INFORMATION AND COMPUTATION 113, 199–219 (1994)

Virtual Shared Memory: Algorithms and Complexity

ANDREW CHIN* AND W. F. MCCOLL†

Programming Research Group, Oxford University, 11 Keble Road, Oxford OX1 3QD, England

We consider the Block PRAM model of Aggarwal *et al.* (*in* "Proceedings, First Annual ACM Symposium on Parallel Algorithms and Architectures, 1989," pp. 11–21). For a Block PRAM model with $n/\log n$ processors and communication latency $l = O(\log n)$, we show that prefix sums can be performed in time $O(l \log n/\log l)$, but list ranking requires time $\Omega(l \log n)$; these bounds are tight. These results justify an intuitive observation of Gazit *et al.* (*in* "Proceedings, 1987 Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation," pp. 139–156) that algorithm designers should, when possible, replace the list ranking procedure with the prefix sums procedure. We demonstrate the value of this technique in choosing between two optimal PRAM algorithms for finding the connected components of dense graphs. We also give theoretical improvements for integer sorting and many other algorithms based on prefix sums, and suggest a relationship between the issue of graph density for the connected components problem and alternative approaches to integer sorting.

1. INTRODUCTION

A central problem of contemporary computing science concerns the extent to which idealized shared memory models of parallel computation, such as the PRAM, can be efficiently implemented on realistic parallel machines, i.e., those with physically distributed memory. In recent years a number of techniques have been developed which show how such *virtual shared memory* architectures can be supported (Valiant, 1990a, 1990b). In this paper we investigate the complexity of a number of fundamental parallel algorithms with respect to a specific virtual shared memory model.

In distributed memory architectures, communication (or non-local memory access) is performed by a sparse interconnection network and hence is subject to considerable delay, or *latency*. "It is becoming abundantly clear," write Aggarwal *et al.* (1990), "that much of the complexity in parallel computing is due to the difficulty in communication

* Supported by a National Science Foundation Graduate Fellowship and a Rhodes Scholarship.

† Supported by the Science and Engineering Research Council under Grant GR/E01010.

0890-5401/94 $6.00

itself." Although optical networks have been studied extensively as a long-term prospect, an access to a non-local memory location will continue to take longer—and be more difficult to control—than a local computation.

The Block PRAM model of Aggarwal *et al.* (1989) introduces the issue of communication latency as a means of focusing attention on the spatial and temporal locality of references to data in parallel algorithms.

DEFINITION. A *Block PRAM* with p processors and latency l is taken to be an exclusive-read, exclusive-write (EREW) PRAM. The latency is expressed as a multiple of instruction cycles and may be taken to be a function of the number of processors: e.g., $l = \Theta(\log_2 p)$ for a hypercube architecture, and $l = \Theta(p^{1/2})$ for a two-dimensional mesh. Each processor is provided with a local memory of unlimited size. There is also a global memory of unlimited size. A processor may access a location of the local memory in unit time. It may also access a block of b consecutive locations in the global memory in time $l + b$. Since no read or write conflicts are allowed, concurrent requests for overlapping blocks are serviced in some arbitrary order. The input initially resides in global memory, and the output must also be stored there.

A Block PRAM algorithm may take up to $l + 1$ times as long to run as its EREW PRAM counterpart. Aggarwal *et al.* (1989) take steps toward a taxonomy for Block PRAM complexity by noting that the gap of $l + 1$ can be reduced for several problems including matrix transposition, matrix multiplication, and the Fast Fourier Transform. When the gap of $l + 1$ cannot be reduced, as in the case of performing general permutations on elements in memory, the computation cannot efficiently use large blocks for communication and we say that the problem has *fine granularity* (Kruskal and Smith, 1988). In this paper, we present both positive and negative results in Block PRAM complexity. We show that, subject to standard assumptions, the Block PRAM time complexity of list ranking is $\Omega(\min(ln/p, (n \log p)/(p \log(n/lp))))$, while that of prefix sums is $\Theta(n/p + l \log n/\log l)$. When $p = n/\log n$ and $l = O(\log n)$, the Block PRAM lower bound on list ranking is $\Omega(l \log n)$. Since list ranking can be performed in time $O(n/p + \log n)$ in the EREW PRAM model (Anderson and Miller, 1988, Cole and Vishkin, 1988b), we may describe list ranking as a second finely granular problem for the Block PRAM model.

The paper is organized as follows. Section 2 discusses the issue of block pipelining and presents a general Block PRAM complexity lower bound based on fan-in arguments. Section 3 describes our results for the two fundamental problems of prefix sums and list ranking. Section 3.1 presents an optimal Block PRAM algorithm for prefix sums. Section 3.2 proves a tight lower bound on the Block PRAM complexity of list ranking. Section 3.3 discusses some implications of these results. Section 4 extends

these results by illustrating how communication latency can influence our choice between two algorithms for the same problem. Section 4.1 gives a Block PRAM algorithm for integer sorting which minimizes the effect of communication latency, but increases the number of processors. Section 4.2 proves a tight lower bound on the Block PRAM complexity of tree contraction. Section 4.3 implements two well-known optimal PRAM algorithms for connected components (Chin *et al.* 1982; Shiloach and Vishkin, 1982) and presents a clear choice between one Block PRAM algorithm based on prefix sums and another based on list ranking. Section 4.4 relates the issue of graph density in the connected components problem to alternative approaches to integer sorting. Section 5 concludes with remarks and open questions.

Although this paper is self-contained, the reader may find it helpful to refer to Aggarwal *et al.* (1989) for a detailed introduction to the Block PRAM model.

2. Block Pipelining and Latency Hiding

In a multiprocessor network, an access by a processor to a single non-local memory location can, in general, take $\Theta(l)$ time steps to complete. Fortunately, this time does not have to be wasted. Instead of idly waiting for the access to complete, the processor can initiate l more non-local memory accesses, one after the other, in time $\Theta(l)$. By the time the last of these accesses is initiated, we may expect the first access to have been completed. In this way a processor can perform l non-local memory accesses not in time $\Theta(l^2)$, but optimally, in time $\Theta(l)$. The effect is that, within constant factors, a steady flow of memory accesses initiated at the processor results in an equally steady flow of memory accesses being performed remotely in the network, as if there were a pipeline between the processor and the non-local memory. The overhead due to communication latency is "tolerated" or "hidden." This technique is therefore called *pipelining* of memory accesses, and the effect of this technique is called *latency hiding*.

Currently, it is debatable which sequences of memory accesses by a processor can be pipelined. There is no theoretical evidence to suggest that any restrictions are necessary. Accordingly, the Asynchronous PRAM model (Gibbons, 1989) and the XPRAM model (Valiant, 1990a) both allow pipelining of arbitrary sequences of accesses to the shared memory. However, there is as yet no practical evidence that *arbitrary pipelining* can be supported (i.e., with small constant factors).

A more conservative alternative is offered by *block pipelining*. In block pipelining, only contiguous blocks of memory are pipelined. For several

years, block pipelining has been achieved and exploited in real multiprocessor networks. As Aggarwal *et al.* (1989) observe: "Typically, it takes a substantial period of time to get the first word from global memory, but after that, subsequent words can be obtained quite rapidly—essentially at the clock speed of the machine.... The size of a block that is transferred is typically correlated with communication latency." Despite the restriction of block pipelining, latency hiding can be applied to arbitrary PRAM algorithms, as the following result shows.

THEOREM 2.1 (Aggarwal, Chandra, and Snir, 1989). *Let ε and k be positive constants. Then T steps of a PRAM computation with q processors and q^k memory can be simulated (probabilistically) in $O(Tq/p)$ steps on a Block PRAM with p processors, provided that $q \geqslant lp^{1+\varepsilon}$.*

With high probability, the $O(q/p)$ memory accesses required by each Block PRAM processor during each simulation step can be grouped into $O(q/pl)$ contiguous blocks of length l, making the simulation efficient.

In practical terms, a successful Block PRAM algorithm will enable us to increase the degree of parallelism when performed on a general purpose parallel computer. On the other hand, a tight lower bound on Block PRAM complexity for a given problem is evidence that general techniques for latency hiding are essentially the best possible.

A simple lower bound applies to a wide variety of computations. A function $f(x_1, ..., x_n)$ is *sensitive on all its variables* if there is a data instance $x_1 = a_1, ..., x_n = a_n$ such that for each i, $1 \leqslant i \leqslant n$, there is a b_i with $f(a_1, ..., a_i, ..., a_n) \neq f(a_1, ..., b_i, ..., a_n)$.

THEOREM 2.2 (Aggarwal *et al.*, 1989). *Let $f(x_1, ..., x_n)$ be sensitive on all its variables. Then any Block PRAM algorithm computing f requires time $\Omega(n/p + l \log n/\log l)$.*

3. PREFIX SUMS VERSUS LIST RANKING

Prefix sums and list ranking are two basic problems in the theory and design of PRAM algorithms (Karp and Ramachandran, 1990). They may be considered *primitive procedures* in the sense that parallel algorithms for many other fundamental problems call for their use as subroutines, but their own algorithms seem to be structurally independent. However, they do share several features. They both may be considered to be operations on arrays; they have linear-time sequential algorithms; and they have optimal EREW PRAM algorithms with running time $O(\log n)$. It is instructive to contrast them in the Block PRAM model.

3.1. *Prefix Sums*

Let $\oplus$ be an associative binary operation with identity 0 that may be computed sequentially in $O(1)$ time. The *prefix sums* $\oplus$ computation takes an array $A = (a_0, a_1, a_2, ..., a_{n-1})$ and returns the array $(a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, ..., a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1})$. If $\oplus$ is addition, this computation is called *prefix addition*. If $\oplus$ is the **copy** operation defined by x **copy** $y = x$, the prefix sums computation can be used to *broadcast*, or make multiple copies of data.

This problem has a trivial linear-time sequential algorithm and a well known $O(n/p + \log n)$ time EREW PRAM algorithm (Ladner and Fischer, 1980). A prefix sums algorithm for the CRCW PRAM model with $O(\log n)$ word size runs in time $O(n/p + \log n/\log\log n)$ (Cole and Vishkin, 1986b, Reif, 1985). The prefix sums computation is sensitive on all its variables since the sum of all the elements in the array A must be computed, so that the lower bound of Theorem 2.2 applies. We now give an optimal Block PRAM algorithm.

ALGORITHM PREFIX $(A, n, \oplus, S)$.

Input: Array A of length n.

Output: Array S, the prefix sums $\oplus$-computation of A.

Comments: The levels of the l-ary tree are represented as the rows of the matrix B. The levels of the tree are numbered $[1 \cdots \lceil \log p/\log l \rceil]$ from the leaves to the root. The matrix is taken to be stored in row-major order as a contiguous array of $O(n)$ entries. The array can be initialized to 0 in time $O(n/p + l)$. Each processor uses two local variables, *count* and *oldcount*; all other variables are global. The output is returned as array S.

```
1   for all i, 0 ≤ i < p do in parallel
        count ← 0
        for k ← 0 until ⌈log p/log l⌉ do
            B(0, i·⌈n/p⌉ + k) ← count
            count ← count ⊕ A(i·⌈n/p⌉ + k)
        B(1, i) ← count
2   for r ← 1 until ⌈log p/log l⌉ do
        for all i, 0 ≤ i < p/l^r do in parallel
            oldcount ← 0
            count ← 0
            for k ← 0 until l − 1 do
                count ← count ⊕ B(r, il + k)
                B(r, il + k) ← oldcount
                oldcount ← count
            B(r + 1, i) ← count
```

3 **for** $r \leftarrow \lceil \log p/\log l \rceil$ **down to** 1 **do**
  for all i, $0 \leqslant i < p/l^r$ **do in parallel**
    for $k \leftarrow 1$ **until** $l-1$ **do**
      $B(r, il+k) \leftarrow B(r+1, i) \oplus B(r, il+k)$
4 **for** all i, $0 \leqslant i < p$ **do in parallel**
  for $k \leftarrow 0$ **until** $\lceil n/p \rceil - 1$ **do**
    $S(i \cdot \lceil n/p \rceil + k - 1) \leftarrow B(1, i) \oplus B(0, i \cdot \lceil n/p \rceil + k)$
5 $S(n-1) \leftarrow S(n-2) + A(n-1)$

Analysis: Steps 1 and 4 each take $O(n/p + l)$ time. Each of the $\lceil \log p/\log l \rceil$ iterations of steps 2 and 3 takes $O(l)$ time. Step 5 takes $O(l)$ time. The overall time is $O(n/p + l \log p/\log l) = O(n/p + l \log n/\log l)$.

THEOREM 3.1. *The complexity of performing prefix sums computations on an array of length n on the Block PRAM is* $\Theta(n/p + l \log n/\log l)$.

In his book on vector models of parallel computation, Blelloch (1990) introduces the *scan vector model* of computation, in which prefix sums are assumed to be primitive operations and implemented in unit time. He argues the practicality of the scan vector model by noting the low circuit complexity (Ladner and Fischer, 1980) and the relatively low empirical running time (Hillis, 1985) of prefix sums computations. Blelloch catalogs a number of problems in graph theory, static networks, and computational geometry for which there are parallel algorithms in the scan vector model which run asymptotically faster than the corresponding benchmark EREW PRAM algorithms. In particular, scan vector model algorithms for connected components, biconnected components, minimum spanning tree, maximum flow, maximum independent set, logic simulation, neural networks, convex hull, building a k-d tree, closest pair in the plane, and line of sight all run $O(\log n)$ times faster than on the EREW PRAM model. Since, by the previous theorem, the cost of adapting a scan vector model algorithm to the Block PRAM model using the procedure PREFIX is $O(l \log n/\log l)$, the scan vector model algorithms run at most $O(l/\log l)$ times slower on a Block PRAM than on an EREW PRAM. This leads us to suggest the Block PRAM model as a basis in PRAM theory for the specific study of prefix sums-based parallel algorithms, beyond the empirical justifications given in Blelloch (1990).

3.2. *List Ranking*

The *linked list* is an alternative to the array in storing sequences of elements in memory. Instead of forming a sequence of consecutive memory locations, the elements in a linked list can appear in any order in shared memory. Stored with each element is a *pointer* giving the address of the

next element of the sequence. Arrays are preferable to linked lists when it is necessary to find the kth element of a sequence. However, linked lists are preferable to arrays when it is necessary to perform many insertion and deletion operations.

DEFINITIONS. A *linked list* of n elements consists of two arrays $A[1 \cdots n]$ and $S[1 \cdots n]$. A is called the *data array* and S is called the *pointer array*. The first element in the list, called the *head*, is stored in $A(1)$. For $1 \leqslant i \leqslant n-1$, if the ith element of the list is stored in $A(j)$, then the $(i+1)$th element of the list is stored in $A(S(j))$. That is, the $(i+1)$th element of the list is stored in $A(S^{(i)}(1))$. The values of $S^{(i)}(1)$, $1 \leqslant i \leqslant n-1$, are distinct integers in the range $\{2, 3, ..., n\}$. By convention, $S^{(n)}(1) = 0$. Given a linked list (A, S), the *list ranking* problem is to compute the equivalent array B. That is, compute $B[1 \cdots n]$ such that $B(i) = A(S^{(i-1)}(1))$. Equivalently, compute elements in an array of *distances* $D[1 \cdots n]$ such that $A(i) = A(S^{(n-D(i)-1)}(1))$ for each element $A(i)$ of the list.

The list ranking problem has a trivial linear time serial algorithm and a standard $O(n \log n/p + \log n)$ time EREW PRAM algorithm (Wyllie, 1979). Optimal $O(n/p + \log n)$ time EREW PRAM algorithms are also known (Anderson and Miller, 1988; Cole and Vishkin, 1988). The latter algorithms are "rather elaborate" (Karp and Ramachandran, 1990), and Anderson and Miller (1988) concede that the standard algorithm "is still probably the best deterministic algorithm" in practice. Trading determinism for simplcity and lower constants, randomized EREW PRAM algorithms taking time $O(n/p + \log n)$ have recently appeared (Anderson and Miller, 1990; Karp and Ramachandran, 1990).

The basic step in all known efficient list ranking algorithms is *shortcutting* individual elements from the linked list. To shortcut an element $A(i)$, $A(i)$ is removed from the list and the pointer $S(S^{-1}(i))$ of its predecessor $A(S^{-1}(i))$ is updated, or "jumped," to its successor $S(i)$. (In this section, we frequently view the array S as an invertible function: $S^{-1}(i)$ denotes the value x, $1 \leqslant x \leqslant N$, such that $S(x) = i$.) The distance computation follows each pointer jumping step: if $S(i) \neq 0$ then $D(i)$ is updated to $D(i) + D(S(i))$. Elements can be shortcut in parallel, provided that no two adjacent elements are shortcut simultaneously. Finding a large set of nonadjacent elements to shortcut—*symmetry breaking*—is a nontrivial problem, and the improved deterministic algorithms in the literature have used the parallel symmetry-breaking technique of Cole and Vishkin (1986a), called "deterministic coin tossing."

The sequential list ranking algorithm can be viewed as shortcutting the elements one at a time from the head of the list. It is observed in Anderson and Miller (1988) that the standard parallel algorithm is inefficient because

it shortcuts the same element from a number of different lists instead of leaving it alone once it has been removed: this technique is called *recursive doubling* (see, e.g., Gibbons and Rytter, 1988; Leiserson and Maggs, 1988). The optimal parallel algorithms address this inefficiency by shortcutting each element exactly once.

DEFINITION. An algorithm for list ranking is *shortcut-based* if in its execution on a list (A, S):

- For each element $A(i)$, some processor executes a *shortcut* step:
  $S(S^{-1}(i)) \leftarrow S(i)$
  Remove $A(i)$

(Note that S, the array of pointers, changes during the algorithm).

- No two consecutive elements $A(i)$, $A(S(i))$ are shortcut in parallel.

In this section we establish an $\Omega(\min(ln/p, (n \log p)/(p \log(2n/lp))))$ lower bound, for $lp \leqslant n$, on the running time of any shortcut-based Block PRAM algorithm for list ranking. The proof uses the following result of Aggarwal *et al.* (1989) that there exist permutations which are hard for Block PRAMs to perform *conservatively* on arrays in global memory. (An algorithm is *conservative* if the only operation allowed is copying of elements in memory.)

THEOREM 3.2 (Aggarwal *et al.* 1989). *There is a permutation Π on n elements such that any conservative Block PRAM algorithm for performing Π on n consecutive locations in shared memory requires time $\Omega(\min(nl/p, n \log p/(p \log(2n/pl))))$ for $pl \leqslant n$.*

To establish our result we require the following slightly stronger version of the theorem.

THEOREM 3.3. *Let S_n be the set of all permutations on $\{0, 1, ..., n-1\}$. Then there is a subset $E \subseteq S_n$ with $\log |E| = o(n \log n)$ such that for all $\Pi \in S_n \setminus E$, any conservative Block PRAM algorithm for performing Π on n consecutive locations in shared memory requires time $\Omega(\min(nl/p, n \log p/(p \log(2n/pl))))$ for $pl \leqslant n$.*

The proof of this stronger version is easily obtained from the original one given by Aggarwal *et al.* (1989), if we note that the number of "easy" permutations can be bounded. This is because the proof is a counting argument bounding the number of different permutations that can be attained by a Block PRAM within a given time bound.

We can now informally describe our problem reduction. Assume there is a fast Block PRAM algorithm for list ranking. Consider a "hard" permuta-

tion. We apply the algorithm to a linked list whose pointers are the actions of the permutation. By observing how the shortcutting takes place, we derive an impossibly fast conservative algorithm for performing the permutation in shared memory.

THEOREM 3.4. *Any shortcut-based Block PRAM algorithm for list ranking requires time* $\Omega(\min(nl/p, n \log p/(p \log(2n/pl))))$ *for* $pl \leqslant n$.

Proof. Let $\mathscr{W}$ be a Block PRAM algorithm which performs list ranking by shortcutting within time $T = T(n, l, p)$. Consider an n-cycle Π on $\{1, ..., n\}$ such that $\Pi \in S_n \setminus E$, where E is the set of "easy" permutations in Theorem 3.2. (There are $(n-1)! = 2^{\Omega(n \log n)}$ n-cycles, so that almost all of them are "hard.") Given an array M, we give a conservative algorithm $\mathscr{V}$ which performs Π on the elements of M in time bounded by $3T$.

Let $L = (A, S)$ be a compact linked list whose pointers are given by the actions of Π; i.e., $S(i) = \Pi(i)$ for $1 \leqslant i \leqslant n$ unless $\Pi(i) = 1$. By convention, we define $S(\Pi^{-1}(i)) = 0$. Execute $\mathscr{W}$ on the problem instance L. Each of the list elements $A(2), A(3), ..., A(n)$ must be shortcut from L. To shortcut $A(i)$, some processor must have exclusive access to each of $S(S^{-1}(i))$ and $S(i)$.

Suppose that processor P shortcuts $A(i)$ beginning at time step t. As part of the algorithm $\mathscr{V}$, we schedule the following exchange procedure for processor P beginning at time step $3t$, where the actual values of i and $A^{-1}(i)$ are taken from tracing the execution of $\mathscr{W}$ (*Temp* is a local variable; all other variables are global):

$$Temp \leftarrow M(A^{-1}(i))$$

$$M(A^{-1}(i)) \leftarrow M(i)$$

$$M(i) \leftarrow Temp$$

These procedures compose the entire algorithm $\mathscr{V}$. We check that $\mathscr{V}$ is well defined. Clearly each exchange procedure requires at most three times as long as the corresponding shortcutting procedure, regardless of how global memory accesses are pipelined, so that the exchanges can be performed at the scheduled times and in the correct sequence. The action of $\mathscr{V}$ on M may be written as a product of 2-cycles. Exchanges which are performed in parallel correspond to shortcutting nonadjacent elements of L and, therefore, to disjoint 2-cycles, which commute. Hence no conflicts in access to global memory are created, and the action of $\mathscr{V}$ is well defined. It should be noted also that $\mathscr{V}$ is obviously a conservative algorithm.

Finally, we establish that $\mathscr{V}$ performs the permutation Π on the elements of M. Let M_0, A_0 be the original input arrays to algorithms $\mathscr{V}$ and $\mathscr{W}$, respectively. Then it is easy to check that the invariant "for each i such that

$A(i)$ is in the list, $M(i) = M_0(A_0^{-1}(A(i)))$" remains true throughout the reduction. Now, when any element $A(i)$ is shortcut in $\mathcal{W}$, the corresponding memory exchange in $\mathcal{V}$ places the value of $M(i)$ into the $A(i)$th location of M; this value is never moved again. But by the invariant, this is the same value that the permutation Π would have placed into the location. The algorithm $\mathcal{V}$ terminates within time $3T$, as required. ▮

Using this reduction, one can observe a 1–1 correspondence between factorizations of Π into 2-cycles and possible sequences of shortcutting steps in reducing L. We also observe that list ranking is finely granular for $l = O(\log p)$, $p = n/\log n$.

COROLLARY 3.5. *Assuming shortcut-based algorithms, the complexity of the problem of ranking a list of length n on a Block PRAM with* $p = n/\log n$ *processors and latency* $l = O(\log p)$ *is* $\Theta(l \log n)$.

Proof. The upper bound follows immediately from any of the optimal $O(n/p + \log n)$ EREW PRAM algorithms for list ranking (Anderson and Miller, 1988; Cole and Vishkin, 1988), since all of them are shortcut-based. ▮

3.3. *Discussion*

There is no evidence to suggest that list ranking can be performed efficiently without shortcutting. This has already cast some doubt on the prospects for optimal list ranking in practice.

- In a discussion paper titled "Are pointer-based parallel algorithms realistic?" Miller (1989) recognizes a growing gap between the body of efficient PRAM algorithms and the real machines being designed to implement them. On the Connection Machine, a single shortcutting step takes about 1000 times as long as a single local instruction step. Miller concludes: "I hope that designers of future general-purpose parallel machines will consider the list ranking problem when they design their machines."

- Leiserson and Maggs (1988) point out that recursive doubling can lead to congestion because all of the active pointers propagate toward the end of the list. This quickly results in too many processors attempting to access pairs of earlier and later pointers. However, they also show that this problem can be eliminated by symmetry breaking (e.g., deterministic coin tossing) and shortcutting each pointer only once.

- Gazit *et al.* (1987) offer a critical comparison between prefix sums and list ranking. They observe that on a hypercube machine with $p = n/\log n$ processors, prefix sums can be computed in time $6 \log n$, but the list ranking algorithms in Anderson and Miller (1988, 1990) and Cole and Vishkin (1988) seem to have running time $O(\log^2 n)$. They conclude that "if

the ultimate purpose of a parallel algorithm is to run it on a fixed connection machine, then we should minimize the number of list rankings we perform; and, whenever possible, replace the list ranking procedure with the prefix sums procedure." However, they do not justify this conclusion by proving an $\omega(n/p + \log n)$ lower bound for list ranking on the hypercube.

Taken together, our results for prefix computations and list ranking give the first rigorous theoretical justification for these emerging practical concerns about the difficulty of pointer jumping. As a shortcut-based procedure, list ranking represents an asymptotic bottleneck in many Block PRAM computations. One way to overcome the list ranking bottleneck is to exploit the similarity between list ranking and prefix computations (observed, e.g., in Karp and Ramachandran, 1990). It is possible although not immediately obvious that we can replace list ranking with prefix sums to obtain asymptotic improvements in complexity.

- Prefix and list ranking computations have been used interchangeably for very simple procedures such as finding the minimum of n values (Gibbons and Rytter, 1988). When such procedures are implemented on the Block PRAM, prefix computations should be used.
- Gazit *et al.* (1987) give an algorithm for tree contraction (see Section 3.2) which replaces a constant proportion of the list ranking operations with prefix sums. This has no overall effect on the algorithm's asymptotic complexity in any model.
- More substantively, we demonstrate in Section 3.3 that we can choose prefix sums over list ranking as the basic step for a Block PRAM algorithm to find the connected components of dense graphs, with an asymptotic savings in complexity.

4. Applications to Parallel Algorithms

We turn now to the design and analysis of algorithms based on prefix sums and list ranking. We show how communication latency may influence our choice between two algorithms for the same problem, and discuss the extent to which algorithms based on the prefix sums procedure should take precedence in the theory of PRAM complexity.

4.1. *Integer Sorting*

Let A be an array of n numbers, or *keys*, in the range $\{1, 2, ..., n\}$. The *integer sorting* problem is to produce an array A' whose elements are the keys arranged in nondecreasing order. The *stable integer sorting problem*

asks us to produce also a permutation π such that $A'(i) = A(\pi(i))$, and $\pi(i) < \pi(j)$ if $A'(i) = A'(j)$ and $i < j$. A deterministic sequential algorithm for stable integer sorting taking $O(n)$ time is well known (Aho *et al.*, 1974), as is a deterministic EREW PRAM algorithm taking time $O(n \log n/p + \log n)$ (Hirschberg, 1978). Specific algorithms for the integer sorting problem work by moving each key into a group of cells (*bucket*) according to its place values (*radices*) when expressed in some base system. Hence integer sorting is sometimes referred to as *bucket sorting* or *radix sorting*. In the general sorting problem, the keys are not assumed to have any numerical value, and must be ordered according to comparisons among the keys. This makes the problem more difficult: the sequential complexity of general sorting is $\Theta(n \log n)$.

Aggarwal *et al.* (1989) describe a method for simulating the Ajtai *et al.* (1983) sorting network ("AKS network") on the Block PRAM model in $O(n \log n/p + l \log n)$ time, so that general keys can be sorted optimally when $lp \leqslant n$.

THEOREM 4.1 (Aggarwal *et al.*, 1989). *On the Block PRAM, n keys can be sorted in time* $O(n \log n/p + l \log n)$.

Although this is a strong result, it does not settle the problem of sorting for practical purposes. Even after improvements by Paterson (1987), the AKS network involves such large constants that results based on its existence still have only theoretical value. Also, there is a gap of $\log l$ from the lower bound of $\Omega(l \log n/\log l)$ given by Theorem 2.2. We now give a stable integer sorting algorithm for which $O(l \log n/\log l)$ time can be attained. The price for this improvement, however, is an asymptotic increase in the number of processors.

ALGORITHM INTEGER SORT (A, A', n, π).

Input: Array A of n integers in the range $\{1, 2, ..., n\}$

Output: Sorted array A'; permutation π such that $A'(i) = A(\pi(i))$, and $\pi(i) < \pi(j)$ if $A'(i) = A'(j)$ and $i < j$.

Comments: The sort is a radix sort where the integers are expressed in base x. We assume that x is an integral power of 2, affecting the running time by at most a constant factor. The bucket for each radix is a row of B, an $x \times n$ matrix. For $1 \leqslant i \leqslant n$, a 1 is placed in the $A(i)$th bucket at the ith column. The number of 1's in each bucket is counted, and ranks are assigned to each 1 in the matrix ocurring in order from left to right, then top to bottom. This is done by combining the prefix of the bucket counts and the prefix sums along each row.

We assume $x \leqslant p \leqslant n^2$; with fewer processors we can appeal to Brent's Theorem (see, for example, Gibbons and Rytter, 1988), which allows us to

trade time for processors. We cannot use more than n^2 processors efficiently. Let $m = \lfloor p/x \rfloor$, the number of processors available for each bucket. Let $R \leftarrow \lceil \log n / \log x \rceil$, the number of rounds in the algorithm.

Steps 4 and 6 are bookkeeping operations. Step 4 puts the full sums which were computed in step 3 into a contiguous array. Step 6 makes m copies of each subtotal in the array G to avoid read conflicts in step 7. In addition to the local variables used in the subroutine PREFIX, each processor uses the local variables *radix* and *rank*; all other variables are global.

```
0   for all i, 0 ≤ i < p do in parallel
        for k ← 0 until ⌈n/p⌉ − 1 do
            π(i · ⌈n/p⌉ + k) ← i · ⌈n/p⌉ + k
            A'(i · ⌈n/p⌉ + k) ← A(i · ⌈n/p⌉ + k)
    for r ← 1 until R repeat steps 1–8
1   for all i, j, 0 ≤ i < x, 0 ≤ j < m do in parallel
        for k ← 0 until ⌈n/m⌉ − 1 do
            B(i, j · ⌈/m⌉ + k) ← 0
2   for all i, 0 ≤ i < x do in parallel
        for k ← 0 until ⌈n/p⌉ − 1 do
            radix ← A'(i · ⌈n/p⌉ + 1) − x^(R−r) · ⌊A'(i · ⌈n/p⌉ + k)/x^r⌋
            radix ← ⌊radix/x^(r−1)⌋
            B(radix, i · ⌈n/p⌉ + k) ← 1
3   for all i, 0 ≤ i < x do in parallel
        PREFIX (B_i, n, +, S_i)
4   for all i, 0 ≤ i < x do in parallel
        T(i) ← S_i(i − 1)
5   PREFIX (T, x, +, G)
6   for all i, 0 ≤ i < x do in parallel
        C(i, 0) ← G(i − 1)
        PREFIX (C_i, m, copy, C'_i)
7   for all i, j, 0 ≤ i < x, 0 ≤ j < m do in parallel
        for k ← 0 until ⌈n/m⌉ − 1 do
            if B(i, j · ⌈n/m⌉ + k) = 1 then
                rank ← S_i(j · ⌈n/m⌉ + k) + C'_i(j)
                ρ(rank) ← j · ⌈n/m⌉ + k
8   for all i, 0 ≤ i < p do in parallel
        for k ← 0 until ⌈n/p⌉ − 1 do
            π(i · ⌈n/p⌉ + k) ← π(ρ(i · ⌈n/p⌉ + k))
            A'(i · ⌈n/p⌉ + k) ← A(π(i · ⌈n/p⌉ + k))
```

Analysis: The initialization step 0 takes $O(n/p + l)$ time. During each round, steps 1 and 7 take time $O(n/m + l)$, steps 2 and 8 take time $O(nl/p)$, step 3 takes time $O(n/m + l \log n / \log l)$, step 4 takes time $O(l)$, step 5 takes time $O(l \log x / \log l)$, and step 6 takes time $O(l \log m / \log l)$. The overall

time complexity is $O((\log n/\log x)\cdot(n(x+l+\log x)/p + l\log n/\log l))$. By setting $x = 2^{\lceil \log(l+(pl\log n)/(n\log l))\rceil}$ we have

THEOREM 4.2. *On the Block PRAM, n integers in the range* $\{1, 2, ..., n\}$ *can be sorted stably in time* $O(nl\log n/(p\log l) + l\log^2 n/\log^2 l)$ *for* $p \leqslant n$, *and in time* $O(l\log^2 n/(\log l \log(pl/n)))$ *for* $n \leqslant p \leqslant n^2$.

COROLLARY 4.3. *On the Block PRAM, n integers in the range* $\{1, 2, ..., n\}$ *can be sorted stably in time* $O(l\log^2 n/\log^2 l)$ *on* $n\log l/\log n$ *processors, and in time* $O(l\log n)/\log l)$ *on* $n^{1+\varepsilon}$ *processors, for any* $\varepsilon > 0$.

4.2. *Tree Contraction*

A (*rooted*) *tree* $T = (V, E, r)$ is a connected directed acyclic graph in which the root r has outdegree 0 and the other vertices have outdegree 1. The leaves of a tree are those vertices having indegree 0; the other vertices are called *internal vertices.* If the leaves of a tree are labeled with constants and the internal vertices are labeled with basic arithmetic instructions, the resulting data structure is an *expression tree.* We define the *value* of an expression tree inductively as follows. The value of a leaf is the value of its label. The value of an internal vertex is the value of its arithmetic instruction applied to its inputs. The value of an expression tree is the value of its root. If the operation at an internal vertex is not commutative, the inputs to that vertex must be ordered in the specification of the expression tree.

An expression tree on n vertices can be represented with three arrays A, S, G on $[1 \cdots n]$, where $A(i)$ gives the description of node i, and i is specified as the $G(i)$th input to vertex $S(i)$. The definition of shortcutting can then be extended in the obvious way from linked lists to expression trees. In the EREW PRAM model, an expression of length n may be evaluated in $O(n/p + \log n)$ time in two stages: first, construction of an expression tree; and second, evaluation, or contraction, of the expresion tree. All of the known efficient tree contraction algorithms are shortcut-based.

THEOREM 4.4 (Bar-On and Vishkin, 1985). *Given an arithmetic expression of length n with operations* $+, -, \times, /$, *and brackets, the corresponding expression tree can be constructed on the EREW PRAM in* $O(n/p + \log n)$ *time.*

THEOREM 4.5 (Gazit *et al.*, 1987). *An expression tree with n vertices can be evaluated on the EREW PRAM in* $O(n/p + \log n)$ *time.*

We can now state our complexity results for tree contraction on the Block PRAM. A straightforward problem reduction gives

PROPOSITION 4.6. *Any shortcut-based Block PRAM algorithm for tree contraction requires time* $\Omega(\min(nl/p, n \log p/(p \log(2n/pl))))$ *for* $pl \leqslant n$.

Proof. From a compact list of length n we can construct an expression tree of size $2n$ as follows: each list has value 1, each internal node computes binary addition, and the list's pointer array is the same as the tree's successor array restricted to the internal nodes. Clearly this reduction can be performed in time $O(n/p + l)$. Contraction of this tree will rank the original list. ∎

COROLLARY 4.7. *The complexity of contracting an expression tree with n vertices on a Block PRAM with* $p = n/\log n$ *processors and latency* $l = O(\log p)$ *is* $\Theta(l \log n)$, *assuming shortcut-based algorithms.*

4.3. *Connected Components*

Let $G = (V, E)$ be a simple undirected graph, where V is a set of n vertices and E is a set of m edges. G is *connected* if there exists a path between every pair of distinct vertices in V. A *connected component* of G is a maximal connected subgraph of G. We define G to be *dense* if $m = \Theta(n^2)$ and *sparse* otherwise. Connected components can be found sequentially by depth-first search in time $O(m + n)$ (i.e., $O(n^2)$ for dense graphs. The best known PRAM algorithms for finding connected components are the following:

- a CRCW PRAM algorithm taking $O((m + n)\,\alpha(m, n)/p + \log n)$ time (Cole and Vishkin, 1986b) ($\alpha(m, n)$ is the inverse Ackermann function, a function that grows so slowly that it is constant for all practical purposes.)
- a CREW PRAM algorithm taking $O(n^2/p + \log^2 n)$ time (Chin *et al.*, 1982; Vishkin, 1984).

Note that the CREW PRAM algorithm is optimal for dense graphs and the CRCW PRAM algorithm is almost optimal for sparse graphs. Whether there exists an optimal $O(\log n)$ time PRAM algorithm for sparse graphs remains an important open problem in parallel complexity theory. We consider two algorithms for finding connected components: the algorithm of Chin *et al.* (1982) mentioned above and the $O((m + n) \log n/p + \log n)$ time CRCW PRAM algorithm of Shiloach and Vishkin (1982).

The main idea of the CREW PRAM algorithm is the *path halving* technique of Hirschberg (1976). Trees in an undirected graph are constructed, or "hooked" together, byhaving each vertex point to its lowest-numbered neighbor. These trees are compressed, and vertices along these paths are merged into "supervertices." The process is then repeated on the graph induced by the supervertices, then on the super-supervertices, and so on.

The number of supervertices is reduced by half after each iteration, so that the algorithm is finished after $\log n$ stages. Let $C[1 \cdots n]$ be an array of integers with $1 \leqslant C(i) \leqslant n$ such that C induces a pointer graph containing no cycles of length $\geqslant 2$; that is, for $k \geqslant 2$, there are no distinct $i_1, ..., i_k$ such that $C(i_1) = i_2, ..., C(i_{k-1}) = i_k$, $C(i_k) = i_1$. The *path halving* problem is to compute C^*, where $C^*(i) = C^{(n-1)}(i)$.

In the algorithms of Chin *et al.* (1982) and Hirschberg (1976), the path halving problem is solved on a CREW PRAM in time $O(n \log n/p + \log n)$: for each of $\log n$ iterations, each processor performs the shortcutting operation $C(i) \leftarrow C(C(i))$. This procedure is essentially the standard parallel algorithm for list ranking where we do not bother to compute ranks. Here, concurrent reads are necessary because $C(i) = C(j)$ can occur for distinct i, j. The path halving problem can also be solved using just the pointer-jumping steps from any shortcut-based tree contraction algorithm.

The CRCW PRAM algorithm of Shiloach and Vishkin (1982) improves on the CREW PRAM algorithm by performing the shortcutting operation only a constant number of times on each processor during each iteration. All of the vertices, not just the supervertices, share the burden of "hooking" trees together. This change creates possible write conflicts in global memory. However, in the Block PRAM model, resolving these conflicts using the prefix sums procedure takes time $O(l \log n/\log l)$, asymptotically faster than performing $\log n$ shortcutting operations in time $O(l \log n)$. The algorithm uses $2m + n$ processors by assigning one processor to each vertex and two processors to each edge, one at each end. The vertex processors are never active at the same time as the edge processors. Among the edge processors, concurrent accesses occur only between processors corresponding to adjacent edges, and then only in accessing data associated with the common vertex. Among the vertex processors, concurrent accesses occur only at pointer jumping steps $C(i) \leftarrow C(C(i))$.

Let $\delta(v)$ denote the number of edges incident to vertex v. To provide for $\delta(v)$ concurrent reads of a datum in global memory associated with v (a *v-datum*), we maintain a block of $\delta(v) + 1$ copies of it. (One copy is kept as the original.) Copies of the initial data can be produced using a prefix **copy** computation. To provide for $\delta(v)$ concurrent writes, each processor writes to its own copy of the v-datum. After each write round, we perform three prefix computations: the first two to find a copy which disagrees with the original (if any), the third to make $\delta(v) + 1$ copies of it. Now suppose we are performing the algorithm on a Block PRAM with p processors, where $n \leqslant p < m$. Then we can schedule the algorithm so that each Block PRAM processor always simulates several CRCW PRAM processors accessing the same v-datum. Then since the copies of each v-datum are stored contiguously, the accesses from each processor can be pipelined.

The simulation procedure performs $O(m)$ operations taking time $O(l \log n/\log l)$, provided $p \geqslant n$. By Brent's Theorem, it follows that each step involving the m edge processors can be simulated in time $O(m/p + nl \log n/(p \log l) + l \log n/\log l)$.

We now show how concurrent reads during a pointer jumping step $C(i) \leftarrow C(C(i))$ can be supported on the Block PRAM. The idea is to make one copy of $C(i)$ for each occurrence of i in the array C. Scheduling the processors to do this is fairly involved, and is described below. For clarity, we describe analgorithm for n processors; with fewer than n processors we can appeal to Brent's Theorem.

PROCEDURE SHORTCUT (C, n).

1 **for all** i, $1 \leqslant i \leqslant n$ **do in parallel**
  $L(i) \leftarrow T(i) \leftarrow U(i) \leftarrow 0$
2 INTEGER SORT(C, C', n, π)
3 $B(1) \leftarrow C'(1)$
  for even i, $2 \leqslant i \leqslant n$ **do in parallel**
  $B(i) \leftarrow C'(i) - C'(i-1)$
  for odd i, $3 \leqslant i \leqslant n$ **do in parallel**
  $B(i) \leftarrow C'(i) - C'(i-1)$
4 **for all** i, $1 \leqslant i \leqslant n$ **do in parallel**
  if $B(i) > 0$ **then do**
  $D(i) \leftarrow C(C'(i))$
  $B(i) \leftarrow 1$
5 PREFIX $(B, n, +, S)$
6 **for all** i, $1 \leqslant i \leqslant n$ **do in parallel**
  if $B(i) = 1$ **then do**
  $L(i) \leftarrow i$
  $T(S(i)) \leftarrow i$
7 **for odd** i, $1 \leqslant i \leqslant n$ **do in parallel**
  if $T(i) > 0$ **and** $T(i+1) \neq 0$ **then** $U(T(i)) \leftarrow T(i+1)$
  if $T(i) > 0$ **and** $T(i+1) = 0$ **then** $U(T(i)) \leftarrow n+1$
  for even i, $2 \leqslant i \leqslant n$ **do in parallel**
  if $T(i) > 0$ **and** $T(i+1) \neq 0$ **then** $U(T(i)) \leftarrow T(i+1)$
  if $T(i) > 0$ **and** $T(i+1) = 0$ **then** $U(T(i)) \leftarrow n+1$
8 **for** $r \leftarrow 0$ **until** $\lceil \log n/\log l \rceil - 1$ **do**
  for all i, $1 \leqslant i \leqslant n$ **do in parallel**
  if $B(i) = 1$ **then do**
  for $k \leftarrow 0$ **until** $l-2$ **do**
  $m \leftarrow L(i) + l^r + (i - L(i)) \cdot (l-1) + k$
  if $m < U(i)$ **then do**
  $D(m) \leftarrow D(i)$

$B(m) \leftarrow 1$
$L(m) \leftarrow L(i)$
$U(m) \leftarrow U(i)$

9 **for** all i, $1 \leqslant i \leqslant n$ **do in parallel**
$C(\pi(i)) \leftarrow D(i)$

Comments: In Step 3, if $B(i) > 0$ then i is a "leader"; the corresponding processor is "active" and will fetch the first copy of $C(C'(i))$.) In step 4, D is the array where the copies will be stored; it need not be initialized. Step 8 is a specialized broadcasting procedure. $B(i)$ specifies whether processor i is active or inactive. An active processor i belongs to a group of processors, numbered from $L(i)$ to $U(i) - 1$ inclusive, which is making copies of the value that $L(i)$ originally read in step 4. All necessary memory accesses having been supported in step 8, the shortcutting is performed in step 9.

Analysis: Steps 1, 3, 4, 6, and 7 take $O(n/p + l)$ time. Steps 5 and 8 take $O(n/p + l \log n/\log l)$ time. Step 9 takes $O(nl/p + l)$ time. Step 2 takes $S(n, l, p)$ time, where $S(n, l, p)$ is the Block PRAM complexity of integer sorting n keys. Since the outputs of an integer sorting algorithm are sensitive on all variables, the lower bound of Theorem 2.2 applies to $S(n, l, p)$ and the overall complexity of the procedure is $O(S(n, l, p) + nl/p)$.

The above algorithm yields the following result.

THEOREM 4.8. *The connected components of a graph with n vertices and m edges can be found on the Block PRAM in time* $O(m \log n/p + nl \log^2 n/(p \log l) + S(n, l, p) \cdot \log n)$.

The following corollary shows that our simulation of the Shiloach and Vishkin (1982) CRCW PRAM algorithm is optimal for sufficiently dense graphs.

COROLLARY 4.9. *If there is a constant* $\varepsilon > 0$ *such that* $m = \Omega(ln^{1+\varepsilon})$, *then the connected components of a graph with n vertices and m edges can be found on the Block PRAM in time* $O(m \log n/p + l \log^2 n/\log l)$.

The biconnectivity algorithms of Tarjan and Vishkin (1985) and Tsin and Chin (1984) work by constructing an auxiliary edge graph whose connected components induce the biconnected components of the original graph. The construction of Tarjan and Vishkin is based on prefix computations and can be performed in $O(m \log n/p + l \log^2 n/\log l)$ time. A number of other fundamental problems have been shown to be reducible to finding connected or biconnected components by Awerbuch *et al.* (1987), Chin *et al.* (1982), Tarjan and Vishkin (1985), and Tsin and Chin (1984). Their results together with Theorem 4.8 yield

COROLLARY 4.10. *The following problems for undirected graphs on n vertices and m edges can all be solved on the Block PRAM in time $O(m \log n/p + nl \log^2 n/(p \log l) + S(n, l, p) \cdot \log n)$: finding biconnected components, finding separation vertices, finding bridges, finding the minimum spanning tree, finding an Euler circuit.*

5. CONCLUSION

As we attempt to implement PRAM algorithms on realistic models of computation, it will be worth considering the cost of converting EREW, CREW, and CRCW PRAM algorithms to the Block PRAM model with a view to improving the prospects for latency hiding. We have discussed some of the new intuitions which will be relevant in this analysis. Many avenues for further research are apparent.

- Having exhibited the value of the prefix sums primitive, Blelloch (1990) concludes by asking whether there might be other useful primitives with economical implementations on realistic architectures. The results in Aggarwal *et al.* (1989) demonstrate the usefulness of matrix transposition as a basic operation. We suggest that any further improvement in running times for the block PRAM model will yield useful primitives. We therefore believe that it will be fruitful to design algorithms specifically for the Block PRAM model.

- Our negative results (Theorem 2.2 and 3.4) represent communication-time tradeoffs in the spirit of Aggarwal *et al.* (1990), Papadimitriou and Ullman (1987), and Papadimitriou and Yannakakis (1988). The reduction of Theorem 3.4 suggests that permutation may become a problem of fundamental importance for the Block PRAM model of computation. Further examples of finely granular problems would be of interest.

- The best known lower bound on the time for a conservative Block PRAM algorithm to perform an *explicitly defined* permutation is $\Omega((n/p) \cdot \log \min(n/l, n/p)/\log(n/lp))$ for $lp \leqslant n$ (Aggarwal *et al.*, 1989). The bound is proved for transpositions of square matrices, which are among the easiest permutations for a Block PRAM to perform, using information-theoretic techniques. It is likely that more advanced methods will be required for improved lower bounds.

- Is path halving easier than list ranking? (Does it help to not have to compute distances?) Can list ranking be done faster, or at all, by some method other than shortcutting?

ACKNOWLEDGMENTS

We are grateful to an anonymous referee for a careful reading of an earlier version of this paper and to Dr. Colin McDiarmid for helpful discussions.

RECEIVED May 21, 1990; FINAL MANUSCRIPT RECEIVED March 30, 1992

REFERENCES

AGGARWAL, A., CHANDRA, A. K., AND SNIR, M. (1989), On communication latency in PRAM computations, *in* "Proceedings, First Annual ACM Symposium on Parallel Algorithms and Architectures," pp. 11–21.

AGGARWAL, A., CHANDRA, A. K., AND SNIR, M. (1990), Communication complexity of PRAMs, *Theoret. Comput. Sci.* **71**, 3–28.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, MA.

AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. (1983), Sorting in $c \log n$ parallel steps, *Combinatorica* **3**, 1–19.

ANDERSON, R. J., AND MILLER, G. L. (1988), Deterministic parallel list ranking, *in* "Proceedings, 3rd Aegean Workshop on Computing," pp. 81–90, Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin.

ANDERSON, R. J., AND MILLER, G. L. (1990), A simple randomized parallel algorithm for list ranking, *Inform. Process. Lett.* **33**, 269–273.

AWERBUCH, B., ISRAELI, A., AND SHILOACH, Y. (1987), Finding Euler circuits in logarithmic parallel time, *in* "Advances in Computing Research, Vol. 4: Parallel and Distributed Computing" (F. Preparata, Ed.), pp. 69–78, JAI Press, London.

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

BLELLOCH, G. (1990), "Vector Models for Data-Parallel Computing," MIT Press, Cambridge, MA.

CHIN, F. Y., LAM, J., AND CHEN, I.-N. (1982), Efficient parallel algorithms for some graph problems, *Comm. ACM* **25**, 659–665.

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

COLE, R., AND VISHKIN, U. (1986b), Approximate and exact parallel scheduling with applications to list, tree and graph problems, *in* "Proceedings, 27th Annual IEEE Symposium on Foundations of Computer Science," pp. 478–491.

COLE, R., AND VISHKIN, U. (1988), Approximate parallel scheduling. I. The basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput.* **17**, 128–142.

GAZIT, H., MILLER, G. L., AND TENG, S. H. (1987), Optimal tree contraction in the EREW model, *in* "Proceedings, 1987 Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation," pp. 139–156, Plenum, New York.

GIBBONS, P. B. (1989), "The Asynchronous PRAM: A Semi-synchronous Model for Shared Memory MIMD Machines," Ph.D. thesis, Department of Computer Science, University of California at Berkeley.

GIBBONS, A. M., AND RYTTER, W. (1988), "Efficient Parael Algorithms," Cambridge Univ. Press, Cambridge, UK.

HILLIS, W. D. (1985), "The Connection Machine," MIT Press, Cambridge, MA.

HIRSCHBERG, D. S. (1976), Parallel algorithms for the transitive closure and the connected components problems, *in* "Proceedings, 8th Annual ACM Symposium on Theory of Computing," pp. 55–57.

HIRSCHBERG, D. S. (1978), Fast parallel sorting algorithms, *Comm. ACM* **21**, 657–661.

KARP, R. M., AND RAMACHANDRAN, V. (1990), Parallel algorithms for shared-memory machines, "Handbook of Theoretical Computer Science" (J. van Leeuwen, Ed.), pp. 869–942, North-Holland, Amsterdam.

KRUSKAL, C. P., AND SMITH, C. H. (1988), Definitions of granularity, *in* "Proceedings, International Symposium on High Performance Computer Systems" (E. Gelenbe, Ed.), pp. 257–268, North-Holland, Amsterdam.

LADNER, R. E., AND FISCHER, M. J. (1980), Parallel prefix computation. *J. Assoc. Comput. Mach.* **27**, 831–838.

LEISERSON, C. E., AND MAGGS, B. M. (1988), Communication-efficient parallel algorithms for distributed random-access machines, *Algorithmica* **3**, 53–77.

MILLER, G. L. (1989), Are pointer-based parallel algorithms realistic?, *in* "Opportunities and Constraints of Parallel Computing" (J. L. C. Sanz, Ed.), p. 85, Springer-Verlag, New York.

PAPADIMITRIOU, C. H., AND ULLMAN, J. D. (1987), A communication-time trade-off, *SIAM Comput.* **16**, 639–647.

PAPADIMITRIOU, C. H., AND YANNAKAKIS, M. (1988), Towards an architecture-independent analysis of parallel algorithms, *in* "Proceedings, 20th Annual ACM Symposium on Theory of Computing," pp. 520–513.

PATERSON, M. S. (1990), Improved sorting networks with $O(\log n)$ depth, *Algorithmica* **5**, 75–92.

REIF, J. H. (1985), An optimal parallel algorithm for integer sorting, *in* "Proceedings, 26th Annual IEEE Symposium on Foundations of Computer Science," pp. 496–504.

SHILOACH, Y., AND VISHKIN, U. (1982), An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms* **3**, 57–67.

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

TSIN, Y. H., AND CHIN, F. Y. (1984), Efficient parallel algorithms for a class of graph theoretic problems, *SIAM J. Comput.* **13**, 269–272.

VALIANT, L. G. (1990a), General purpose parallel architectures, *in* "Handbook of Theoretical Computer Science" (J. van Leeuwen, Ed.), pp. 943–971, North-Holland, Amsterdam.

VALIANT, L. G. (1990b), A bridging model for parallel computation, *Comm. ACM* **33**, 103–110.

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

WYLLIE, J. C. (1979), "The Complexity of Parallel Computation," Ph.D. Thesis, Department of Computer Science, Cornell University, Ithaca, NY.