin tossing technique. For the technique depends crucially on the fact that each variable is represented by few bits (say $O(\log m)$ bits), while in the PRAM–INFINITY model this constraint does not exist; in fact, there is no restriction on the number of bits representing a variable. This is analogous to the loss of bucket sort when we adopt the decision tree model. (See Aho, Hopcroft, and Ullman (1974) for both an $\Omega(n \log n)$ time lower bound for sorting n elements in a decision tree model and an $O(n)$ time bucket sort algorithm.)

We show how to apply our coin tossing technique to the *list-ranking problem* defined below.

Input: A linked list of length n. It is given in an array of length n, not necessarily in the order of the linked list. Each of the n elements (except the last element in the linked list) has the array index of its successor in the linked list.

The problem: For each element, compute the number of elements following it in the linked list.

The list ranking problem is often encountered in the design of parallel

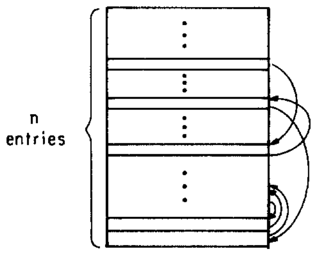

FIG. 1. The input.

algorithms. For instance, the fundamental "Euler tour technique" for computing various tree functions (see Tarjan and Vishkin (1985); Vishkin (1985)) has the same efficiency as the new algorithm presented here.

The problem has a trivial linear time serial algorithm and a simple deterministic parallel algorithm (the *standard* parallel algorithm). The standard parallel algorithm runs in time $O(\log n)$ using n processors. Wyllie (1979) conjectured that $\Omega(n)$ processors are required in order to get $O(\log n)$ time. If true, this would imply, in particular, that there is no optimal speed-up parallel algorithm for $n/\log n$ processors. Recently, Kruskal, Rudolph, and Snir (1985) presented an optimal speed-up algorithm for this problem that runs in $O(n^{\varepsilon})$ time using $n^{1-\varepsilon}$ processors, for fixed ε, $1 \geqslant \varepsilon > 0$. Vishkin (1984b) proposed the use of randomized parallel algorithms for this problem. A randomized parallel algorithm which runs in $O(n/p)$ time using $p \leqslant n/(\log n \log^* n)$ processors on an EREW PRAM was given. The probability that this will indeed be the running time converges rapidly to one as n grows. In particular, this optimal speed-up algorithm runs in "about" $O(\log n)$ time using "about" $n/\log n$ processors.

In this paper we present new deterministic parallel algorithms. Our strongest results are:

1. $O(\log n \log^* n)$ time using $n/(\log n \log^* n)$ processors. This algorithm achieves optimal speed-up.

2. $O(\log n)$ time using $n \log^{(k)} n/\log n$ processors, for any fixed positive integer k, thereby showing that Wyllie's conjecture is incorrect. In the above, $\log^{(k)}$ denotes the kth iterate of the log function (e.g. $\log^{(3)} n = \log\log\log n$).

Recently, the new deterministic coin tossing technique has been applied to obtain new, efficient parallel algorithms for computing connected and biconnected components and minimum spanning trees (Cole and Vishkin, 1986).

The next section presents the new deterministic coin tossing technique for breaking an (apparently) symmetric situation. Among other things, Section 3 reviews an optimal speed-up deterministic parallel algorithm that uses balanced trees. The algorithm is used later for two purposes: (1) as a subroutine, and (2) to explain the new list ranking algorithm. The new algorithm essentially grafts the new technique onto the framework of the balanced tree algorithm. In Section 4 we describe the basic version of our algorithm that runs in time $O(\log n \log\log n)$ using $n/(\log n \log\log n)$ processors. This algorithm achieves optimal speed-up; it will be quite adequate for all practical purposes. In Section 5 we describe the faster optimal algorithms and our other results.

2. The Deterministic Coin Tossing Technique

2.1. *The Basic Technique*

We illustrate the deterministic coin tossing technique by using it to break the (apparently) symmetric situation that arises in the following problem.

Input: A connected directed graph $G(V, E)$. The in-degree of each vertex is exactly one. The out-degree of each vertex is exactly one. Such a graph is called a *ring* since it forms a directed circuit. Let $n = |V|$. We define a subset U of V to be an *r-ruling set of G if*:

(1) No two vertices of U are adjacent.

(2) For each vertex v in V there is a directed path from v to some vertex in U whose edge length is at most r.

The r-ruling set problem: Find an r-ruling set of V.

In order to demonstrate our basic technique we give an $O(1)$ time algorithm using n processors for the $\lceil \log n \rceil$-ruling set problem. The algorithm is given for the EREW PRAM. In Section 2.2 we present a recursive application of the technique. It leads to an $O(k)$ time algorithm using n processors for the $\lceil \log^{(k)} n \rceil$-ruling set problem. In particular, it provides an $O(\log^* n)$ time algorithm using n processors for the 2-ruling set problem. In Section 2.3 we describe a non-recursive approach that provides an $O(\log n)$ time algorithm using $n/\log n$ processors for the 2-ruling set problem.

Assumptions about the input representation. The vertices are given in an array of length n. The entries of the array are numbered from 0 to $n-1$. The numbers are represented as binary strings of length $\lceil \log n \rceil$. We refer to each binary symbol (bit) of this representation by a number between 0 and $\lceil \log n \rceil - 1$. The rightmost (least significant) bit is called bit number 0 and the leftmost bit is called bit number $\lceil \log n \rceil - 1$. Each vertex has a pointer to the next vertex in the ring (representing its outgoing edge). For simplicity we assume that $\log n$ is an integer.[1]

Here is a verbal description of an algorithm for the $\log n$-ruling set problem. The algorithm is given later. Processor i, $0 \leqslant i \leqslant n-1$, is assigned to entry i of the input array(for simplicity, entry i is called vertex i). It will attach the number i to vertex i. So, the *present* "serial" number of vertex i, denoted $\text{SERIAL}_0(i)$, is i. Next, we attach to vertex i a new serial number, denoted $\text{SERIAL}_1(i)$, as follows. Let i_2 be the vertex following i. (That is (i, i_2) is in E.) Let j be "the index of the rightmost bit in which i and i_2 differ." Processor i assigns j to $\text{SERIAL}_1(i)$.

EXAMPLE. Let i be ...010101 and i_2 be ...111101. The index of the rightmost bit in which i and i_2 differ is 3 (recall the rightmost bit has number 0). Therefore, $\text{SERIAL}_1(i)$ is 3.

Remark (due to B. Schieber). j can be computed by a constant number of standard operations, as follows. Without loss of generality suppose $i \geqslant i_2$ (otherwise interchange the two numbers). Set $h = i - i_2$, and $k = h - 1$. (So h has a 1 for bit number j, and a 0 for bits of lesser significance, while k has a 0 for bit number j, and a 1 for bits of lesser significance; also, h and k agree on the bits of higher significance.) Compute $l = h \oplus k$, where $\oplus$ is the exclusive-or operation. We observe l is the unary representation of $j+1$. So it just remains to convert this value from unary to binary, and then to subtract one.

Next, we show how to use the information in vector SERIAL_1 in order to find a $\log n$-ruling set.

FACT 1. For all i, $\text{SERIAL}_1(i)$ is a number between 0 and $\log n - 1$ and needs only $\lceil \log\log n \rceil$ bits for its representation. For simplicity we will assume that $\log\log n$ is an integer.

Let i_1 and i_2 be, respectively, the vertices preceding and following i. $\text{SERIAL}_1(i)$ is a local minimum if $\text{SERIAL}_1(i) \leqslant \text{SERIAL}_1(i_1)$ and $\text{SERIAL}_1(i) \leqslant \text{SERIAL}_1(i_2)$. A local maximum is defined similarly.

FACT 2. The number of vertices in the shortest path from any vertex in G to the next (vertex that provides a) local extremum (maximum or minimum), with respect to SERIAL_1, is at most $\log n$.

Observe that several local minima (or maxima) may form a "chain" of successive vertices in G. Requirement (1), in the definition of an r-ruling set, does not allow us to include all these local minima in the set of selected vertices. Our algorithm exploits the alternation property (defined below) of vector SERIAL_1 to overcome this problem.

[1] The base of all logarithms in the paper is 2.

The alternation property. Let i be a vertex and j be its successor. If bit number $\text{SERIAL}_1(i)$ of $\text{SERIAL}_0(i)$ is 0 (resp. 1), then this bit is 1 (resp. 0) in $\text{SERIAL}_0(j)$. (For $\text{SERIAL}_1(i)$ is the index of the rightmost bit on which $\text{SERIAL}_0(i)$ and $\text{SERIAL}_0(j)$ *differ.*)

Suppose that $i_1, i_2 \ldots$ is a chain in G such that $\text{SERIAL}_1(i)$ is a local minimum (resp. maximum) for every i in the chain. Then:

FACT 3. For all vertices in the chain SERIAL_1 is the same (i.e., $\text{SERIAL}_1(i_1) = \text{SERIAL}_1(i_2) = \ldots$). (By definition of local minimum).

Below, we consider bit number $\text{SERIAL}_1(i_1)$ of SERIAL_0 for all vertices in the chain.

FACT 4. The following sequence of bits is an alternating sequence of zeros and ones.

Bit number $\text{SERIAL}_1(i_1)$ of $\text{SERIAL}_0(i_1)$, bit number $\text{SERIAL}_1(i_2)(= \text{SERIAL}_1(i_1))$ of $\text{SERIAL}_0(i_2), \ldots,$ bit number $\text{SERIAL}_1(i_j)$ $(= \text{SERIAL}_1(i_1))$ of $\text{SERIAL}_0(i_j), \ldots.$

(This is readily implied by the alternation property.)

We can now understand why we called our technique deterministic coin tossing. We associated zeros and ones with the vertices, based on their original serial numbers; these serial numbers were set deterministically. This association allows us to treat (apparently) similar vertices differently. Finally, note that coin tossing can be used for similar purposes.

We return to the algorithm. We select the following subset of vertices. We select all vertices i that are local minima and satisfy one of the following two conditions:

(1) Neither of i's neighbors (the vertices adjacent to i) is a local minimum.

(2) Bit number $\text{SERIAL}_1(i)$ is 1.

We say an unselected vertex is *available* if neither of its neighbors was selected and it is a local maximum. We select all available vertices i that satisfy one of the following two properties.

(1) Neither of i's neighbors is available.

(2) Bit number $\text{SERIAL}_1(i)$ is 1.

The selected vertices form a log n-ruling set. Requirement (1) is satisfied since we never select two adjacent vertices. Requirement (2) is satisfied by

Fact 2 and since every local extremum either is selected or is a neighbor of a vertex that was selected.

Less informally we write the algorithm as follows. (Later, we will refer to this as the basic step.)

```
for Processor $i$, $0 \leqslant i \leqslant n-1$, pardo (perform in parallel)
    $\mathrm{SERIAL}_0(i) := i$
    $\mathrm{SERIAL}_1(i) :=$ "the minimal bit in which $\mathrm{SERIAL}_0(i)$ differs from
        $\mathrm{SERIAL}_0$ of the following vertex"
    if $\mathrm{SERIAL}_1(i)$ is a local minimum with respect to the two neighbors of $i$
    then if either of the following is satisfied:
            (1) neither of the vertices adjacent to $i$ is a local minimum
            (2) bit number $\mathrm{SERIAL}_1(i)$ of $\mathrm{SERIAL}_0(i)$ is 1
        then select $i$
    if neither $i$ nor any of its neighbors were selected and if $\mathrm{SERIAL}_1(i)$ is
        a local maximum with respect to the two neighbors of $i$
    then (** $i$ is available, and **) if either of the following is satisfied:
            (1) neither of the vertices adjacent to $i$ is available
            (2) bit number $\mathrm{SERIAL}_1(i)$ of $\mathrm{SERIAL}_0(i)$ is 1
        then select $i$
```

We have shown:

THEOREM 2.1. *A* $\log n$*-ruling set can be obtained in* $O(1)$ *time using* n *processors.*

Below, we show how to apply the basic step repeatedly in order to find a 2-ruling set.

2.2. *The* kth *Application of the Basic Step*

In order to prepare the input for the kth application of the basic step, we "delete" from G the vertices that where selected in the previous $k-1$ applications, their neighbors, and the edges incident to any vertex being deleted.

The input for the kth application of the basic step is the remaining graph and vector SERIAL_{k-1}. SERIAL_{k-1} will play the role played above by SERIAL_0 and a new vector SERIAL_k will play the role of SERIAL_1. The degree of each vertex in the input graph is at most 2 (if the directions of the edges are ignored). It is very simple to extend the basic step to handle vertices whose degree is $\leqslant 1$. Vertices whose degree is 2 are treated as in the basic step (unless they have a neighbor whose degree is 1). The k-th application of the basic step will be as follows. (For an explanation see Fact 5 below.)

```
for processor $i$, $0 \leqslant i \leqslant n-1$, pardo
    if vertex $i$ or one of its neighbors have been selected
        in a previous application of the basic step
    then "delete" vertex $i$ and the edges incident to it
for processor $i$, $0 \leqslant i \leqslant n-1$, such that $i$ is in the remaining graph pardo
  case 1 deg($i$) = 2
        then compute SERIAL$_k(i)$
            if the degree of each of $i$'s two neighbors is 2
            then apply the basic step to $i$
  case 2 deg($i$) = 0
        then select $i$
  case 3 deg($i$) = 1
        then if either of the following is satisfied
            (1) the degree of $i$'s neighbor is 2
            (2) $i$'s neighbor is its successor
        then select $i$
```

The following fact helps to clarify the operation of the kth application of the basic step.

FACT 5. Let i, j be adjacent in the input graph for the kth application. Then, $\mathrm{SERIAL}_{k-1}(i) \neq \mathrm{SERIAL}_{k-1}(j)$. (For $k=1$ this inequality clearly holds. We show that it also holds if $k>1$. If they were equal each of them had to be a local maximum or local minimum at the $(k-1)$st application. The selection of the ruling set implies that each local maximum or local minimum v is either selected or has a neighbor that is selected. Therefore, v must have been deleted and cannot be included in this input graph.)

FACT 6. It is easy to deduce that the output graph consists of simple paths each comprising at most $\log\log \cdots \log n$ vertices where the sequence includes k "log"s. (Again, we assume for simplicity that each application of a sequence of logs to n produces only integers.)

We finish this description with three obvious conclusions.

(1) After a total of $\log^* n$ applications we delete all vertices in the graph.

(2) The vertices that were selected form a 2-ruling set.

(3) The cardinality of a 2-ruling set (in a ring) is at least $n/3$.

If our original input is a directed path of n vertices, rather than a ring, we obtain a 2-ruling set by applying the basic step $\log^* n$ times, as above. To obtain a $\log^{(k)} n$-ruling set we apply the basic step k times.

We have shown:

THEOREM 2.2. *A $\log^{(k)} n$-ruling set can be obtained in $O(k)$ time using n processors.*

COROLLARY 2.1. *A 2-ruling set can be obtained in $O(\log^* n)$ time using n processors.*

General Remarks. (1) Readers familiar with randomized algorithms may be tempted to solve these problems using randomization. We already mentioned that Vishkin (1984b) did so for the (related) list ranking problem. Our deterministic technique was inspired by such a randomized approach.

(2) The $\lceil \log n \rceil$-ruling set algorithm is valid even for models of distributed computation that allow only local communication and do not have a shared memory like a PRAM. We do not elaborate on this.

2.3. *An Optimal 2-Ruling Set Algorithm*

First, we find a log n-ruling set using the basic step, above. Below, we describe how to add more vertices to the log n-ruling set to produce a 2-ruling set. These additional vertices are selected using the numbers SERIAL_1 associated with each vertex, as follows.

```
for $i = 0$ to $\log n - 1$ do
    for each vertex $v$ for which $\text{SERIAL}_1(v) = i$ pardo
        if $v$ is not in the ruling set and neither of the neighbors of $v$ is in
          the ruling set
        then add $v$ to the ruling set
```

Note that if $\text{SERIAL}_1(v) = i$, and if neither v nor its neighbors are in the ruling set, then neither of the neighbors w of v has $\text{SERIAL}_1(w) = i$. Thus this procedure selects a set of non-adjacent vertices. When the procedure is finished, any vertex that was not selected must have a selected vertex as a neighbor. Thus this procedure selects a 2-ruling set.

Clearly, the procedure can run in $O(\log n)$ time. At first sight, it appears to require $\Theta(n)$ processors to achieve this running time (simply assign a processor to each vertex v). We show that, in fact, this time can be achieved using only $n/\log n$ processors. To do this we perform two instructions:

INSTRUCTION 1. We sort the vertices by their SERIAL_1 number. The outcome of this sort is that each vertex v will be given a number $\text{RANK}(v)$, $1 \leq \text{RANK}(v) \leq n$. No two vertices will have the same RANK.

INSTRUCTION 2. For each v, $\text{RANK}(v) := \text{RANK}(v) + in/\log n$, where $i = \text{SERIAL}_1(v)$.

We then process the vertices in $2\log n$ rounds. In round j ($1 \leqslant j \leqslant 2\log n$), we process all vertices v such that $(j-1)\,n/\log n < \mathrm{RANK}(v) \leqslant jn/\log n$.

Instruction 2 guarantees that we never simultaneously process two vertices whose SERIAL_1 number is different.

Instruction 1 simply needs a bucket sort of n numbers in the range $[0, \log n - 1]$. The rest of this section shows how to perform such a sort in $O(\log n)$ time using $n/\log n$ processors. We remark that the bucket sort, while not performed in place, nonetheless will require only $O(n)$ space. It may be helpful to read Section 3 at this point; it reviews the prefix sum parallel algorithm, used below.

The sort proceeds in three stages. First, we count, for each number i, the number of vertices v for which $\mathrm{SERIAL}_1(v) = i$. Second, using a prefix sum sequential algorithm, we count the number of vertices v for which $\mathrm{SERIAL}_1(v) < i$, in $O(\log n)$ time. Third, for each vertex v, we determine a unique value $\mathrm{RANK}(v)$. No two vertices get the same RANK.

The first stage proceeds in two substages. First, we divide the vertices into groups of size $\log n$. For each group, in $O(\log n)$ time, using one processor per group, we count the number of vertices v for which $\mathrm{SERIAL}_1(v) = i$, $0 \leqslant i < \log n$. (We also determine, on the fly, for each vertex v, how many vertices w, preceding v in the group, satisfy $\mathrm{SERIAL}_1(w) = \mathrm{SERIAL}_1(v)$.) We obtain $n/\log n$ sets of $\log n$ counts, one set per group. Second, using a prefix sum parallel algorithm (or rather, $\log n$ of them), for each number i, we sum the $n/\log n$ associated counts (for each i, one count per group). Clearly, this stage, implemented with $n/\log n$ processors, uses $O(\log n)$ time.

The second stage is straightforward. In the third stage, for each vertex v, we compute $\mathrm{RANK}(v)$ using a single processor and $O(1)$ time, where several processors may read from the same memory location. (It is easy to simulate this computation in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM.) $\mathrm{RANK}(v)$ will be: one, plus the number of vertices u such that $\mathrm{SERIAL}_1(u) < \mathrm{SERIAL}_1(v)$ (computed in the second stage), plus the number of vertices w such that $\mathrm{SERIAL}_1(w) = \mathrm{SERIAL}_1(v)$ and w appears before v in the input array. The last number is obtained by adding the number of such vertices w that appear in groups prior to the group of v and the number of such vertices w that appear prior to v in its own group. Both numbers were computed in the first stage.

It now follows that the algorithm for bucket sort, with $\log n$ buckets, uses $n/\log n$ processors and $O(\log n)$ time. We conclude

THEOREM 2.3. *A 2-ruling set can be obtained in $O(\log n)$ time using $n/\log n$ processors.*

Remark. It is easy to modify the bucket sort algorithm to sort n numbers in the range $[0, m-1]$, $m \geqslant \log n$. The algorithm will use $n/\log n$ processors and $O(\log n \log m/\log\log n)$ time. (Each number should be represented using digits that can take on $\log n$ values; proceed as in the standard bucket sort for multi-digit numbers.) Also, for $m \geqslant t \geqslant \log n$, using n/t processors, we achieve a time of $O(t \log m/\log t)$ (replace $\log n$ by t in the above algorithm).

3. Balanced Tree Algorithms

3.1. *Preliminaries*

Theorem (Brent). *Any synchronous parallel algorithm taking time t that consists of a total of x elementary operations can be implemented by p processors within a time of $\lfloor x/p \rfloor + t$.*

Proof of Brent's Theorem. Let x_i denote the number of operations performed by the algorithm in time $i(\sum_1^t x_i = x)$. We use the p processors to "simulate" the algorithm. Since all the operations at time i can be executed simultaneously, they can be computed by the p processors in $\lceil x_i/p \rceil$ units of time. Thus, the whole algorithm can be implemented by p processors in time

$$\sum_1^t \lceil x_i/p \rceil \leqslant \sum_1^t (\lfloor x_i/p \rfloor + 1) \leqslant \lfloor x/p \rfloor + t. \quad \blacksquare$$

Remark. Brent's theorem is stated for models of computation where not all computational overheads are taken into account. Specifically, the proof of Brent's theorem poses two implementation problems. The first is to evaluate x_i at the beginning of time i in the algorithm. The second is to assign the processors to their jobs.

Recall the following standard *deterministic parallel algorithm* for the list-ranking problem (defined in the Introduction). Say that we have n processors. Assign a processor to each of the n elements. Denote the pointer of element i of the input array by $D(i)$ and initialize $R(i) := 1$, $1 \leqslant i \leqslant n$. We set $D(t) :=$ "end of list" (where t is the last element in the linked list), D ("end of list") $:=$ "end of list" and R ("end of list") $:= 0$.

```
Iterate ⌈log n⌉ times:
for processor $i$, $1 \leqslant i \leqslant n$, pardo
    $R(i) := R(i) + R(D(i))$; $D(i) := D(D(i))$ (To be called the short-cut
      operation, performed by $i$ at $D(i)$). (See Fig. 2.)
```

Note that $\Omega(n \log n)$ short-cuts are made by this algorithm. It runs in time $O((n \log n)/p + \log n)$ using p processors on an EREW PRAM and solves the list ranking problem, by placing the results in the vector R.

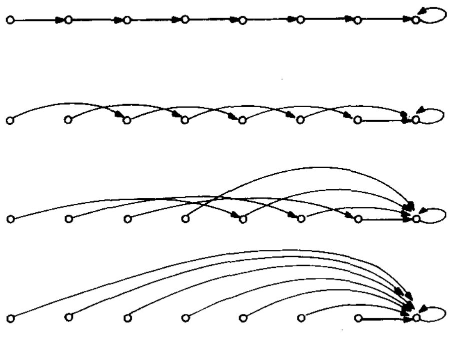

FIG. 2. The standard deterministic parallel algorithm.

Implementation Remark 1. In order to derive this running time from Brent's theorem n has to be broadcast to all p processors. This takes an additional $O(\log p)$ time.

Implementation Remark 2. As presented the algorithm is not EREW since there are concurrent reads at "end of list". This can be avoided by instructing every processor i to quit when $D(i) =$ "end of list".

3.2. *Balanced Binary Tree Parallel Algorithms.*

One simple pattern of optimal speed-up deterministic parallel algorithms uses the balanced binary tree. This pattern was used, among many others, by Wyllie (1979); Chin, Lam, and Chen (1981); Vishkin (1984a). (Apparently, Fisher, and Ladner (1980) were the first to suggest using this pattern.) Let us first demonstrate this pattern on the problems of computing sums and prefix sums.

Input: An array of n numbers $A(1)$, $A(2)$,..., $A(n)$. Assume, without loss of generality, that $\log_2 n$ is an integer.

Problem: Compute their sum.

Algorithm: "Plant" a balanced binary tree with n leaves on the array. The nodes of the tree at level h are denoted $[h, j]$, $1 \leqslant j \leqslant 2^{\log n - h}$. See Fig. 3. Leaf $[0, j]$ corresponds to $A(j)$. Associate a number $B[h, j]$ with node $[h, j]$ of the tree.

```
Initialization: for all $1 \leqslant j \leqslant n$ pardo $B[0, j] := A(j)$.

for $h := 1$ to $\log n$ do
    for all $1 \leqslant j \leqslant 2^{\log n - h}$ pardo $B[h, j] := B[h-1, 2j-1] + B[h-1, 2j]$.
$B[\log n, 1]$ holds the desired sum.
```

Think first about an n processor implementation of this summation algorithm. It runs in $O(\log n)$ time. Then apply the proof of Brent's

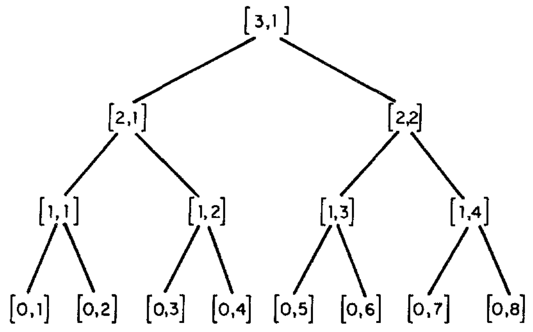

FIG. 3. The balanced binary tree.

theorem to get an alternate implementation that uses only $n/\log n$ processors and runs in $O(\log n)$ time. This summation algorithm can be extended to solve the following prefix sum problem.

Input: Same as for the summation problem.

Problem: Compute $\sum_1^i A(j)$ for all $1 \leqslant i \leqslant n$.

Algorithm: Perform the summation algorithm given above, thereby obtaining all the B values. An additional "down-sweep" of the tree (from the root to the leaves), which roughly amounts to reversing the operation of the summation algorithm, will complete the job.

Associate another number $C[h, j]$ with each node $[h, j]$.

Initialization: $C[\log n, 1] := 0$.

```
for $h := \log n - 1$ downto 0 do
        for all $1 \leqslant j \leqslant 2^{\log n - h}$ pardo
                if $j$ is odd
                then $C[h, j] := C[h+1, (j+1)/2]$
                else $C[h, j] := C[h+1, j/2] + B[h, j-1]$.
for all $1 \leqslant j \leqslant n$ pardo $C[0, j] := C[0, j] + B[0, j]$.
```

$C[0, j]$, $1 \leqslant j \leqslant n$, hold the desired prefix sums. This algorithm can also be implemented to run in $O(n/p + \log n)$ time using p processors on an EREW PRAM. (Apply Brent's theorem and Implementation Remark 1.)

A wishful thought. We want to find an algorithm for the list ranking problem that performs a total of $O(n)$ short-cuts. If we could "plant" a balanced binary tree in our linked list (in the order of the linked list) it would solve our problem: enter a one at each leaf and apply the prefix sum algorithm. A closer look at the summation part of such a prefix sum computation reveals the following:

The operation of the **for** statement (of the summation algorithm) for

$h = 1$ corresponds to short-cuts at every odd location in the linked list. This results in a new linked list that connects only the even locations of the original list, thereby halving its length. Then, the **for** statement for $h = 2$ corresponds to short-cuts at odd locations of the new linked list, and so on. See Fig. 4. Observe that the **for** statement of the summation algorithm never performs a short-cut at two successive elements of the linked list at hand; and, therefore, the "input" to any operation of this **for** statement is a single linked list.

Remark. The problem, of course, is that we do not know how to plant a balanced binary tree with respect to the linked list without actually first solving the list ranking problem itself, since this "planting" needs the ranking mod 2, mod 4, mod 8,... as explained above.

Each operation of the **for** statement has the following two features.

(1) The output is a single list whose length is half the length of the input.

(2) It takes $O(1)$ parallel time to execute.

We will use an algorithm which approximates these two features. In our new algorithm we plant an "approximately balanced tree" (it will be a 2–3 tree). Each leaf of the tree corresponds to an element of the list, and each level of the tree corresponds to an iteration of the **for** statement. For a given level of the tree, the nodes at this level correspond to those elements of the list over which shortcuts have not yet been made (by iterations of the **for** statement corresponding to lower levels of the tree). For each level of the tree we divide the elements of the list (corresponding to nodes at this level) into two sets: those that are shortcut (by the corresponding iteration of the **for** statement), called *victims*, and those that are not shortcut (called *survivors*). In order to approximately achieve properties (1) and (2) above, we require these two sets to meet the following two constraints:

(a) If an element is a survivor then its successor (if any) is a victim.

(b) One, at least, of every three adjacent elements is a survivor.

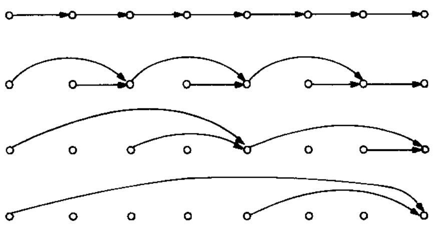

FIG. 4. A "short-cut analogy" to the balanced binary tree algorithm.

By (a) at most one half of the elements are survivors. By (b) each survivor need perform at most two shortcut operations to remove all the victims from the list. Hence in $O(1)$ parallel time (using n processors) we obtain a single linked list containing at most half as many elements (assuming we can separate the elements into survivors and victims).

But a 2-ruling set provides an appropriate set of survivors!

4. The Basic List Ranking Algorithm

Initialization: $m := n$. As in the standard deterministic algorithm, denote the pointer of element i by $D(i)$ and initialize $R(i) := 1$, $0 \leqslant i \leqslant n-1$.

The algorithm which is given later should be read together with the commentary below. The purpose of the **while** loop of the algorithm is to "thin out" the input linked list into a list of length $\leqslant n/\log n$. The input to each iteration of the **while** loop is a linked list of length m stored in an array of length m. Vector D contains, for each element, the next element in this linked list.

The purpose of Step 2 is to enter either the value 1 or the value 0 into RULING(j), for each j, $0 \leqslant j \leqslant m-1$, so that those elements with RULING(j) $= 1$, $1 \leqslant j \leqslant m$, form a 2-ruling set of the directed graph. Step 2 uses the algorithm of Section 2.3 for finding a 2-ruling set.

In Step 3 we shortcut, in parallel, over each j such that RULING(j) $= 0$. The resulting list will contain exactly those elements in the 2-ruling set, of which there are at most $m/2$. We make some further comments on the operation of this step.

(a) Each element j for which RULING(j) $= 1$ (an element of the 2-ruling set) is followed by at least one and at most two elements for which RULING is 0.

(b) Each element over which we perform a shortcut will remain with no incoming pointers. Such elements will be "deleted" in Step 4.

(c) The parameter t stands for the present time. (This parameter increases as the algorithm progresses.) The information in $OP(i, t)$ enables us, later on, to reconstruct the operation of processor i at time t. This is used in Step 6 to derive the final value of $R(D(j))$ by subtracting the present value of $R(j)$ from the final value of $R(j)$. For this reason we prefer here to name the processors performing the operations rather than to use the framework of Brent's theorem.

Step 4 contracts the input array for the present **while** loop iteration into a new array that contains exactly those elements in the new linked list.

When we arrive at Step 5, the length of the linked list at hand is

$\leqslant n/\log n$. Step 5 applies the standard parallel list ranking algorithm in order to find the ranking of each element in this linked list.

Step 6 extends the list rankings to all elements of the original linked list using the information in $OP(.,.)$.

```
$t := 1$; ($t$ is the present time)
**while** $m > n/\log n$ **do**

    **Step 1** (Initialization for the present **while** loop iteration).

        **for** $j$, $0 \leqslant j \leqslant m-1$, **pardo**
            $\mathrm{SERIAL}_0(j) := j$

    **Step 2.** Compute a 2-ruling set into vector RULING, using the
    algorithm of Section 2.3. From now on we specify for each instruction
    the processors that perform it. Suppose $p$ processors are available.
    Processor $i$, $1 \leqslant i \leqslant p$, is assigned to segment $[(i-1)\,m/p, \ldots, im/p - 1]$ of
    the array that forms the input to this **while** loop iteration. (For simplicity
    we assume that $m/p$ is an integer. Otherwise, we could assign Processor $i$
    to the segment including all the integers in the half open interval
    $((i-1)\,m/p - 1;\ im/p - 1]$.)

    **Step 3.**

    **for** Processor $i$, $1 \leqslant i \leqslant p$, **pardo**
        **for** $j := (i-1)\,m/p$ **to** $im/p - 1$ **do**
            **if** $\mathrm{RULING}(j) = 1$
            **then** $OP(i, t) := (D(j), j, R(j))$;
                $R(j) := R(j) + R(D(j))$; $D(j) := D(D(j))$(shortcut).
             **if** $\mathrm{RULING}(D(j)) = 0$
            **then** $OP(i, t) := (D(j), j, R(j))$;
                $R(j) := R(j) + R(D(j))$;
                $D(j) := D(D(j))$(shortcut).

    **Step 4.** Perform the balanced binary tree prefix-sum computation
    described in the previous section with respect to the vector RULING. As
    a result,

    (1) $m := \sum_j \mathrm{RULING}(j)$, and

    (2) each element $j$ with $\mathrm{RULING}(j) = 1$ gets its entry number in a (con-
    tracted) array of length $m$ containing the output linked list.
    (This array is the input for the next iteration (if any) of the **while** loop.)
**od**
```

Let T be the last time unit for which an assignment into $OP(\ ,\ )$ was performed.

Step 5. Apply a simulation of the standard deterministic parallel algorithm by p processors to the current array.

Step 6.

```
for Processor $i$, $1 \leqslant i \leqslant p$, pardo
    for $t := T$ downto 1 do
        $R(OP(i, t).1) := R(OP(i, t).2) - OP(i, t).3$.
        (Comment. $OP(i, t).k$, $k = 1, 2, 3$, represent the fields of $OP(i, t)$.
        If $OP(i, t)$ is undefined, the instruction is interpreted to be a null
        operation. Also, recall Comment (c) in the verbal description of
        Step 3.)
```

Implementation Remark. Each time m gets a new value, broadcast it to all processors as in Implementation Remark 1 of the previous section.

Complexity. We start by evaluating the operation and time requirements of the algorithm (so, at present, we assume that we have an unlimited number of processors available). Later, we use Brent's theorem to derive processor and time bounds. Initialization requires $O(n)$ operations and $O(1)$ time. Let us focus on one iteration of the **while** loop.

Step 1 takes $O(m)$ operations and $O(1)$ time.
Step 2 takes $O(m)$ operations and $O(\log m)$ time.
Step 3 takes $O(m)$ operations and $O(1)$ time.
Step 4 takes $O(m)$ operations and $O(\log m)$ time.

So each iteration of the **while** loop takes $O(m)$ operations and $O(\log m)$ time. Each such iteration results in a linked list whose length is $\leqslant \frac{1}{2}$ the length of the list when the iteration started. Therefore, after $O(\operatorname{loglog} n)$ iterations we get a list whose length is $\leqslant n/\log n$. Summing up the operation and time complexity of the **while** loop gives $O(n)$ operations and $O(\log n \operatorname{loglog} n)$ time.

Step 5 takes $O(n)$ operations and $O(\log n)$ time.

Step 6 requires the same number of operations and time as all the iterations of Step 3, since it follows its "footsteps".

So we have a total of $O(n)$ operations and $O(\log n \operatorname{loglog} n)$ time. Applying Brent's theorem we get $O(n/p)$ time using any number $p \leqslant n/(\log n \operatorname{loglog} n)$ of processors. We know that any such result can be alternatively stated as $O(\log n \operatorname{loglog} n)$ time using $n/(\log n \operatorname{loglog} n)$ processors. We leave the reader to verify that the implementation problems as per the remark following Brent's theorem can be readily overcome. We have shown:

THEOREM 4.1. *The list ranking problem can be solved in time $O(n/p)$ using $p \leqslant n/(\log n \operatorname{loglog} n)$ processors.*

5. The Fast Optimal Algorithm

We describe an algorithm that runs in time $O(n/p)$ using any number $p \leqslant n/(\log n \log^* n)$ of processors. A variant of the algorithm will yield our second, non-optimal result.

The basic algorithm (of the previous section) had two stages. In the first stage (the **while** loop) we employed an optimal algorithm (given a list of length m it performed $O(m)$ operations); had we performed the **while** loop $O(\log m)$ times to finish shortcutting the list, the algorithm would have taken $O(\log^2 m)$ time. In the second stage (step 5) we used an algorithm that performed relatively more operations (for a list of length m, $O(m \log m)$ operations), but it had the advantage of being faster ($O(\log m)$ time). To profit from this we needed to ensure that the numbers of operations performed by the two stages were roughly the same. And, in fact, this was the case, because the list processed in the second stage was sufficiently shorter. Our present algorithm pushes this methodology further. The algorithm has three main stages, each one processing a relatively shorter list. Stage 1 uses a slow optimal algorithm; its effect is to slightly reduce the length of the input list. Stage 2 uses an almost optimal algorithm; it is faster. Its effect is to further reduce the length of the list. Stage 3 uses the standard deterministic parallel algorithm that misses optimality by a logarithmic factor, but it is the fastest of the three algorithms. The overall result is a fast optimal algorithm. We remark that stage 2, itself, can be considered as a succession of (about $\log^* n$) algorithms, each succeeding algorithm being slightly faster and slightly further from optimal. This methodology was also used in (Vishkin, 1983b). In (Cole and Vishkin, 1986) we call it the accelerating cascades technique.

The input for Stage 1 is the input linked list of length n. The output of stage 1 (and input for Stage 2) is a linked list of length $\leqslant n/(\log^* n)^2$. The output of Stage 2 (input for Stage 3) is a linked list of length $\leqslant n/(\log n)^2$, Each of the linked lists mentioned above is given in an array whose size is the same as the length of the list. Stage 3 simply consists of applying the standard deterministic parallel algorithm.

Remarks. The algorithm will be described in less detail than the preceding algorithms. In particular:

1. At each timestep of stages 1 and 2 we have a linked list that was obtained from the input list by propagating pointers over vertices that were omitted (as in the previous section). In particular, every edge, in any of the linked lists that are obtained throughout these stages, corresponds to a directed path in the original input list. We must maintain a vector (like R in the previous section) that holds, for each such edge, the length of its original path. However, in this presentation we focus only on the transi-

tions from a given linked list to a shorter one and avoid mentioning updates of this vector.

2. Note that in (stages 1 and 2) we only mentioned contractions of a linked list into a shorter one (the up-sweep part using the term of Section 3). We will systematically omit the corresponding down-sweep part throughout this section. No new ideas (beyond Sect. 4) are required in order to fill in this part.

Let k be the integer such that $\log^{(k+1)}n < \log^* n \leqslant \log^{(k)}n$. Note that $k \leqslant \log^* n$. The algorithm proceeds as follows.

Stage 1. This stage applies the **while** loop of the basic algorithm (of Sect. 4) $2(\log^{(k+1)}n)$ times. Thus the output of this stage is a linked list of length $\leqslant n/(\log^{(k)}n)^2$. Clearly, this stage performs $O(n)$ operations in time $O(\log n \log^* n)$.

Stage 2. Stage 2 consists of $k-1$ iterations of Procedure 1.

Iteration i **of Procedure 1** $(1 \leqslant i \leqslant k-1)$.

Let $j = k - i$.

Input. A linked list of length at most $n/(\log^{(j+1)}n)^2$, given in an array having the same length as the list.

Output. A linked list of length at most $n/(\log^{(j)}n)^2$, given in an array having the same length as the list.

1. Apply $2\log^{(j+1)}n - 2\log^{(j+2)}n$ iterations of Routine 1.

   Iteration g **of Routine 1,** $0 \leqslant g < 2\log^{(j+1)}n - 2\log^{(j+2)}n$.

   Input. A linked list of length $m \leqslant 2^{-g}n/(\log^{(j+1)}n)^2$, given in an array of length $\leqslant n/(\log^{(j+1)}n)^2$. (The vertices of the linked list are "spread over" the array which may have more entries than the length of the list. Redundant entries of the array (i.e., entries that represent vertices which are not in the input list for iteration g) are marked as such. The reason for this "wasteful" representation of the input is that iterations of Routine 1 "save time" by not contracting their input array to include only their output list. Only the end of Procedure 1 contracts the linked list at hand).

   Output. A linked of length $m_1 \leqslant m/2$, given in an array of length $\leqslant n/(\log^{(j+1)}n)^2$.

   (a) Apply the recursive version of the basic step (see Sect. 2.2) to obtain a 2-ruling set. (Denote the cardinality of this ruling set by m_1.)

   Explanation. The output list of the present iteration of Routine 1 will consist of the vertices of the ruling set. So for each vertex v in the ruling set the remaining job is to traverse the sublist of v (a list of length $O(1)$); as above, we call this the shortcutting operation.

(b) Shortcut (step 3 of the **while** loop of the basic algorithm).
This completes iteration g of Procedure 1.

Step 2 below concludes the present iteration of Procedure 1.

2. A prefix sum computation is applied in order to contract the input array into an array containing only the vertices of the linked list at hand.

Time complexity of Stage 2. Complexity of iteration g of Routine 1: By Corollary 2.1, using $n/[\log^{(j+1)}n]^2$ processors, step (a) takes time $O(\log^* n)$ time and step (b) takes $O(1)$ time. This yields a bound of $O(n \log^* n/(\log^{(j+1)}n)^2)$ operations and $O(\log^* n)$ time.

Complexity of iteration i of Procedure 1: Step 1 consists of $O(\log^{(j+1)}n)$ invocations of Routine 1. Step 2 needs $O(n/(\log^{(j+1)}n)^2)$ operations and $O(\log n)$ time. Thus the ith iteration of Procedure 1 performs $O(n \log^* n/\log^{(j+1)}n)$ operations in time $O(\log^* n \cdot \log^{(j+1)}n + \log n) = O(\log n)$.

So, overall, Stage 2 performs $O(\sum_{j=1}^{k-1} n \log^* n/\log^{(j+1)}n) = O(n)$ operations in time $O(k \log n)$.

Stage 3 requires $O(\log n)$ time and $O(n/\log^2 n)$ operations. It is also easy to bound the time and number of operations required by the down-sweep part (which is missing in the above description) by the same time and number of operations as for stages 1 and 2.

Putting everything together, remembering that $k \leqslant \log^* n$, and applying Brent's theorem, we deduce

THEOREM 5.1. *The list ranking problem can be solved in time $O(n/p)$ using $p \leqslant n/(\log n \log^* n)$ processors. The implementation problems as per the remark following Brent's theorem can be readily overcome.*

We turn to our other main result.

THEOREM 5.2. *The list ranking problem can be solved in time $O(k \log n)$ using $n \log^{(k)} n/\log n$ processors, for any fixed k.*

Proof. By way of motivation, we observe that, in the algorithm just described, stage 2 is faster than stage 1 (on equal length inputs), but requires more operations. Therefore, by substituting stage 2 for stage 1, we might expect to reduce the running time and increase the total number of operations. So, in the above algorithm, we replace Stage 1 with Routine 1 applied $2 \log^{(k+1)} n$ times, where the input for the gth iteration is a linked list of length $\leqslant 2^{-g}n$, stored in an array of length n. Then we perform the rest of the above algorithm with no change. We achieve a running time of $O(k \log n)$ taking $O(n \log^{(k+1)} n \log^* n) \leqslant O(n \log^{(k)} n)$ operations. Our result follows by Brent's theorem. ▌

This theorem shows that Wyllie's conjecture which was mentioned in the introduction is not correct.

6. Open Problems

(1) Is there an optimal speed-up algorithm for the list ranking problem using $n/\log n$ processors and running in time $O(\log n)$?

(2) We recall that the new coin tossing technique distinguishes the PRAM model from the more abstract PRAM–INFINITY model. We are not aware of any other technique having this property. Are there others? In addition, this remark calls for a "metatheoretical" discussion of the applicability of PRAM–INFINITY lower bounds to PRAMs. We note that a lower bound in the PRAM–INFINITY model is stronger than the same lower bound in the decision tree model, a model that is often used when proving lower bounds. Also, non-trivial lower bounds have been proved for the PRAM–INFINITY model. Thus it seems useful to ascertain the applicability and limitations of such lower bounds.

Acknowledgments

Much gratitude to Baruch Schieber for helpful discussions and comments and to Dennis Shasha for useful comments on an earlier draft.

Received October 28, 1985; accepted December 24, 1985

References

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass.

Chin, F. Y., Lam, J. and Chen, I. (1981), Optimal parallel algorithms for the connected component problems, *in "Proceedings, 1981 International Conference on Parallel Processing,"* pp. 170–175.

Cole, R. and Vishkin, U. (1986), Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms, *in "Proceedings, 18th Annual ACM Symposium on Theory of Computing,"* pp. 206–219.

Fich, F. E., Meyer auf der Heide, F., Ragde, P., and Wigderson, A. (1985), One, two, three... infinity: lower bound for parallel computation *in "Proceedings, 17th Annual* ACM *Symposium on Theory of Computing,"* pp. 48–58.

Fisher, M. and Ladner, L. (1980), Parallel prefix computation, *J. Assoc. Comput. Mach.* **27** (4) 831–838.

Kruskal, C. P., Rudolph, L. and Snir, M. (1985), Efficient parallel algorithms for graph problems, *in "Proceedings, 1985 International Conference on Parallel Processing,"* pp. 180–185.

MEYER AUF DER HEIDE, F., AND WIGDERSON, A. (1985), The complexity of parallel sorting *in "Proceedings, 26th IEEE Annual Conference on Foundations of Computer Science,* pp. 532–540.

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

VISHKIN, U. (1983), "Synchronous Parallel Computation—a Survey," TR 71, Dept. of Computer science, Courant Institute, New York University, New York.

VISHKIN, U. (1983b), An optimal parallel algorithm for selection, manuscript.

VISHKIN, U. (1984), An optimal parallel connectivity algorithm, *Discrete Appl. Math.* **9**, 197–207.

VISHKIN, U. (1984), Randomized speed-ups in parallel computation, *in "Proceedings, 16th Annual ACM Symposium on Theory of Computing,"* 230–239.

VISHKIN, U. (1985), On efficient parallel strong orientation, *Inform. Process. Lett.* **20**, 235–240.

WYLLIE, J. C. (1979), "The Complexity of Parallel Computation," TR79-387, Department of Computer Science, Cornell University, Ithaca, N.Y.