RANDOMIZED PARALLEL COMPUTATION

Sanguthevar Rajasekaran
Dept. of CISE, Univ. of Florida

John H. Reif
Dept. of CS, Duke University

ABSTRACT

The scope of this paper is to survey randomized parallel algorithms available for various computational problems. We do not claim to give a comprehensive list of all the randomized parallel algorithms that have been discovered so far. We only discuss representative examples which demonstrate the special features of both randomization and parallelization. The areas of CS we consider are: 1) routing and sorting, 2) processor load balancing, 3) algebra, and 4) graph theory. We also discuss in this paper various ways of derandomizing randomized parallel algorithms.

1 Introduction

1.1 History of Randomization

The technique of randomizing an algorithm to improve its efficiency was first introduced in 1976 independently by Rabin [59] and Solovay and Strassen [82]. Since then, this idea has been used to solve myriads of computational problems successfully. And today randomization has become a powerful tool in the design of both sequential and parallel algorithms.

Even though the idea of randomization is at least as old as Hoare's quicksort algorithm [28], these previous approaches presuppose a distribution on the space of all possible inputs, which is not a valid assumption at all. For example, Hoare's quicksort algorithm may run for a long period of time on certain inputs. But all such bad input permutations are only a small fraction. If we assume (which indeed Hoare does) that each input permutation is equally likely to occur, then quicksort algorithm is very well practical because with very high probability the given input permutation will not be a bad one and hence the algorithm will terminate quickly. But, Hoare's assumption of a uniform distribution on the input space is questionable, since the input distribution may vary quite unpredictably. [59] and [82] rectify this problem by introducing randomization into the algorithm itself.

Preprint of paper appearing in Chapter 11 of Concurrent Computations: Algorithms, Architecture and Technology, S.K. Tewksbury, B.W. Dickinson and S.C. Schwartz, ed., 1988, pp. 181 -202.

Informally, a randomized algorithm (in the sense of [59] and [82]) is one which bases some of its decisions on the outcomes of coin flips. We can think of the algorithm with one possible sequence of outcomes for the coin flips to be different from the same algorithm with a different sequence of outcomes for the coin flips. Therefore, a randomized algorithm is really a family of algorithms. For a given input, some of the algorithms in this family might run for an indefinitely long time. The objective in the design of a randomized algorithm is to ensure that the number of such bad algorithms in the family is only a small fraction of the total number of algorithms. If for *any* input we can find at least $(1-\epsilon)$ (ϵ being very close to 0) portion of algorithms in the family that will run quickly on that input, then clearly, a random algorithm in the family will run quickly on any input with probability $\geq (1-\epsilon)$. In this case we say that this family of algorithms (or this randomized algorithm) runs quickly with probability at least $(1-\epsilon)$. ϵ is called the error probability. Observe that this probability is independent of the input and the input distribution.

To give a flavor for the above notions, we now give an example of a randomized algorithm. We are given a polynomial of n variables $f(x_1, \ldots, x_n)$ over a field F. It is required to check if f is identically zero. We generate a random $n-$vector $(r_1, \ldots, r_n)$ ($r_i \in F, i = 1, \ldots, n$) and check if $f(r_1, \ldots, r_n) = 0$. We repeat this for k independent random vectors. If there was at least one vector on which f evaluated to a non zero value, of course f is nonzero. If f evaluated to zero on all the k vectors tried, we conclude f is zero. It can be shown (see section 2.3.1) that the probability of error in our conclusion will be very small if we choose a sufficiently large k. In comparison, the best known deterministic algorithm for this problem is much more complicated and has a much higher time bound.

1.2 Advantages of Randomization

Advantages of randomized algorithms are many. Two extremely important advantages are their simplicity and efficiency. A major portion of randomized algorithms found in the literature are extremely simpler and easier to understand than the best deterministic algorithms for the same problems. The reader would have already got a feel for this from the above given example of testing if a polynomial is identically zero. Randomized algorithms have also been shown to yield better complexity bounds. Numerous examples can be given to illustrate this fact. But we won't enlist all of them here since the algorithms described in the rest of the paper will convince the reader.

A skeptical reader at this point might ask: How dependable are randomized algorithms in practice, after all there is a non zero probability that they might fail? This skeptic reader must realize that there is a probability (however small it might be) that the hardware itself might fail. Adleman and Manders [1] remark that if we can find a fast algorithm for a problem with an error probability $< 2^{-k}$ for some integer k independent of the problem size, we can reduce the error probability far below the hardware error probability by making k large enough.

1.3 Randomization in Parallel Algorithms

The tremendously low cost of hardware nowadays has prompted computer scientists to design parallel machines and algorithms to solve problems very efficiently. In an early paper Reif [68] proposed using randomization in parallel computation. In this paper he also solved many algebraic and graph theoretic problems in parallel using randomization. Since then a new area of CS research has evolved that tries to exploit the special features offered by both randomization and parallelization.

This paper demonstrates the power of randomization in obtaining efficient parallel algorithms for various important computational problems.

1.4 Different types of randomized algorithms

Two types of randomized algorithms can be found in the literature: 1) those that always output the correct answer but whose run-time is a random variable with a specified mean. These are called Las Vegas algorithms; and 2) those that run for a specified amount of time and whose output will be correct with a specified probability. These are called Monte Carlo algorithms. Primality testing algorithm of Rabin [59] is of the second type.

The error of a randomized algorithm can either be 1-sided or 2-sided. Consider a randomized algorithm for recognizing a language. The output of the circuit is either *yes* or *no*. There are algorithms which when outputting *yes* will always be correct, but when outputting *no* they will be correct with high probability. These algorithms are said to have 1-sided error. Algorithms that have non zero error probability on both possible outputs are said to have 2-sided error.

1.5 Contents of this paper

The scope of this paper is to survey randomized parallel algorithms available for various computational problems. We do not claim to give a comprehensive list of all the randomized parallel algorithms that have been discovered so far. We only discuss representative examples which demonstrate the special features of both randomization and parallelization. The areas of CS we consider are: 1) routing and sorting, 2) processor load balancing, 3) algebra, and 4) graph theory. We also discuss in this paper various ways of derandomizing randomized parallel algorithms.

2 Parallel Machine Models

A large number of parallel machine models have been proposed. Some of the widely accepted models are: 1) fixed connection machines, 2) shared memory models, 3) the boolean circuit model, and 4) the parallel comparison trees. In the randomized version of these models, each processor is capable of making independent coin flips in addition to the computations allowed by the corresponding deterministic version. The *time complexity* of a parallel machine is a function of its input size. Precisely, time complexity is a function $g(n)$ that is the maximum over all inputs of size n of the time elapsed when the first processor begins execution until the time the last processor stops execution.

Just like big-O function serves to represent the complexity bounds of deterministic algorithms, $\widetilde{O}$ serves to represent the complexity bounds of randomized algorithms. We say a randomized algorithm has resource (time, space etc.) bound $\widetilde{O}(g(n))$ if there exists a constant c such that the amount of resource used by the algorithm (on any input of size n) is no more than $c\alpha g(n)$ with probability $\geq 1 - 1/n^{\alpha}$. We say a parallel algorithm is *optimal* if its processor bound P_n and time bound T_n are such that $P_nT_n = \widetilde{O}(S)$ where S is the time bound of the best known sequential algorithm for that problem. Next we describe the machine models.

2.1 Fixed connection networks

A fixed connection network is a directed graph $G(V, E)$ whose nodes represent processors and whose edges represent communication links between processors. Usually we assume that the degree of each node is either a constant or a slowly increasing function of the number of nodes in the graph.

2.2 Shared memory models

In shared memory models, a number (call it P) of processors work synchronously communicating with each other with the help of a common block of memory accessible by all. Each processor is a random access machine [3]. Each step of the algorithm is an arithmetic operation, a comparison, or a memory access. Several conventions are possible to resolve read or write conflicts that might arise while accessing the shared memory. EREW PRAM is the shared memory model where no simultaneous read or write is allowed on any cell of the shared memory. CREW PRAM is a variation which permits concurrent read but not concurrent write. And finally, CRCW PRAM model allows both concurrent read and concurrent write. Read or write conflicts in the above models are taken care of with a priority scheme. For a discussion of further variations in PRAM models see [21].

2.3 Boolean circuits

A Boolean circuit on n input variables is a directed acyclic graph (*dag*) whose nodes of indegree 0 are labelled with variables & their negations, and the other nodes (also called gates) are labelled with a boolean operation (like $\vee, \wedge$ etc.). The size of a circuit is defined to be the number of gates in the circuit. Fan-in and depth of a circuit C are defined to be the maximum indegree of any node, and the depth of the dag respectively. Let Σ be an alphabet set, and consider any language over this alphabet set, $W \subseteq \Sigma^n$. We say that the Boolean circuit C accepts W if on any input $w \in W$, C evaluates to 1. The circuit size complexity of W is defined to be the size of the smallest circuit accepting W.

Let $A \subseteq \Sigma^*$. The circuit size complexity of A is a function $g : n \rightarrow n$ such that $g(n)$ is the circuit size complexity of A^n where $A^n = A \cap \Sigma^n$. The above definition implies that a recognizer for a language is a family of Boolean circuits $< \alpha_n >$, each circuit of which accepts strings of a particular length. We say A has polynomial size circuits if its circuit size complexity is bounded above by a polynomial in the input size. The Boolean circuits defined above are language recognizers and have a single output. The notion of a Boolean circuit can easily be extended to Boolean circuits that compute an arbitrary function $f : \{0,1\}^n \rightarrow \{0,1\}^m$. The definitions of depth and size remain unchanged.

2.4 Parallel comparison trees

A parallel comparison tree model (proposed by Valiant [89]) is the same as a sequential comparison tree model of [3], except that in the parallel model at each node P comparisons are made. The computation proceeds to a child of the node, the child being determined by the outcomes of the comparisons made. This model is a much more powerful model than the ones seen before.

2.5 The Complexity Classes NC and RNC

We say a Boolean circuit family $< \alpha_n >$ is uniform if the description of α_n (for any integer n) can be generated by a deterministic Turing machine (TM) in space $O(\log n)$ with n as the input.

With the definition of uniformity at hand, we can define a new complexity class, viz., the set of all functions f computable by circuits of "small" depth. We define NC^k to be the set of all functions f computable by a uniform circuit family $< \alpha_n >$ with size $n^{O(1)}$ and depth $O(\log^k n)$. Also define $NC = \cup_k NC^k$.

A randomized Boolean circuit is the same a Boolean circuit except that each node of the former, in addition to performing a Boolean operation can also make a coin flip. A randomized Boolean circuit on n input variables is said to compute a function f if it outputs the correct value of f

with probability $> 1/2$. Define RNC^k to be the set of all functions computable by a uniform family of randomized Boolean circuits $< \alpha_n >$ with size $n^{O(1)}$ and depth $O(\log^k n)$. Also, define $RNC = \cup_k RNC^k$.

It should be mentioned that NC and RNC contain the same set of functions even when any of the standard shared memory computer models are used to define them. Most of the problems studied in this paper belong to RNC.

3 Randomized Parallel Algorithms for Routing and Sorting

How fast a parallel computer can run is determined by two factors: 1) how fast the individual processors can compute, and 2) how fast the processors can communicate among themselves. The rapid advance in VLSI technology makes it possible to increase the computational power of individual processors arbitrarily and hence the time efficiency of a parallel machine is essentially determined by the inter-processor communication speed. One of the main problems in the study of parallel computers has been to design an $N-$processor realistic parallel computer whose communication time is as low as permitted by the topology of its interconnection. A realistic parallel machine is understood to be a fixed connection network. In-degree and out-degree in this network must be low due to physical limitations.

Such a realistic parallel computer is not only a parallel machine on its own right, but also it can be used to simulate any of the ideal computers (like PRAMs) very efficiently. One step of communication in such a machine is: Each processor has a distinct packet of information that it wants to send to some other processor. The task (also called routing) is to route the packets to their destinations such that at the most one packet passes through any wire at any time, and all the packets arrive at their destinations quickly. A communication step ends when the last packet has reached its destination.

The problem of designing a realistic computer is then to come out with a topology for a fixed connection network and to design an efficient routing algorithm for that topology. A fixed connection network topology together with its routing function is called a *communication scheme.*

Valiant [90] was the first to present an efficient randomized routing algorithm for a fixed connection network called the binary $n-$cube. If N is the number of nodes in the network, the degree of the binary $n-$ cube is $n = O(\log N)$ and his routing algorithm runs in time $\widetilde{O}(\log N)$. This work was followed by Upfal's [88] who gave a routing algorithm with the same time bound for a constant degree network. Both these algorithms had a queue size (i.e., the maximum number of packets that any node will have at any time during execution) of $\widetilde{O}(\log N)$. Pippenger's [57] routing algorithm for a constant degree fixed connection network not only had the same time complexity but also it had a queue size of $O(1)$. Details of these algorithms are given in section 2.1.1.

In section 2.1.2 we will describe some of the important randomized sorting algorithms found in the literature. Sorting is the process of rearranging a sequence of values in ascending or descending order. Many application programs like compilers, operating systems etc. use sorting extensively to handle efficiently tables and lists. Both due to its practical value and theoretical interest, sorting has been an attractive area of research in CS.

A large number of deterministic parallel algorithms have been proposed for special purpose networks called *sorting networks* (see [8] and [93]). Of these the most efficient algorithm is due to Batcher [8]. His algorithm sorted N numbers in time $O(\log^2 N)$ time using $N \log^2 N$ processors. Designing an algorithm to sort N numbers in $O(\log N)$ parallel time using a linear number of processors was a long open problem. Reischuk [75] discovered a randomized algorithm that employed N PRAM processors and ran in time $\widetilde{O}(\log N)$. This algorithm is impractical owing to its large memory requirements. It was Reif and Valiant [73] who first gave a practical sorting algorithm (FLASHSORT) that was optimal and ran on a network called cube connected cycles (CCC). This was a randomized algorithm that used N processors to sort N numbers in time $\widetilde{O}(\log N)$. Since $\Omega(N \log N)$ is a well known lower bound [3] for sequential comparison sorting of N keys, their algorithm is indeed optimal. A summary of this algorithm appears in section 2.1.2.

When the keys to be sorted are from a finite set, sorting becomes simple. Bucket-sort algorithm [3] can be used to sort N integer keys (keys in the range $[1, N]$) in N sequential steps. Does there exist an optimal parallel algorithm for sorting integer keys (INTEGER SORT)? In otherwords, is there a parallel algorithm that can sort N integer keys in $O(\log N)$ time using only $N/\log N$ processors? This question is answered in the affirmative by the randomized PRAM algorithm of Rajasekaran and Reif [62]. Details of this algorithm will also appear in section 4.3.

It should be mentioned here that deterministic algorithms have been found to sort N general keys on a constant degree network of linear size in time $O(\log N)$ by AKS [4] and Leighton [46]. Unfortunately these algorithms have large constants in their time bounds making them impractical. This is a good instance where randomization seems to help in obtaining practical parallel algorithms (remember [73]'s FLASHSORT has better constants). Recently Cole [14] has given a deterministic optimal sorting algorithm for the CRCW PRAM model that has a small constant in its time bound. But obtaining an optimal sorting algorithm (either deterministic or randomized) on any fixed connection network with a small constant in the time bound is still an open problem.

3.1 Routing Algorithms

3.1.1 Valiant's algorithm

Valiant's routing algorithm [90] runs on an $n-$cube. Each node in the cube is named by an $n-$bit binary vector $(x_1, \ldots, x_n)$. There are 2^n nodes in the network and the degree of each node is n. For any node $e = (x_1, \ldots, x_n)$, let $\overline{e_i}$ stand for $(x_1, \ldots, x_{i-1}, \overline{x_i}, x_{i+1}, \ldots, x_n)$, where $\overline{x_i}$ is the complement of x_i. Every node e is connected to n neighbours one in each dimension, i.e., e is connected to $\overline{e_i}, i = 1, \ldots, n$.

The algorithm consists of two phases. In the first phase, each packet is sent to a random destination and in the second phase packets in the random destinations are sent to their correct destinations. Each phase runs in time $\widetilde{O}(n)$ and hence the whole algorithm runs in the same time bound.

In phase I, at each time unit, every packet p chooses one of the dimensions at random. It decides to traverse along this direction with probability 1/2. Once a packet chooses a dimension, it will not choose it again. If a packet decides to traverse along the chosen dimension, it will enter a queue in its current location. In phase II, each packet does the same thing, except that now the set of dimensions to be traversed by a packet is determined by the shortest path between its destination and the node that it starts its second phase from. Also, once a packet chooses a dimension in phase II, it will traverse along it.

Each node contains n queues one for packets to be transmitted along each dimension. Each packet p is associated with a set $U_p \subseteq \{1, 2, \ldots, n\}$. In phase I, it consists of the set of dimensions along which possible transmissions have not yet been considered, and in phase II it consists of dimensions along which transmission still has to take place. Each of the phases is said to be finished when U_p is empty for every p. One of the special features of Valiant's algorithm is that it is *oblivious*, i.e., the route taken by any packet is independent of the routes taken by the other packets. This property is important for an algorithm to be applied in a distributed environment.

3.1.2 Upfal's algorithm

Upfal's [88] routing algorithm is applicable to a class of communication schemes he calls *Balanced Communication Schemes* (BCS). Here, we will consider only an example of a BCS. The topology of this communication scheme is a 2-way Digit Exchange Graph (2DEG). The number of nodes in the graph (N) is $m2^m$. Here again each processor is named by a binary vector of length $\log m + m$. The m rightmost bits of a node's name is called its *address* and the other bits constitute its *prefix*. The degree of each node is 2. The two edges leaving the processor with an address $b_0, \ldots, b_\alpha, \ldots, b_{m-1}$ and a preifx α are connected to processors with the addresses

$b_0, \ldots, b_\alpha, \ldots, b_{m-1}$ and $b_0, \ldots, \overline{b_\alpha}, \ldots, b_{m-1}$ and prefixes $(\alpha + 1) \bmod m$. Each group of processors with the same prefix form one *stage* of the network.

Upfal's algorithm is also a two phase algorithm, where packets are sent to random destinations in the first phase and from random destinations to their actual destinations in the second phase. Packets are given priority numbers, which are integers in the range $[1, 3m - 1]$. The priority of a packet strictly increases on transitions. At any node packets with the least priority numbers are given precedence over the others when their is a contention for an outgoing edge. In otherwords, packets that have travelled less distance are given priority over those that have travelled more.

Consider a packet initially located in a processor with prefix α and which is destined for a processor with address $b_0, \ldots, b_{m-1}$ and prefix β. The first phase of the algorithm is performed in two steps. First step of phase A takes the packet to a random destination with the same prefix α. The packet undergoes m transitions. At each time, the packet chooses randomly one of the edges that leave the node the packet is currently in. Since there are m stages in the network, the packet will end up in a node with prefix α. In the second step of phase A, the packet traverses to a random address in stage (prefix) β. This is done by $(\beta - \alpha) \bmod m$ transitions. Here again, the packet chooses a random edge leaving its current location.

Phase B takes the packet to its final destination. At a transition leaving processor with prefix i, the packet can enter either a processor with the same address or a processor with an address different in the ith bit. The packet traverses according to the ith bit in its destination address. Thus, m transitions are sufficient in phase B. The analysis of this algorithm is done using a new technique called *critical delay sequences*. The algorithm is shown to run in time $\widetilde{O}(m)$.

Pippenger's [57] communication scheme has a topology of d-way Digit Exchange Graph. It is very much similar to a 2DEG. It has kd^k nodes and degree d. His algorithm successfully routes N packets in time $\widetilde{O}(\log N)$. This algorithm also has a maximum queue size of $O(1)$. Both Upfal's and Pippenger's algorithm are oblivious. Recently Ranade [60] has given a simple algorithm for constant queue routing on a Butterfly network.

3.2 Sorting Algorithms

In this section we will discuss [73]'s FLASHSORT algorithm. The sorting problem is: Given a set K of N keys $\{k_1, \ldots, k_N\}$ and a total ordering $<$ on them. To find a permutation $\sigma = (\sigma(1), \ldots, \sigma(N))$ such that $k_{\sigma(1)} < \ldots < k_{\sigma(N)}$. Define the rank of a key k to be $\text{rank}(k) = |\{k' \in K / k' < k\}|$.

3.2.1 FLASHSORT

FLASHSORT runs on a fixed connection network called the cube connected cycles (CCC) having N nodes with labels from $\{0, 1, \ldots, N-1\}$ and constant degree. A CCC is nothing but an n-cube with each one of its nodes being replaced by a cycle of n nodes. The n edges leaving each n-cube node now leave from distinct nodes of the corresponding cycle. The CCC thus has $n2^n$ nodes.

Each of the nodes of the CCC initially contains a key. FLASHSORT sorts these keys by routing each packet $k \in K$ to a node $j\ = \text{rank}(k)$. The algorithm consists of 4 steps. Step 1 finds a set of $2^n/n^{\epsilon}, \epsilon < 1$ elements called the *splitters* that divide K when regarded as an ordered set, into roughly equal intervals. In step 2, keys in each interval (determined by the splitters) are routed to the sub-cube they belong to. This routing task is achieved using the two phase algorithm of [90] described in section 2.1.1. After the second step, the keys will be approximately sorted. In step 3, the rank of each key is determined and finally in step 4, each packet is routed to the node corresponding to its rank.

In the above algorithm it is assumed that each node has a local memory of $O(\log N)$. The algorithm has a time bound of $\widetilde{O}(\log N)$ and is optimal.

4 Randomized Algorithms for Load Balancing

If the number of arithmetic (or comparison) operations to be performed by an algorithm is S and if we employ P processors to run the algorithm in parallel, then the best run time possible is S/P. This best run time occurs when the work load is shared equally by all the processors. In this case we obtain an optimal algorithm. Optimal parallel algorithms with run time $O(\log N)$ are of special importance and these are called linear parallel algorithms. In this section we will discuss some of the linear parallel algorithms found in the literature.

4.1 Parallel Tree Contraction

Tree problems arise frequently in CS. A simple example is evaluating an arithmetic expression, which can be viewed as tree evaluation. Miller and Reif [54] give an optimal algorithm for parallel tree contraction. Some of the many applications of this algorithm are: dynamic expression evaluation, testing for isomorphism of trees, list ranking, computing canonical forms for planar graphs etc. For the list ranking problem Cole and Vishkin [15] have presented an optimal deterministic algorithm. For evaluating arithmetic expressions of length N Brent [11] has given a parallel algorithm that employs N PRAM processors and runs in time $O(\log N)$. His algorithm needs a lot of preprocessing. If no preprocessing is for free, then his algorithm seems to require $O(\log^2 N)$ time.

The problem of evaluating an arithmetic expression without any preprocessing is called *dynamic expression evaluation.*

An arithmetic expression is defined as follows. 1) Constants are arithmetic expressions; 2) If E_1 and E_2 are arithmetic expressions, then so are $E_1 \pm E_2$, $E_1 * E_2$, and E_1/E_2. As one can see, an arithmetic expression can be represented as a binary tree whose internal nodes are operators and whose leaves are constants. This tree is called an expression tree.

[11]'s algorithm first finds (by preprocessing) a *separator set* for the expression tree. A *separator set* is a set of edges which when removed from the tree will divide the tree into almost equal sized disjoint sub-trees. The algorithm evaluates the sub-expressions defined by the separator set in parallel and combines the results. Since the size of the sub-trees decreases by a constant factor at each step, only $O(\log N)$ steps are needed. This algorithm, clearly, is a top down algorithm. [54] give a bottom up algorithm for dynamic expression evaluation.

They define two operators viz., RAKE and COMPRESS on rooted trees such that at most $O(\log N)$ of these operations are needed to reduce any tree with N nodes to a single node. They also give an efficient implementation of these operations on a CRCW PRAM.

Let $T = (V, E)$ be a rooted tree with N nodes and root r. Let RAKE be the operation of removing all the leaves from T (this operation intuitively corresponds to evaluating a node if all of its children have been evaluated or partially evaluating a node if some of its children have been evaluated). We say a sequence of nodes $v_1, \ldots, v_k$ is a *chain* if v_{i+1} is the only child of v_i for $1 \leq i \leq k$ and v_k has exactly one child that is not a leaf. Let COMPRESS be the operation on T that contracts all the maximal chains of T in one step. If a node has been partially evaluated except for one child, then the value of the node is a linear function of the child, say $aX + b$, where X is a variable. Thus a chain is a sequence of nodes each of which is a linear function of its child.

Let CONTRACT be the simultaneous application of RAKE and COMPRESS. [54] prove only $O(\log N)$ CONTRACT operations are necessary to reduce T to its root. Their CRCW PRAM randomized algorithm runs in time $\widetilde{O}(\log N)$ employing $O(N/\log N)$ processors.

4.2 Parallel Hashing

PRAM models of parallel computation are powerful, elegant, and algorithms written for PRAM machines are simpler. But these models are not practical. One of the problems in the theory of parallel computation is to simulate shared memory on a more realistic computer viz., a fixed connection network. Karlin and Upfal's [32] randomized scheme for implementing shared memory on a butterfly network enables N processors to store and retrieve an arbitrary set of N data items in $\widetilde{O}(\log N)$ parallel steps. Since a butterfly network has a constant degree, this time bound is optimal. A number of authors have worked on this problem before. The best previous result was due to [6] who showed that T arbitrary PRAM steps can be simulated in a bounded degree network in $\widetilde{O}(T \log^2 N)$ time. [32]'s algorithm improves this bound to an optimal $\widetilde{O}(T \log N)$.

[32]'s PRAM simulation is based on a scheme they call *parallel hashing*. The scheme combines the use of universal hash functions [12] for distributing the variables among the processors and a randomized algorithm (very much similar to the one given in section 2.1.1) for executing memory requests. Simulation of a single step of PRAM step involves communication between each processor that wishes to access a variable and the processor storing that variable. A single PRAM instruction is executed by sending messages to the processor storing the variable that each processor wishes to access and back in the case of read instruction.

Ranade [60] also gives an optimal algorithm for the above problem which is better than [32]'s.

Other examples of optimal algorithms are: INTEGERSORT of Rajasekaran and Reif [62], connectivity algorithm of Gazit [24], etc.

5 Parallel Randomized Algorithms in Algebra

Examples considered in this section are: 1) testing a polynomial for identity; 2) testing if $AB = C$ for integer $N \times N$ matrices A, B, and C; and 3)testing for the existence of a perfect matching in a graph.

5.1 Testing for Polynomial Identity

Given a polynomial $Q = Q(x_1, \ldots, x_N)$ in N variables over the field F. It is desired to check if Q is identically 0. If I is any set of elements in the field F such that $|I| \geq c$ degree(Q), then the number of elements of $I \times \ldots \times I$ which are zeros of Q is at most $|I|^N/c$. This fact suggests the following randomized algorithm for solving the given problem: 1) choose an I such that $|I| >$ degree(Q) and $c > 2$; 2) choose at random m elements of $I \times \ldots \times I$ and on each vector of values check if the polynomial is non zero. If on all these m vectors Q evaluates to zero, conclude Q is identically zero. If on at least one vector Q evaluates to a non zero value, then Q is non zero.

If $c > 2$ and Q is non zero, probability that a random vector of $I \times \ldots \times I$ is a zero of Q is at most 1/2. Therefore, probability that only zeros of Q are picked in m independent steps is at most 2^{-m}. If $m = O(\log N)$, this probability is at most $N^{-\alpha}, \alpha > 1$. This randomized algorithm can easily be parallelized. We have $N \log N$ collection of processors, each collection chooses a random N-vector from $I \times \ldots \times I$ and evaluates Q on it. The results are combined later. Since we can evaluate a multivariate polynomial in $O(\log N)$ time using a polynomial number of processors, the entire algorithm runs in time $\widetilde{O}(\log N)$ using $N^{O(1)}$ processors.

5.2 Is AB = C?

Given $N \times N$ integer matrices A, B, and C. To test if $AB = C$. Reif [68] shows that a randomized PRAM with time bound $\widetilde{O}(\log N)$ and processor bound $N^2/\log N$ can solve this problem. The idea is to choose randomly and independently m column vectors $x \in \{-1, 1\}^N$ and test if $A(Bx) = Cx$. This test is done by a randomized PRAM within time $\widetilde{O}(\log N)$ and $(N^2/\log N)$ processors by forming $N/\log N$ binary trees of processors, each of size $2N$ and depth $O(\log N)$ and pipelining the required dot products.

Freivalds [23] shows that if $AB \neq C$, for a random x, probability$[A(Bx) = Cx] < 1/2$. And hence, the probability of error in the above algorithm can be made arbitrarily small, by choosing a sufficiently large m.

5.3 Testing for the Existence of a Perfect Matching

Given an undirected graph $G = (V, E)$ with vertices $V = \{1, \ldots, N\}$. To test if G has a perfect matching. Tutte [87] showed that a graph has a perfect matching iff a certain matrix of indeterminates called the *Tutte matrix* is non-singular. The Tutte matrix M is defined as: 1) $M_{ij} = x_{ij}$ if $(i, j) \in E$ and $i < j$; 2) $M_{ij} = -x_{ij}$ if $(i, j) \in E$ and $i > j$; and 3) $M_{ij} = 0$ otherwise. Here $x_{ij}, j = 1, \ldots, N$ and $i = 1, \ldots, j - 1$ are indeterminates. Since the determinant of M is a multivariate polynomial, we can test if it is identically zero in parallel quickly using the algorithm of section 2.3.1.

6 Randomized Parallel Graph Algorithms

Graph theory finds application in every walk of life, more so in the field of computer science. Efficient sequential algorithms have been found for numerous graph problems. But unfortunately, not many efficient parallelization of these algorithms have been made. Randomization seems to play an important role in parallelizing graph algorithms as evidenced from the literature. In this section we will demonstrate this with representative examples. In particular, we will look at: 1) Symmetric complementation games of Reif [69]; 2) the random mating lemma of [70]; 3) the depth first search algorithm of Aggarwal and Anderson [2]; and 4) the maximal independent set algorithm of Luby [49].

6.1 Symmetric Complementation

Reif [69] has discovered a class of games he calls *symmetric complementation games*. These games are interesting since their related complexity classes include many well known graph problems.

Some of these problems are: 1) finding minimum spanning forests; 2) k-connectivity; 3) k-blocks; 4)recognition of chordal graphs, comparability graphs, interval graphs, split graphs, permutation graphs, and constant valance planar graphs. For all these problems [69] gives sequential algorithms requiring simultaneously logarithmic space and polynomial time. Furthermore, he also gives randomized parallel algorithms requiring simultaneously logarithmic time and a polynomial number of processors thus showing that all these problems are in RNC.

6.2 The Random Mating Lemma and Optimal Connectivity

Let $G = (V, E)$ be any graph. Suppose we assign for each vertex $v \in V$, independently and randomly $\text{SEX}(v) \in \{male, female\}$. Let vertex v be active if there exists at least one departing edge $(v, u) \in E (v \neq u)$. We say the vertex v is mated if $\text{SEX}(v) = male$ and $\text{SEX}(u) = female$ for at least one edge $(v, u) \in E$. Then, the Random Mating Lemma of [70] states that with probability $1/2$ the number of mated vertices is at least $1/8$ of the total vertices.

The random mating lemma naturally leads to an elegant randomized algorithm [70] for the connectivity problem, i.e., the problem of computing the connected components of a graph, that has a run time of $\widetilde{O}(\log |V|)$ on the PRAM model using $|V| + |E|$ processors. In an earlier thesis Shiloach and Vishkin [80] have presented a deterministic algorithm for connectivity that has the same resource bounds. But, the random mating lemma has been used by Gazit [24] to obtain an optimal randomized algorithm for connectivity that runs in time $\widetilde{O}(\log |V|)$ and uses $(|V| + |E|)/\log(|V|)$ processors.

6.3 Depth First Search

The problem of performing Depth First Search (DFS) in parallel was studied by many authors ([67],[17], etc.) and they conjectured that DFS was inherently sequential. However, NC algorithms were given for DFS of some restricted class of graphs by Smith [81] for planar graphs, and Ghosh and Bhattacharjee [25] for directed acyclic graphs. It was Aggarwal and Anderson [2] who first gave an RNC algorithm for DFS.

The DFS problem is: Given a graph $G = (V, E)$ and a vertex r, construct a tree T that corresponds to a depth first search of the graph starting from the vertex r. The DFS algorithm of [2] is a divide and conquer algorithm. They first find an initial portion of the DFS tree which allows them to reduce the problem to DFS in trees of less than half the original size. This algorithm has $O(\log N)$ levels of recursion.

An initial segment is a rooted subtree T' that can be extended to some DFS tree T. They give an RNC algorithm to construct an initial segment with the property that the largest component of $V - T'$ (i.e., G after removal of T' from it) has size at most $N/2$ (N being th size of the original graph). This initial segment T' is then extended to a DFS as follows. Let C be a connected component of $V - T'$. There is a unique vertex $x \in T'$ of greatest depth that is adjacent to some vertex of C. Let $y \in C$ be adjacent to x. Construct a DFS tree for C rooted at y. Connect this tree to T' using the edge from x to y. This is performed independently for each component. This algorithm is applied recursively to finding DFS of components of $V - T'$. Since finding an initial segment is in RNC, the whole algorithm is in RNC.

6.4 Maximal Independent Set

Given an undirected graph $G(V, E)$. The problem is to find a maximal collection of vertices such that no two vertices in the collection are adjacent. There is a trivial linear time sequential algorithm for this problem [3]. An efficient parallel algorithm did not exist for this problem until Karp and Wigderson [34] presented a randomized algorithm that utilized $O(N^2)$ EREW PRAM processors and ran in time $\widetilde{O}(\log^4 N)$. Luby [49] later gave a simplified randomized algorithm with a time bound of $\widetilde{O}(\log^2 N)$ and processor bound $O(M)$ (where M is the number of edges and N is the number of vertices in G). This algorithm also has the property that it can be made deterministic. Some details of this property will be discussed in section 2.5.2.

A summary of [49]'s algorithm for the maximal independent set (MIS) problem: This is an iterative algorithm. At each step of the algorithm, certain number of nodes will be decided to be in the MIS and certain number of edges will be deleted from G. The algorithm iterates on the resultant graph $G'(V', E')$. At each step of the algorithm at least a constant fraction of edges will be removed. Therefore, the algorithm will terminate after $\widetilde{O}(\log N)$ iterations.

At any iteration of the algorithm that starts with $G'(V', E')$, every node $v \in V'$ decides (independently) to add itself to the MIS with probability $1/2d(v)$, $d(v)$ being its degree. Let I' be the set of all nodes that decide to add themselves to the MIS in this step. Check if there is an edge between any pair of vertices in I'. For every such edge, the node with the smaller degree is deleted from I'. A tie is broken arbitrarily. Add the resultant I' to the MIS. And finally delete all the nodes in I' and its neighbours from G'. The resultant graph is what the next iteration of the algorithm starts with.

Since each step of the algorithm gets rid of a constant fraction of edges, the algorithm terminates in $\widetilde{O}(\log N)$ iterations. Also notice that at any step of the algorithm no more than $O(M)$ processors are needed. Each step of the algorithm takes $\widetilde{O}(\log N)$ time and hence the whole algorithm runs in time $\widetilde{O}(\log^2 N)$.

7 Derandomization of Parallel Randomized Algorithms

Computer Scientists have made some effort to make randomized algorithms deterministic owing mainly to the fact that our computers are deterministic and are unable to generate truly random numbers. Four techniques have been discovered so far: 1) the usage of psuedo-random numbers in the place of truly random numbers; 2) reducing the size of the probability space so an exhaustive search in the space can be done; 3)deterministic coin tossing; and 4) combinatorial construction. No work has been done to parallelize the fourth technique. This section is devoted to introducing the reader to the first three methodologies as applicable to parallel algorithms.

7.1 Psuedo-Random Number Generation

Informally, a psuedo-random sequence is a sequence of bits generated by a deterministic program from a random *seed* such that a computer with "limited" resources won't be able to distinguish it from a truly random sequence. Examples of psuedo-random number generators include linear congruential generators, additive number generators, etc. [36]. All of the randomized algorithms that are currently being used in practice use only psuedo-random generators of one type or the other in the place of true random generators.

In [72] a parallel NC algorithm is given which can generate N^c (for any $c > 1$) psuedo-random bits from a seed of N^ϵ, ($\epsilon < 1$) truly random bits. This takes polylog time using $N^{\epsilon'}$ processors where $\epsilon' = k\epsilon$ for some fixed constant $k > 1$. The psuedo-random bits output by this algorithm can not be distinguished from truly random bits in polylog time using a polynomial number of processors with probability $\geq \frac{1}{2} + \frac{1}{N^{O(1)}}$ if the multiplicative inverse problem can not be solved in RNC.

As a corollary to their algorithm, they also show that given any parallel algorithm (over a wide class of machine models) with time bound $T(N)$ and processor bound $P(N)$, it can be simulated by a parallel algorithm with time bound $T(N) + O(\log N \ \log\log N)$, processor bound $P(N)N^{\epsilon'}$, and only using N^ϵ truly random bits.

7.2 Probability Space Reduction

Any randomized algorithm generates certain number (say R) of mutually independent random numbers (say in the range $[1, L]$) during its execution. Any randomized algorithm can be made deterministic by running it on every possible sequence of R numbers (in the range $[1, L]$). The set of all these sequences (that constitute a probability space) usually is of exponential size. There are algorithms [49] which will run even if the random numbers generated are only pairwise independent.

When the random numbers are only pairwise independent, the size of the probability space might reduce tremendously enabling us to do an exhaustive search quickly.

In Luby's [49] randomized algorithm for MIS (section 2.4.4), remember, if the program starts an iteration with the graph $G'(V', E')$, each node $v \in V'$ decides to add itself to the MIS with a probability $1/2d(v)$, $d(v)$ being the degree of v. It was assumed that the decisions of the nodes were mutually independent. It turns out that the algorithm runs correctly with the same time and processor bounds even if they are only pairwise independent.

[49] also constructs a probability space of size $O(N^2)$ (N being the number of nodes in the input graph) and defines N pairwise independent events $E_1, \ldots, E_N$ in this space such that Prob.$[E_v] = 1/2d(v)$ for each $v \in V$. This probability space then makes his algorithm deterministic with the same time bound and a processor bound of $O(MN^2)$. This idea has been extended to k-wise ($k \geq 2$) independent events by [33]. They also show that using $k \log N$ random bits we can construct a sequence of N random bits such that every k bits are mutually independent.

7.3 Deterministic Coin Tossing

Consider the following problem: Given a connected directed graph $G(V, E)$. The indegree and outdegree of each vertex is 1. Such a graph forms a directed circuit and is called a *ring*. We define a subset U of V to be an *r-ruling set* of G if: 1)no two vertices of U are adjacent; and 2) for each vertex $v \in V$ there is a directed path from v to some vertex in U whose edge length is at the most r. The problem is to find an r-ruling set of G for a given r.

We can use the following simple randomized algorithm: 1) in the first step, each node in the graph chooses to be in the ruling set with a probability of $1/r$; and 2) in the second step each group of adjacent (in G) nodes chosen in step1, randomly choose one of them to be in the ruling set.

Cole and Vishkin [15] give a deterministic algorithm for obtaining an r-ruling set. We will consider here only the case $r = \log n$. Their algorithm finds a $\log n$-ruling set in $O(1)$ time on the EREW PRAM model.

input representation: The vertices are given in an array of length n. The entries of the array are numbered from 0 to $n-1$ (each being a $\log n$ bit binary number). Each vertex has a pointer to the next vertex in the ring.

algorithm Processor i is assigned to entry i of input array (for simplicity entry i is called the vertex i). Let $SERIAL_0(i) = i$ for $i = 0, \ldots, n-1$. Let i_2 be the vertex following i in the ring and j be the index of the right most bit in which i and i_2 differ. Processor i sets $SERIAL_1(i) = j, i = 0, \ldots, n-1$.

Note that for all i, $SERIAL_1(i)$ is a number between 0 and $\log n - 1$. We say $SERIAL_1(i)$ is a local minimum if $SERIAL_1(i) \leq SERIAL_1(i_1)$ and

$SERIAL_1(i) \leq SERIAL_1(i_2)$ (where i_1 and i_2 are vertices preceding and following i respectively). Define a local maximum similarly. It can be seen that the number of vertices in the shortest path from any vertex in G to the next local extremum is at the most $\log n$. Thus the extrema satisfy condition 2 for the $\log n$ ruling set. From among local extrema, there might be *chains* of successive vertices in G. [15] pick a unique node deterministically from each such chain. Thus a subset of all the extrema forms a $\log n$-ruling set.

[15] call the above technique *deterministic coin tossing*. They use this technique to solve many graph problems like list ranking, selecting the nth smallest out of n elements, finding minimum spanning forest in a graph etc.

8 Conclusion

Randomization is a powerful tool in developing parallel algorithms which have small processor and time bounds. It also appears to significantly simplify the structure of parallel algorithms. This fact has been demonstrated in this paper. Techniques for making randomized algorithms deterministic have also been explored.

References

[1] Adleman,L., and Manders,K., 'Reducibility, Randomness and Untractability,' Proc. 9th ACM Symposium on Theory Of Computing, 1977, pp.151-163.

[2] Aggarwal,A., and Anderson,R., 'A Random NC Algorithm for Depth First Search,' Proc. of the 19th annual ACM Symposium on Theory Of Computing, 1987, pp.325-334.

[3] Aho,A.U., Hopcroft,J.E., Ullman,J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Massachusetts, 1977.

[4] Ajtai,M., Komlós,J., and Szemerédi,E., 'An $O(n \log n)$ Sorting Network,' Combinatorica, 3, 1983, pp.1-19.

[5] Alon,N. and Azar, Y., "The Average Complexity of Deterministic and Randomized Parallel Comparison Sorting Algorithms," Proc. IEEE Symposium on Foundations of Computer Science, 1987, pp.489-498.

[6] Alt, H., Hagerup, T., Mehlhorn, K., and Preparata, F.P., "Simulations of Idealized Parallel Computers on more Realistic ones,' Preliminary report, 1985.

[7] Angluin,D., and Valiant,L.G., “Fast Probabilistic Algorithms for Hamiltonian Paths and Matchings,” Journal of Computer and Systems Science, 18, 1979, pp.155-193.

[8] Batcher,K.E., ‘Sorting Networks and their Applications,’ Proc. 1968 Spring Joint Computer Conference, vol.32, AFIPS Press, 1968, pp.307-314.

[9] Beame,P., and Hastad,J., “Optimal Bounds for Decision Problems on the CRCW PRAM,” Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp.83-93.

[10] Borodin and Hopcroft, “Routing, Merging, and Sorting on Parallel Models of Computation,” Proc. 14th Annual ACM Symposium on Theory of Computing, 1982, pp.338-344.

[11] Brent,R.P., ‘The Parallel Evaluation of Generalized Arithmetic Expressions,’ Journal of ACM, vol.21, no.2, 1974, pp.201-208.

[12] Carter,L., and Wegman,M., ‘Universal Class of Hash Functions,’ Journal of CSS, vol.18, no.2, 1979, pp.143-154.

[13] Chernoff,H., “A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations,” Annals of Math. Statistics 23, 1952, pp.493-507.

[14] Cole,R., ‘Parallel Merge Sort,’ Proceedings of the IEEE Foundations Of Computer Science, 1986, pp.511-516.

[15] Cole,R., and Vishkin,U., ‘Approximate and Exact Parallel Scheduling with Applications to List, Tree, and Graph Problems,’ Proc. of the IEEE Foundations Of Computer Science, 1986, pp.478-491.

[16] Deo,N., *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall Publishing Company, New York, 1974.

[17] Eckstein,D., and Alton,D., ‘Parallel Graph Processing using Depth First Search,’ Proc. Conference on Theoretical Computer Science at Univ. of Waterloo, 1977, pp.21-29.

[18] Feller,W., *An Introduction to Probability Theory and its Applications*, John Wiley & sons Publishing Company, New York, 1968.

[19] Fich,F.E., “Two Problems in Concrete Complexity Cycle Detection and Parallel Prefix Computation,” Ph.D. Thesis, Univ. of California, Berkeley, 1982.

[20] Floyd and Rivest, “Expected Time Bounds for Selection,” Communications of the ACM, vol.18, no.3, 1975, pp.165-172.

[21] Fotune,S., and Wyllie,J., ‘Parallelism in Random Access Machines,’ Proc. 10th Annual ACM Symposium on Theory Of Computing, 1978, pp.114-118.

[22] Frazer and McKellar, “Samplesort: A Sampling Approach to Minimal Storage Tree Sorting,” Journal of the ACM, vol.17, no.3, 1977, pp.496-502.

[23] Freivalds, ‘Fast Probabilistic Algorithms,’ 8th MFCS, 1979.

[24] Gazit,H., ‘An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph,’ Proc. IEEE Symposium on Foundations Of Computer Science, 1986, pp.492-501.

[25] Ghosh,R.K., and Bhattacharjee,G.P., ‘A Parallel Search Algorithm for Directed Acyclic Graphs,’ BIT, vol.24, 1984, pp.134-150.

[26] Hagerup,T., “Towards Optimal Parallel Bucket Sorting,” Information and Computation 75, 1987, pp.39-51.

[27] Harary,F., *Graph Theory*, Addison-Wesley Publications, Massachusetts, 1969.

[28] Hoare,C.A.R., ‘Quicksort,’ Computer Journal, vol.5, no.1, 1962, pp.10-15.

[29] Hoeffding,W., “On the Distribution of the Number of Successes in Independent Trials,” Annals of Math. Statistics 27, 1956, pp.713-721.

[30] Hopcroft,J.E., and Tarzan,R.E., “Efficient Algorithms for Graph Manipulation,” Communications of the ACM, vol.16, no.6, 1973, pp.372-378.

[31] Johnson,N.J., and Katz,S., *Discrete Distributions*, Houghton Miffin Company, Boston, Massachusetts, 1969.

[32] Karlin,A.R., and Upfal,E., ‘Parallel Hashing–An Efficient Implementation of Shared Memory,’ Proc. 18th Annual ACM Symposium on Theory Of Computing, 1986, pp.160-168.

[33] Karp,R.M., Upfal,E., and Wigderson,A., ‘The Complexity of Parallel Computation on Matroids,’ IEEE Symposium on Foundations Of Computer Science, 1985, pp.541-550.

[34] Karp,R.M., and Wigderson,A., ‘A Fast Parallel Algorithm for the Maximal Independent Set Problem,’ Proc. 16th Annual ACM Symposium on Theory Of Computing, 1984, pp.266-272.

[35] Kirkpatrick,D., and Reisch,S., “Upper Bounds for Sorting Integers on Random Access Machines,” Theoretical Computer Science, vol28, 1984, pp.263-276.

[36] Knuth,D.E., *The Art of Computer Programming*, vol.2, *Seminumerical Algorithms*, Addison-Wesley Publications, Massachusetts,1981.

[37] Knuth,D.E., *The Art of Computer Programming*, vol.3, *Sorting and Searching*, Addison-Wesley Publications, Massachusetts, 1973.

[38] Kozen,D., "Semantics of Probabilistic Programs," Journal of Computer and Systems Science, vol.22, 1981, pp.328-350.

[39] Krizanc,D., "Routing and Merging in Parallel Models of Computation," Ph.D. Thesis, Aiken Computing Lab., Harvard University, Cambridge, Massachusetts, 1988.

[40] Krizanc,D., Rajasekaran,S., and Tsantilas,Th., "Optimal Routing Algorithms for Mesh-Connected Processor Arrays," Proc. Third International Aegean Workshop on Parallel Computation and VLSI Theory, 1988. Springer-Verlag Lecture Notes in Computer Science 318.

[41] Kruskal,C., Rudolph,L., and Snir,M., "Efficient Parallel Algorithms for Graph Problems," Proc. International Conference on Parallel Processing, 1986, pp.869-876.

[42] Kumar,M., and Hirschberg,D.S., "An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and its Application in Parallel Sorting Schemes," IEEE Transactions on Computers, vol. C32, 1983, pp.254-264.

[43] Kuzera,L., "Parallel Computation and Conflicts in Memory Access," Information Processing Letters 14(2), 1982, pp.93-96.

[44] Kunde,M., "Optimal Sorting on Multi-Dimensionally Mesh-Connected Computers," STACS 1987, Springer-Verlag Lecture Notes in Computer Science 247, 1987, pp.408-419.

[45] Ladner,R.E., and Fischer,M.J., "Parallel Prefix Computation," Journal of the ACM, 27(4), 1980, pp.831-838.

[46] Leighton,T., 'Tight Bounds on the Complexity of Parallel Sorting,' Proc. 16th Annual ACM Symposium on Theory Of Computing, 1984, pp.71-80.

[47] Lipton, R.J., and Tarjan, R.E., "Applications of a Planar Separator Theorem," SIAM Journal on Computing, vol.9, no.3, 1980, pp.615-627.

[48] Lipton, R.J., and Tarjan, R.E., "A Separator Theorem for Planar Graphs," SIAM Journal on Applied Mathematics, vol.36, no.2, 1979, pp.177-189.

[49] Luby,M., ‘A Simple Parallel Algorithm for the Maximal Independent Set Problem,’ Proc. 17th Annual ACM Symposium on Theory Of Computing, 1985, pp.1-10.

[50] Ma,Y., Sen,S., and Scherson,D., “The Distance Bound for Sorting on Mesh Connected Processor Arrays is Tight,” Proc. IEEE Symposium on Foundations of Computer Science, 1986, pp.255-263.

[51] MacLauren,M.D., “An Efficient Sorting Algorithm,” Journal of the ACM, vol.13, 1966, pp.404-411.

[52] Meggido, “Parallel Algorithms for Finding the Maximum and the Median in Almost Surely in Constant Time,” Preliminary Report, CS Department, Carnegie-Mellon University, Pittsburg, PA, October 1982.

[53] Mehlhorn,K., *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag Publications, 1984.

[54] Miller,G.L., Reif,J.H., ‘Parallel Tree Contraction and its Applications,’ Proc. IEEE Symposium on Foundations Of Computer Science, 1985, pp.478-489.

[55] Nassimi,D., and Sahni,S., “Data Broadcasting in SIMD Computers,” IEEE Transactions on Computers, vol.C30, no.2, 1981, pp.101-107.

[56] Pan, V., and Reif, J.H., “Fast and Efficient Solutions of Linear Systems,” Proc. 17th Annual Symposium on Theory Of Computing, 1985, pp.143-152.

[57] Pippenger,N., ‘Parallel Communication with Limited Buffers,’ Proc. IEEE Symposium on Foundations Of Computer Science, 1984, pp.127-136.

[58] Preparata, “New Parallel Sorting Schemes,” IEEE Transactions on Computers, vol. C27, no.7, 1978, pp.669-673.

[59] Rabin,M.O., ‘Probabilistic Algorithms,’ in: Traub,J.F., ed., *Algorithms and Complexity*, Academic Press, New York, 1976, pp.21-36.

[60] Ranade,A.G., “How to Emulate Shared Memory,” Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp.185-194.

[61] Rajasekaran,S., and Reif,J.H., ‘Derivation of Randomized Algorithms,’ Aiken Computing Lab. Technical Report TR-16-84, 1984.

[62] Rajasekaran,S., and Reif,J.H., “Optimal and Sub-Logarithmic Time Sorting Algorithms,’ Aiken Computing Lab. Technical Report, 1985. Also appeared as ‘An Optimal Parallel Algorithm for Integer Sorting,” in the Proc. of the IEEE Symposium on FOCS, 1985, pp.496-503.

[63] Rajasekaran,S., and Reif,J.H., “Randomized Parallel Computation,” Invited Paper, Foundations of Computer Theory Conference, Kazen, USSR, August 1987. Also presented in the 1987 Princeton Workshop on Algorithm, Architecture, and Technology Issues in Models of Concurrent Computations. Springer-Verlag Lecture Notes in Computer Science 278, pp.364-376.

[64] Rajasekaran,S., and Sen,S., “On Parallel Integer Sorting,” Technical Report CS-1987-38, Duke University, 1987.

[65] Rajasekaran,S., and Tsantilas,Th., “An Optimal Randomized Routing Algorithm for the Mesh and a Class of Efficient Mesh-Like Routing Networks,” Proc. 7th Conference on Foundations of Software Technology and Theoretical Computer Science, Pune, India, Dec.1987. Springer-Verlag Lecture Notes in Computer Science 287, pp.226-241.

[66] Reif, J.H., “An $n^{1+\epsilon}$ Processor $O(\log\log n)$ Time Probabilistic Sorting Algorithm,” SIAM Symposium on the Applications of Discrete Mathematics, Cambridge, Massachusetts, 1983, pp.27-29.

[67] Reif,J.H., ‘Depth First Search is Inherently Sequential,’ Information Processing Letters, vol.20, no.5, June 1985, pp.229-234.

[68] Reif,J.H., ‘On Synchronous Parallel Computations with Independent Probabilistic Choice,’ SIAM Journal on Computing, vol.13, no.1, 1984, pp.46-56.

[69] Reif,J.H., ‘Symmetric Complementation,’ Journal of the ACM, vol.31, no.2, 1984, pp.401-421.

[70] Reif,J.H., ‘Optimal Parallel Algorithms for Integer Sorting and Graph Connectivity,’ Technical Report TR-08-85, Aiken Computing Lab., Harvard University, Cambridge, Mass. 02138, 1985.

[71] Reif and Scherlis, “Deriving Efficient Graph Algorithms,” Logics of Programs Workshop, Pittsburg, PA, 1984. Also in Springer-Verlag Lecture Notes in Computer Science 164, pp.421-441.

[72] Reif,J.H., and Tygar,J.D., ‘Efficient Parallel Psuedo-Random Number Generation,’ Aiken Computing Lab. Technical Report, 1984.

[73] Reif,J.H., and Valiant,L.G., ‘A Logarithmic Time Sort for Linear Size Networks,’ Proc. 15th Annual Symposium on Theory of Computing, 1983, pp.10-16, also in JACM 1987.

[74] Reingold,E., Nievergelt,J., and Deo,N., *Combinatorial Algorithms: Theory and Practice*, Prentice Hall Publishing Company, New York, 1977.

[75] Reischuk,R., ‘A Fast Probabilistic Parallel Sorting Algorithm,’ Proc. IEEE Symposium on Foundations Of Computer Science, 1981, pp.212-219.

[76] Sado,K., and Igarishi,Y., “Some Parallel Sorts on a Mesh Connected Processor Array and their Time Efficiency,” Journal of Parallel and Distributed Computing, vol.3, no.3, 1986, pp.398-410.

[77] Scherlis, “Expression Procedures and Program Derivation,” Ph.D. Thesis, Stanford University, 1980.

[78] Schnorr,C.P., and Shamir,A., “An Optimal Sorting Algorithm for Mesh Connected Computers,” Proc. ACM Symposium on Theory of Computing, 1986, pp.255-263.

[79] Shiloach,Y., and Vishkin,U., “Finding the Maximum, Merging, and Sorting in a Parallel Computation Model,” Journal of Algorithms 2, 1981, pp.81-102.

[80] Shiloach,Y., and Vishkin,U., ‘An $O(\log n)$ Parallel Connectivity Algorithm,’ Journal of Algorithms, volume 3, 1983, pp.57-67.

[81] Smith,J.R., ‘Parallel Algorithms for Depth First Searches: I. Planar Graphs,’ International Conference on Parallel Processing, 1984, pp.299-301.

[82] Solovay,R., and Strassen,V., ‘A Fast Monte-Carlo Test for Primality,’ SIAM Journal of Computing, vol.6, 1977, pp.84-85.

[83] Tarzan,R.E., “Depth First Search and Linear Graph Algorithms,” SIAM Journal of Computing 1(2), 1972, pp.146-160.

[84] Tsantilas,Th., Ph.D. Thesis in Preparation, Aiken Computing Lab., Harvard Univ., Cambridge, Massachusetts, 1988.

[85] Thompson,C.D., “The VLSI Complexity of Sorting,” IEEE Transactions on Computers, vol.C32, no.12, 1983, pp.1171-1184.

[86] Thompson,C.D., and Kung,H.T., “Sorting on a Mesh Connected Parallel Computer,” Communications of the ACM, vol.20, no.4, 1977, pp.263-271.

[87] Tutte,W.T., ‘The Factorization of Linear Graphs,’ Journal of the London Mathematical Society, vol.22, 1947, pp.107-111.

[88] Upfal,E., ‘Efficient Schemes for Parallel Communication,’ Journal of the ACM, vol.31, no.3, 1984, pp.507-517.

[89] Valiant,L.G., ‘Parallelism in Comparison Problems,’ SIAM Journal of Computing, vol.14, 1985, pp.348-355.

[90] Valiant,L.G., ‘A Scheme for Fast Parallel Communication,’ SIAM Journal of Computing, vol.11, no.2, 1982, pp.350-361.

[91] Valiant,L.G., and Brebner,G.J., “Universal Schemes for Parallel Communication,” Proc. ACM Symposium on Theory of Computing, 1981, pp.263-277.

[92] Vishkin,U., “Randomized Speed-Ups in Parallel Computation,” Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp.230-239.

[93] Voorhis,V., ‘On Sorting Networks,’ Ph.D. Thesis, Stanford University CS department, 1971.

[94] Welsh,D.J.A., ‘Randomized Algorithms,’ Discrete Applied Mathematics, vol.5, 1983, pp.133-145.

[95] Wilks, *Mathematical Statistics*, John Wiley and Sons Publications, New York, 1962.