# General Purpose Parallel Computing

W F McColl *
*Programming Research Group*
*Oxford University*

**Abstract**

A major challenge for computer science in the 1990s is to determine the extent to which general purpose parallel computing can be achieved. The goal is to deliver both scalable parallel performance and architecture independent parallel software. (Work in the 1980s having shown that either of these alone can be achieved.) Success in this endeavour would permit the long overdue separation of software considerations in parallel computing, from those of hardware. This separation would, in turn, encourage the growth of a large and diverse parallel software industry, and provide a focus for future hardware developments.

In recent years a number of new routing and memory management techniques have been developed which permit the efficient implementation of a single shared address space on distributed memory architectures. We also now have a large set of efficient, practical shared memory parallel algorithms for important problems. In this paper we discuss some of the current issues involved in the development of systems which support fine grain concurrency in a single shared address space. The paper covers algorithmic, architectural, technological, and programming issues.

## 1. Introduction

The general purpose sequential computer is ubiquitous in contemporary society. A major challenge for computer science in the 1990s is to produce a credible framework which would allow parallel computing to develop in a way which would result in it eventually replacing sequential computing, not only in specific scientific and engineering applications, but in all areas.

> *Can parallel computing become the mainstream, rather than the interesting (and in some cases, important) sideshow that it is at present?*

At present, it is not commercially attractive to develop parallel software since the number of machines in use is small and current parallel software has to be extremely architecture dependent to achieve efficiency.

> *How can we bootstrap the parallel software industry?*

In this paper we will attempt to address these issues and suggest some solutions. The first question which needs to be addressed is the following:

*Author's address: Programming Research Group, Oxford University, 11 Keble Road, Oxford OX1 3QD, UK. This paper was written while the author was a Visiting Scientist at NEC Research Institute, Princeton, USA.

*What should be the main driving force in future parallel computing developments?*

There are three obvious candidates - hardware, software, or some kind of intermediate computational model. We will argue below that having an intermediate model as the main driving force offers the best hope for progress from our present position. First, let us consider the alternatives.

For most of the 1980s, low level hardware considerations have been the main driving force in parallel computing. Rapid progress in VLSI technology has permitted the development of a wide variety of distributed memory multicomputer architectures [29, 130, 232, 233, 234, 235, 236, 274]. These systems consist of a set of general purpose microprocessors connected by a sparse network, e.g. array, butterfly or hypercube. The relatively low speed and capacity of such networks forces the programmer to think in terms of a model in which one has multiple private address spaces connected in some complex way, e.g. in a hypercube structure, with explicit message passing by the programmer [124, 125, 135] for all non-local memory requests. The key to algorithmic efficiency in such systems is the careful exploitation of network locality. By minimising the number of nodes through which a message has to travel one can substantially improve efficiency. Despite the programming difficulties inherent in this approach, a large amount of scientific and technical applications software has been developed for such systems. In positive terms, this work has demonstrated conclusively that for many important applications, scalable parallel performance can be achieved in massively parallel systems [112, 113], despite the reservations expressed by Amdahl [20]. However, in this message passing approach, most of the effort in software development tends to be devoted to the various low level process mapping activities which need to be performed to achieve efficiency. Besides being extremely tedious in many cases, this usually produces software which cannot easily be adapted to another architecture. In a world of rapidly changing parallel architectures, this architecture-dependence has proved to be a major weakness, and it has inhibited the growth of the field beyond the area of scientific research.

An alternative approach, which has been extensively pursued by computer science researchers in the last decade, is to make software the driving force. A variety of approaches of this kind have been investigated. They differ in terms of the type of programming language considered, e.g. functional [39, 126, 127] , single assignment, logic, mostly functional, and in the computational model which they adopt, e.g. graph reduction, rewriting, dataflow. However, they share a number of similarities, particularly in comparison to the framework proposed in this paper. One example of this approach is where one starts by noting that a high level functional language [39, 126, 127] (if properly used) can often expose a large amount of implicit parallelism in a computational problem. The decision to work with a functional language, for reasons of architecture-independence, naturally leads to a decision to adopt, say, graph reduction as the model of parallel computation. The technological (hardware) goal is then to develop a scalable massively parallel architecture for graph reduction [210, 211]. This "software first" approach has a great deal of merit given that hardware is changing rapidly and that the cost and time required to produce software makes architecture-independence in software a major goal. Unfortunately, however, the amount of progress which has been made on the development of efficient parallel architectures for graph reduction, dataflow or rewriting has not been particularly impressive so far, despite much effort. The experiences of the last decade suggest that, in the pursuit of efficiency, it is often necessary to compromise some of the elegance and simplicity of such approaches.

To a large extent, this has already happened in dataflow implementations of functional languages, e.g. the implementation of Id [199] on Monsoon [206]. Modern dataflow architectures [128, 129, 200, 201, 206] are in many important respects quite close to those described in this paper. For example, they must achieve latency tolerance through multithreading since, in the dataflow model, memory accesses are *split transactions* [201]. (In the dataflow model, a read may be requested before the value is computed.)

The third alternative is to have some model of parallel computation as the driving force. Around 1944, von Neumann produced a proposal [49, 269] for a general purpose stored-program sequential computer which captured the fundamental principles of Turing's work [249] in a practical design. The design, which has come to be known as the "von Neumann computer", has served as the basic model for almost all sequential computers produced from the late 1940s to the present time. As noted in [120], "The paper by Burks, Goldstine and von Neumann ([49]) was incredible for the period. Reading it today, one would never guess this landmark paper was written more than 40 years ago, as most of the architectural concepts seen in modern computers are described there." For an account of the principles of modern general purpose sequential (i.e. von Neumann) computer design, see e.g. [119, 120, 208]. For sequential computation, the stability of the von Neumann model has permitted the development, over the last three decades, of a variety of high level languages and compilers. These have, in turn, encouraged the development of a large and diverse software industry producing portable applications software for the wide range of von Neumann machines available, from personal computers to large mainframes. The stability of the underlying model has also allowed the development of a robust complexity theory for sequential computation, and a set of algorithm design and software development techniques of wide applicability. General purpose sequential computing based on the von Neumann model has developed vigorously over the last four decades. The widespread adoption of the model has not proved to be a harmfully constraining influence, in fact, it has been quite the reverse. A variety of hardware approaches have flourished within the framework provided by the model. The stability it has provided has been invaluable for the development of the software industry.

No single model of parallel computation has yet come to dominate developments in parallel computing in the way the von Neumann model has dominated sequential computing [84, 258, 259].

> *Can we identify a robust model of parallel computation which offers the prospect of achieving the twin goals of general purpose parallel computing - scalable parallel performance and architecture-independent parallel software?*

Success in this endeavour would permit the long overdue separation of software and hardware considerations in parallel computing. This separation would, in turn, encourage the growth of a large and diverse parallel software industry, and provide a focus for future hardware developments.

The achievement of these goals would have profound consequences for the future development of both the computing industry and the academic subject of computer science. Given this fact, one might suspect that this issue would be central to much of the current research in parallel computing. However, at present, relatively little work is being done with these goals directly in mind. Much of the practical work in massively parallel computing today is concerned with the development of scientific applications software, without particular regard for the development of a credible strategy which would permit portability of that software as new architectures appear.

The current situation in parallel computing is remarkably chaotic when compared with that of sequential computing. With no agreed model to provide a focus for technological innovation, parallel hardware suppliers continue to develop, and attempt to market, systems with widely differing characteristics. Those people with the unenviable task of choosing a parallel system for their organisation are faced with the prospect of investing substantial resources in the purchase of such a machine, and in the development of software for it, only to find that the software quite quickly becomes obsolete.

At the present time, the MFLOP performance of the processors used in parallel systems is increasing rapidly. Unfortunately, this is not being matched by corresponding increases in communications performance. This rising imbalance is likely to further increase the difficulty of achieving architecture-independence in software. An important general message of the results in this paper is that architecture-independence is more likely to be achieved in those parallel systems which invest more substantially in communications performance than in processor performance. It is striking that few of the commercial parallel systems being produced today seem to reflect this basic idea.

The current chaos in parallel computing has led many to conclude that the answer to the above question, on the prospects for agreement on a model, is no. Advocates of "heterogeneous parallel computing" [163] take as their starting point the idea that no convergence on a model is likely to take place. They argue that a wide variety of designs which are to some extent "special purpose" will continue to be produced and marketed, and that the primary function of parallel computing should be to develop languages and communications networks for the coordination of these ensembles of devices. It is again striking that many in computing have already accepted the inevitability of this rather pessimistic scenario, especially as no serious theoretical impediments to the achievement of the goals of general purpose parallel computing have yet been identified, despite much effort to find them. One can contrast this with the situation in complexity theory where the ideas of NP-completeness have demonstrated in a precise way that many desirable goals in terms of algorithmic performance, for problems in AI, scheduling, optimisation etc. are unlikely to be achievable and that we must, in some way, limit our expectations. There is no compelling evidence that general purpose parallel computing, as described above, cannot be achieved. We can be reasonably confident that, as future hardware developments alone fail to significantly increase the market for parallel systems, the manufacturers of those systems will see it as in their interests to seek convergence on a model, rather than to seek to avoid it. A major goal for computer science today is to develop the ideas and techniques which will provide the required solutions when that change in thinking comes about.

In this paper we will describe one possible way forward for parallel computing, based on the bulk synchronous parallel (BSP) model of computation [258]. Although this approach has many strengths, we would not want to argue that it is the only viable approach. Two alternatives, which merit serious consideration, are the actor model [9, 10] and the dataflow model [128, 129, 199, 200, 206]. The most fundamental difference between these two approaches and the BSP model is that they both have at their core the idea of local (usually pairwise) synchronisation events, whereas the BSP model, as well as various PRAM [80, 89, 149, 224, 259, 266] and data parallel models [43, 123], have the idea of global barrier synchronisation as the basic mechanism. Another significant difference is that in the BSP, PRAM and data parallel approaches there is usually tight control of ordering and scheduling by the programmer. In contrast, a major attraction of the dataflow approach is that the programmer is freed from consideration of such issues. Although we have stressed the differences between these various approaches, there is reason to believe that, at the architectural level, the BSP, PRAM, actor and dataflow models will require a number of similar mechanisms for efficient implementation, in particular, high performance global communications, uniform memory access, and multithreading to hide network latencies.

It is perhaps not unreasonable to summarise the current situation with respect to these various approaches as follows. Work on the actor and dataflow models is much more highly developed in the areas of programming languages and methodologies than it is in the area of algorithm design, analysis and complexity. In contrast, for the BSP and PRAM models we have a highly developed set of techniques for the design and analysis of algorithms, but we do not yet have an established framework for the programming of such systems.

## 2. Idealised Parallel Computing

Various idealised shared memory models of parallel computation have been used in the study of parallel algorithms and their complexity. Three such models are the PRAM, the circuit, and the comparison network. In this section we describe these models, and give a number of simple examples of efficient shared memory parallel algorithms which can be implemented on them. Most of the circuits and comparison networks described can be translated into PRAM algorithms in a straightforward manner. We also discuss various ways in which the efficiency of shared memory parallel algorithms can be measured. The class $\mathcal{NC}$ has, over the past decade, provided a very simple and robust framework for the classification of problems in $\mathcal{P}$, in terms of their parallel time complexity on a PRAM. A large number of important problems have been shown to lie in $\mathcal{NC}$, i.e. to be solvable on a PRAM in polylogarithmic time using a polynomial number of processors. Other problems have been shown to be $\mathcal{P}$-complete, i.e. to have no $\mathcal{NC}$ algorithm unless $\mathcal{P} = \mathcal{NC}$. The class $\mathcal{NC}$ and the notion of $\mathcal{P}$-completeness have allowed major advances to be made in our theoretical understanding of shared memory parallel algorithms and their complexity. However, as we now move forward to the point of developing parallel architectures based on the PRAM model, we require a rather more refined complexity theory which takes account of the amount of work done by a parallel algorithm. We describe an approach due to Kruskal, Rudolph and Snir [161] which captures this idea in a convenient way. At the end of the section, we discuss the communication complexity of PRAM algorithms. We describe various results showing tradeoffs between the time required for a parallel computation and the total number of messages which must be sent. Such results may provide a theoretical basis for the future development of software tools which efficiently schedule shared memory parallel algorithms for implementation on distributed memory architectures.

### 2.1. The PRAM

A *parallel random access machine (PRAM)* [80, 83, 149, 266, 276] consists of a collection of processors which compute synchronously in parallel and which communicate with a common global random access memory. In one time step, each processor can do (any subset of) the following - read two values from the common memory, perform a simple two-argument operation, write a value back to the common memory. There is no explicit communication between processors. Processors can only communicate by writing to, and reading from, the common memory. The processors have no local memory other than a small fixed number of registers which they use to temporarily store the argument and result values. In a *Concurrent Read Concurrent Write (CRCW) PRAM*, any number of processors can read from, or write to, a given memory cell in a single time step. In a *Concurrent Read Exclusive Write (CREW) PRAM*, at most one processor can write to a given memory cell at any one time. In the most restricted model, the *Exclusive Read Exclusive Write (EREW) PRAM*, no concurrency is permitted either in reading or in writing. The CRCW PRAM model has a large number of variants which differ in the convention they adopt for the effect of concurrent writing. Three simple examples of such conventions are: two or more processors can write so long as they write the same value, one of the processors attempting to write will succeed but the choice of which one will succeed will be made nondeterministically, the lowest numbered processor will succeed (assuming some appropriate numbering.) In other CRCW models [221] one might have the possibility of concurrent writing in which the memory location is updated to the sum of the written values, or to the minimum of the written values.

As a simple example of a CREW PRAM computation, consider the problem of computing $ab+ac+bd+cd$ from inputs $a, b, c, d$. Let $p_i t_j$ denote the computation performed by processor $i$ at time step $j$. Then we have

$$\begin{aligned} p_1 t_1 &: \; b + c \Rightarrow x \\ p_1 t_2 &: \; a * x \Rightarrow y \\ p_2 t_2 &: \; x * d \Rightarrow z \\ p_1 t_3 &: \; y + z \Rightarrow \text{result} \end{aligned}$$

The complexity of a PRAM algorithm is given in terms of the number of time steps and the maximum number of processors required in any one of those time steps. The above example requires three time steps and two processors.

From the perspective of this paper, the most important characteristic of the PRAM model is that it is a 1-level memory (or shared memory) model, i.e. all of the memory locations are uniformly far away from all of the processors, the processors have no local memory and there is no kind of memory hierarchy based on ideas of network locality. These simplifying properties of the PRAM model have made it extremely attractive as a robust model for the design and analysis of algorithms.

## 2.2. Circuits

A *circuit* [77, 270, 272] is a directed acyclic graph with $n$ input nodes (in-degree 0) corresponding to the $n$ inputs to the problem, and a number of *gates* (in-degree 2) corresponding to two-argument functions. In a Boolean circuit, the gates are labelled with one of the binary Boolean functions $NAND, \wedge, NOR, \vee, \rightarrow, \oplus$ etc. In a typical arithmetic circuit, the input nodes are labelled with some value from $\mathbb{Q}$, the set of rational numbers, and the gates are labelled with some operation from the set $\{+, -, *, /\}$. The size of a circuit is the number of gates

Let $g_i$ denote the function computed by gate $i$. Then we have the following arithmetic circuit for $ab + ac + bd + cd$.

$$g_1 \;=\; b + c$$

$$\begin{aligned} g_2 &= a * g_1 \\ g_3 &= g_1 * d \\ g_4 &= g_2 + g_3 \end{aligned}$$

An example of a Boolean circuit is the following, which computes the two binary digits $<d_1, d_0>$ of $x_1 + x_2 + x_3$.

$$\begin{aligned} g_1 &= x_1 \wedge x_2 \\ g_2 &= x_1 \oplus x_2 \\ g_3 &= g_2 \wedge x_3 \\ g_4 &= g_2 \oplus x_3 \ (= d_0) \\ g_5 &= g_1 \vee g_3 \ (= d_1) \end{aligned}$$

The parallel complexity of a circuit is the depth of the circuit, i.e. the maximum number of gates on any directed path. The parallel complexity of both of the above examples is three.

## 2.3. Comparison Networks

It is well known that $\Theta(n\ log\ n)$ binary comparisons are necessary and sufficient to sort $n$ elements drawn from an arbitrary totally ordered set. The lower bound follows from a simple information theoretic argument. The matching upper bound can be obtained, e.g. by a simple recursive mergesort algorithm. A convenient model for the investigation of the parallel complexity of comparison problems such as sorting, merging and selection is the comparison network [32, 66, 155]. Comparison networks have the attractive property that they are oblivious (as are circuits). An *oblivious algorithm* is one in which the sequence of operations performed is independent of the input data.

A *comparison element* is a two-input two-output device which computes the minimum $min(x, y)$ and the maximum $max(x, y)$ of its inputs $x, y$. In an *n-line comparison network*, the $n$ inputs $<x_1, x_2, \ldots, x_n>$ are presented on the $n$ lines and at each successive level of the network at most $n/2$ disjoint pairs of lines are put through comparison elements. After each level, the $n$ lines carry the inputs in some permuted order. The *size* of a comparison network is the number of elements.

If we let $l_i$ denote level $i$ and $(j, k)$ denote a comparison element connecting lines $j, k$, then the following is a comparison network of size five which sorts four elements

$$\begin{aligned} l_1 &: \{(1,2), (3,4)\} \\ l_2 &: \{(1,3), (2,4)\} \\ l_3 &: \{(2,3)\} \end{aligned}$$

i.e. if we present the inputs $<x_1, x_2, x_3, x_4>$ on the four lines then after these three levels of comparison elements the values will appear on the lines in sorted order.

The parallel complexity of a comparison network is simply the *depth* of the network, i.e. the number of levels.

## 2.4. Addition

Let $ADD_n(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$ where $x_i \in \mathbb{Q}$. A circuit of depth $\lceil log_2 n \rceil$ for $ADD_n$ can easily be obtained by constructing a balanced binary tree of $+$-gates, with $n$ leaves corresponding to the arguments $x_1, \ldots, x_n$. This corresponds to an $n/2$ processor PRAM algorithm with complexity $\lceil log_2 n \rceil$ The optimality of this construction, in terms of depth, follows from the functional dependency of $ADD_n$ on each of its $n$ arguments. If we now define $OR_n(x_1, \ldots, x_n) = \bigvee_{i=1}^{n} x_i$ where $x_i \in \{0, 1\}$, then we have a very similar problem to that of computing $ADD_n$. We can easily obtain a PRAM algorithm of complexity $\lceil log_2 n \rceil$ and a Boolean circuit of depth $\lceil log_2 n \rceil$ for $OR_n$. That this circuit depth is optimal follows from functional dependency. However, as Cook and Dwork [64] observed, there is rather more to the question of the PRAM complexity of $OR_n$. If we allow concurrent write, then $OR_n$ can be computed in one parallel step in an obvious way; processor $i$ reads $x_i$ from memory location $i$ and if $x_i = 1$ it writes a 1 into location 0. Cook and Dwork [64] show that even on an EREW PRAM, $OR_n$ can be computed in less than $\lceil log_2 n \rceil$ steps. They derive an upper bound of $(0 \cdot 72) log_2 n$ on the number of steps required. However, they also show that a lower bound of $\Omega(log_2 n)$ holds, and thus only a constant factor improvement is possible.

## 2.5. Polynomial Evaluation

Let $P_n(a_0, a_1, \ldots, a_n, x) = \sum_{i=0}^{n} a_i x^i$ where $a_i, x \in \mathbb{Q}$. The standard sequential algorithm for polynomial evaluation is *Horner's Rule* where to calculate $P_n$ we successively compute

$$\begin{aligned} p_n &= a_n \\ p_i &= (p_{i+1} * x) + a_i \quad for \;\; i = n-1, n-2, \ldots, 1, 0. \end{aligned}$$

Then $P_n = p_0$. The sequential complexity of polynomial evaluation has been studied for many years. It is known that $2n$ arithmetic operations are required to evaluate a general polynomial of degree $n$, given by its coefficients [46]. Thus, in terms of sequential complexity, Horner's Rule is optimal. However, it is very unsuitable for parallel computation since at every step in the computation the immediately preceding subresult is required. If instead, we evaluate each term $a_i x^i$ of the polynomial independently, in parallel, using a balanced binary tree of $*$-gates, and we then sum the values of the terms using a balanced binary tree of $+$-gates then we have a circuit of depth $2\lceil log_2(n+1) \rceil$. This circuit is *exponentially better*, in terms of depth, than a circuit based on Horner's Rule, although the number of gates (sequential complexity) is now $O(n^2)$ rather than $O(n)$. The $2\lceil log_2(n+1) \rceil$ upper bound can be further improved to $log_2 n + O(\sqrt{log_2 n})$ by using a simple recursive parallel algorithm due to Munro and Paterson [198] which splits the polynomial into consecutive blocks of terms and factors out the appropriate power of $x$. Kosaraju [157] has shown that the algorithm of Munro and Paterson is optimal, in terms of circuit depth, for polynomial evaluation.

## 2.6. Prefix Sums

Let $x_1, x_2, \ldots, x_n$ be a set of values and $\circ$ be an associative operation on that set. The *prefix sums problem* is to compute $p_i = x_1 \circ x_2 \circ \cdots \circ x_i$ for all $1 \leq i \leq n$. The straightforward method yields a circuit of size $n-1$ but its depth is also $n-1$. By computing each $p_i$ independently we can obtain a circuit of size $O(n^2)$ and depth $\lceil log_2 n \rceil$. An important result of Ladner and Fischer [165] shows that the prefix sums problem can be computed by a circuit of size $O(n)$ and depth $O(log\ n)$. This construction can be applied to produce small fast parallel circuits for a variety of important problems, including $n$-bit addition, $n$-bit multiplication, and Boolean sorting. It can also be used for the efficient parallel simulation of finite state automata. We will describe the application of parallel prefix computation to the problem of $n$-bit addition.

A binary number $<a_{n-1}, a_{n-2}, \ldots, a_0> \in \{0,1\}^n$ represents the value $\sum_{i=0}^{n-1} a_i * 2^i$. Given two $n$-bit binary numbers $X = <x_{n-1}, x_{n-2}, \ldots, x_0>$ and $Y = <y_{n-1}, y_{n-2}, \ldots, y_0>$, the *n-bit addition problem* is to compute the $(n+1)$-bit representation $Z = <z_n, z_{n-1}, \ldots, z_0>$ of $X + Y$.

In the normal "school method" we first compute $z_0 = x_0 \oplus y_0$ and the initial carry bit $c_0 = x_0 \wedge y_0$. We then use $n-1$ full adders to compute $z_i, c_i$ from $x_i, y_i$ and $c_{i-1}$. Finally, we let $z_n = c_{n-1}$. This method yields a circuit of size $O(n)$ and depth $O(n)$. The prefix method consists of three stages:

**Stage 1:** Compute $u_j = x_j \wedge y_j$, $v_j = x_j \oplus y_j$ for all $0 \leq j < n$.

**Stage 2:** Compute the carry bits $c_j$ for all $0 \leq j < n$.

**Stage 3:** Compute the outputs. $z_0 = v_0$, $z_j = v_j \oplus c_{j-1}$ for all $1 \leq j < n$, $z_n = c_{n-1}$.

Stages 1 and 3 can both be carried out in linear size and constant depth, therefore we need only consider Stage 2. Let $A_{(u,v)}(c) = u \vee (v \wedge c)$ for $c \in \{0,1\}$. Then we have

$$c_i = A_{(u_i,v_i)} \circ A_{(u_{i-1},v_{i-1})} \circ \cdots \circ A_{(u_0,v_0)}(0)$$

where $\circ$ denotes function composition. Since $(u_j, v_j) \in \{(0,0),(0,1),(1,0)\}$ and

$$\begin{aligned} A_{(0,0)} \circ A_{(u,v)} &= A_{(0,0)} \\ A_{(0,1)} \circ A_{(u,v)} &= A_{(u,v)} \\ A_{(1,0)} \circ A_{(u,v)} &= A_{(1,0)} \end{aligned}$$

it follows that the operation $\circ$ on sets of functions $A_{(u_j,v_j)}$ is associative. Therefore, we can use a prefix circuit to compute all the carry bits $c_i$. We need only design a circuit for the operation $\circ$. Let $A_{(u,v)} = A_{(u_2,v_2)} \circ A_{(u_1,v_1)}$. Then $(u,v) = (u_2 \vee (u_1 \wedge v_2), v_1 \wedge v_2)$ and therefore we can construct a subcircuit of size three and depth two for the operation $\circ$. We have shown that $n$-bit addition can be realised by a Boolean circuit of size $O(n)$ and depth $O(log\ n)$.

In functional programming [39], the second-order function *scan* corresponds to the prefix sums computation. The above mentioned results, and the work of Blelloch [41, 42, 43] and others, have shown it to be a parallel primitive of extremely wide applicability.

## 2.7. Matrix Multiplication

Let $A, B$ be two $n \times n$ matrices of rational numbers. Then the product of $A, B$ is an $n \times n$ matrix $C$, where $c_{i,j} = \sum_{k=1}^{n} a_{i,k} * b_{k,j}$. The exact determination of the sequential complexity of matrix multiplication is a major open problem in the field of computational complexity [62, 202, 247]. At the present time, the best known algorithm (asymptotically, as $n \rightarrow \infty$) requires only $O(n^{2.376})$ arithmetic operations [65] as opposed to the standard $O(n^3)$ which follows from the definition. No lower bound larger than the trivial $\Omega(n^2)$ is known.

In contrast, determining the shared memory parallel complexity of matrix multiplication is trivial. We can evaluate each $c_{i,j}$ term independently, in parallel, by a balanced binary tree of depth $\lceil log_2 n\rceil + 1$. Functional dependency shows this bound for $c_{i,j}$ to be optimal and so we have an optimal time bound for parallel matrix multiplication on the idealised PRAM and circuit models [number of processors $= O(n^3)$]. The simplicity of this solution is entirely attributable to the fact that in the PRAM and circuit models, the complexity and cost of communication is completely ignored. Those who have developed and implemented algorithms for distributed memory architectures over the last decade know, of course, that on such architectures the issue of managing communication is the dominant one.

## 2.8. Linear Recurrences

The parallel evaluation of recurrences was discussed in the mid 1960s by Karp, Miller and Winograd [148]. Let us first consider the computation of the very simple recurrence which defines Fibonacci numbers. The $m^{th}$ Fibonacci number $f_m$ is given by the second order linear recurrence

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_m &= f_{m-1} + f_{m-2} \quad for \;\; m \geq 2 \end{aligned}$$

This definition can be directly translated into an arithmetic circuit with $m-1$ gates (and depth $m-1$) which successively computes $f_2, f_3, \ldots, f_m$. As in the case of Horner's Rule we have a circuit with no direct parallel speedup. If instead, we use the unconventional definition

$$(f_{m-1}\; f_m) \;=\; (f_0\; f_1)\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{m-1}$$

then we see immediately that $f_m$ can be calculated by an arithmetic circuit of *size and depth* $O(log_2 m)$ if we compute the matrix power efficiently by repeated squaring.

The above result for Fibonacci numbers is a special case of the following more general result by Greenberg et al. [108] on the parallel evaluation of $k^{th}$ order linear recurrences. If we have $F = (f_0\; f_1\; \ldots\; f_{k-1})$ and $f_m = \sum_{j=1}^{k} a_{k-j} * f_{m-j}$ for $m \geq k$, then $(f_{m-k+1}\; \ldots\; f_m) = F * M^{m-k+1}$ where $M$ is the $k \times k$ matrix

$$\left(\begin{array}{c|c} 0 \;\cdots\; 0 & a_0 \\ \hline & a_1 \\ I & \vdots \\ & a_{k-1} \end{array}\right)$$

and therefore the parallel complexity of computing $f_m$ is at most $O(log_2 k \cdot log_2(m-k))$.

For a practical application of this result we consider the problem of solving linear systems. Let $B$ be an $n \times n$ non-singular, lower triangular matrix, and $\underline{c}$ be an $n$-element vector. In solving the linear system $B\underline{x} = \underline{c}$ by 'back substitution' we use the recurrence $x_i = (c_i - \sum_{j=1}^{i-1} b_{i,j} * x_j)/b_{i,i}$ for $1 \leq i \leq n$. If we let $x_i = 0$ for $i < 1$ then we can rewrite this recurrence in the form $(x_i \; x_{i-1} \; \ldots \; x_{i-n+1} \; 1) \;=\; (x_{i-1} \; x_{i-2} \; \ldots \; x_{i-n} \; 1) * M_i$ where

$$M_i \;=\; \left(\begin{array}{c|ccc|c} -\frac{b_{i,i-1}}{b_{i,i}} & & & & \\ \vdots & & & & \\ -\frac{b_{i,1}}{b_{i,i}} & & I & & 0 \\ 0 & & & & \\ \vdots & & & & \\ 0 & & & & \\ \hline 0 & 0 & \cdots\cdots\cdots & 0 & 0 \\ \frac{c_i}{b_{i,i}} & 0 & \cdots\cdots\cdots & 0 & 1 \end{array}\right)$$

Therefore we can design an arithmetic circuit of depth $O(log^2 n)$ which solves $B\underline{x} = \underline{c}$ to obtain $\underline{x}$. This circuit corresponds to a PRAM algorithm with parallel time $O(log^2 n)$ and number of processors $O(n^4)$. In contrast, a direct PRAM implementation of back substitution would have parallel time $O(n)$, but would require only $O(n)$ processors.

## 2.9. Merging

Let $<x_1, x_2, \ldots, x_m>$ and $<y_1, y_2, \ldots, y_n>$ be two sorted sequences. The *merging problem* is to produce a single sorted sequence consisting of the $m+n$ elements. This can, of course, be performed by a simple sequential algorithm which uses at most $m+n-1$ binary comparisons. We will describe two efficient parallel comparison networks for merging, both due to Batcher [32, 155]. The two techniques are known as odd-even merging and bitonic sorting.

In odd-even merging, we merge the "odd sequences" $<x_1, x_3, x_5, \ldots>$ and $<y_1, y_3, y_5, \ldots>$, obtaining $<v_1, v_2, v_3, \ldots>$; and merge the "even sequences" $<x_2, x_4, x_6, \ldots>$ and $<y_2, y_4, y_6, \ldots>$, obtaining $<w_1, w_2, w_3, \ldots>$. (These two merges are performed in parallel). Finally, we apply comparison elements to the pairs $(w_1, v_2), (w_2, v_3), (w_3, v_4), \ldots$ to complete the merging.

This recursive method yields the following upper bounds:

$$\begin{aligned} size(n) &\leq 2 * size(n/2) + O(n) \\ &= O(n \; log \; n) \end{aligned}$$

$$\begin{aligned} depth(n) &\leq depth(n/2) + 1 \\ &\leq \lceil log_2 n \rceil \end{aligned}$$

A sequence $<z_1, z_2, \ldots, z_p>$ is *bitonic* if and only if $z_1 \geq \cdots \geq z_k \leq \cdots \leq z_p$ for some $1 \leq k \leq p$. An *n-line bitonic sorter* is a comparison network which will sort any bitonic sequence of length $n$. Merging can be performed by sorting the bitonic sequence $<x_m, x_{m-1}, \ldots, x_1, y_1, y_2, \ldots, y_n>$. Noting that any subsequence of a bitonic sequence is bitonic, it follows that we can construct an $n$-line bitonic sorter by first sorting the two bitonic subsequences $<z_1, z_3, \ldots>$ and $<z_2, z_4, \ldots>$ in parallel, and then applying comparison elements to the pairs $(z_1, z_2), (z_3, z_4), \ldots$ to complete the sort. This alternative method yields essentially the same upper bounds on size and depth as odd-even merging. It does, however, have some advantages in terms of simplicity of description. A bitonic sorter with $2^n$ lines numbered $0, 1, 2, \ldots, 2^n - 1$ can be defined (nonrecursively) in the following way:

Lines $i, j$ are compared on level $k$ if and only if $i, j$ differ only in their $k^{th}$ most significant bit.

It is quite easy to prove that both the size and depth of these merging networks are optimal to within a constant factor [155]. This shows that, for the problem of merging, comparison networks are much less powerful than general (adaptive) algorithms, at least when one compares the total number of comparisons performed. As we shall see in the next section, this does not apply in the case of the related problem of sorting.

## 2.10. Sorting

An n-line sorting network can be constructed recursively using either of the efficient merging networks described above [32, 155]. To sort the set $x_1, x_2, \ldots, x_n$ we first (recursively) sort the two subsets $x_1, x_2, \ldots, x_{n/2}$ and $x_{(n/2)+1}, x_{(n/2)+2}, \ldots, x_n$ in parallel. The two sorted sequences can then be combined using one of the above merging networks of size $O(n\ log\ n)$ and depth $O(log\ n)$. This yields the following upper bounds for sorting $n$ elements by a comparison network.

$$\begin{aligned} size(n) &\leq 2 * size(n/2) + O(n\ log\ n) \\ &= O(n\ log^2 n) \end{aligned}$$

$$\begin{aligned} depth(n) &\leq depth(n/2) + log\ n \\ &= O(log^2 n) \end{aligned}$$

From 1968 until 1983 this was the best known oblivious sorting algorithm. In 1983, Ajtai, Komlós and Szemerédi [11] succeeded in producing a remarkable $n$-line sorting network of size $O(n\ log\ n)$ and depth $O(log\ n)$, both of which are of course asymptotically optimal. A more efficient version, which improves the constant factors involved, has since been produced by Paterson [207]. Although the constant factors are still too large to make the networks competitive with those obtained from, say, odd-even merging, this is a result of major theoretical significance as it shows that for the important problem of sorting, adaptive algorithms are not (asymptotically) more powerful than oblivious ones.

## 2.11. Selection

The $t^{th}$ *selection problem* is to determine the $t$ largest elements in a set of size $n$. In this section we describe the design of some fast parallel comparison networks for this problem. When $t \approx n/2$, we cannot do significantly better than using an $O(n\ log\ n)$ size, $O(log\ n)$ depth sorting network [11] since any comparison network which determines the median of $n$ elements must have size $\Omega(n\ log\ n)$ [155]. We shall be concerned with the case where $t$ is fixed as $n \rightarrow \infty$.

The following construction is due to Yao [277]. To produce a $t^{th}$ selection network of small depth we use a construction called a $(t, n)$-eliminator. Let $f(t, n)$ be a function which satisfies $f(t, n) \geq t$. A $(t, n)$-*eliminator* is a comparison network with the following property: Of the $n$ output lines there are $f(t, n)$ designated lines among which the largest $t$ elements are found. We now show how to construct a family $E(t, n)$ of $(t, n)$-eliminators.

First, we consider the case where $t = 1$. $E(1, 2)$ is simply a comparison element. An $n$-line $E(1, n)$ is recursively defined as follows: The first level consists of the elements $(1, n), (2, n-$

$1), (3, n-2), \ldots$. The rest of the network is simply an $E(1, n/2)$ network on lines $(n/2)+1, (n/2)+2, \ldots, n$. A simple analysis shows that

$$\begin{aligned} depth(E(1,n)) &= \lceil log_2 n \rceil \\ f(1,n) &= 1 \end{aligned}$$

For $t > 1$, an $n$-line $E(t,n)$ is recursively defined as follows: The first level again consists of the elements $(1,n), (2,n-1), (3,n-2), \ldots$. The rest of the network consists of an $E(\lfloor t/2 \rfloor, \lfloor n/2 \rfloor)$ network on lines $1, 2, \ldots, \lfloor n/2 \rfloor$ and an $E(t, \lceil n/2 \rceil)$ network on the remaining lines. For $t = 2$, we obtain

$$\begin{aligned} depth(E(t,n)) &\leq max\{depth(E(t,n/2)), (log\ n) - 1\} + 1 \\ &\leq \lceil log_2 n \rceil \\ f(2,n) &\leq f(2,n/2) + 1 \\ &= O(log\ n) \end{aligned}$$

and, in general, for any fixed $t \geq 2$,

$$\begin{aligned} depth(E(t,n)) &\leq \lceil log_2 n \rceil \\ f(t,n) &= O((log\ n)^{\lfloor log_2 t \rfloor}) \end{aligned}$$

To obtain a fast parallel comparison network for the $t^{th}$ selection problem we simply compose appropriately sized eliminator networks. Applying $E(t,n)$ to the $n$ input lines gives a set of $n_1 = f(t,n) \approx (log\ n)^{log\ t}$ lines on which the $t$ largest elements are now known to lie. We can then apply an $E(t, n_1)$ to reduce the number of lines to $n_2 = f(t, n_1)$, and an $E(t, n_2)$ to further reduce it to $n_3 = f(t, n_2)$. The depth of the network at this stage of the construction is $log\ n + log\ n_1 + log\ n_2 = log\ n + log\ t\ loglog\ n + O(logloglog\ n)$ and the number of lines has been reduced to $O((logloglog\ n)^{log\ t})$. This is sufficiently small that we can now use, for example, a sorting network on those lines to complete the computation of the $t$ largest elements. This gives a $t^{th}$ selection network of total depth $log\ n\ +\ log\ t\ loglog\ n\ +\ O(logloglog\ n)$.

## 2.12. Algebraic Path Problem

A *closed semiring* is an algebraic structure $(S, \oplus, \otimes, I_\oplus, I_\otimes)$ with the following properties:

$\oplus$ is a commutative monoid ($\oplus$ satisfies the closure, associative, commutative properties, and has identity element $I_\oplus$).

$\otimes$ is a monoid ($\otimes$ satisfies the closure, associative properties, and has identity element $I_\otimes$).

$\oplus$ is idempotent.

$\otimes$ is right and left distributive over $\oplus$.

For all $s \in S$, $s \otimes I_\oplus = I_\oplus$.

Let $G = (V, A)$ be a directed graph on $\mid V \mid = n$ vertices, in which each $<i, j> \in A$ has an associated weight $s_{ij} \in S$. Define an $n \times n$ matrix M of weights $m_{ij}$ corresponding to the arcs of $G$: $m_{ij} = s_{ij}$ if $<i, j> \in A$, $I_\oplus$ otherwise. The *Algebraic Path Problem (APP)* is to compute $M^* = \bigoplus_{k=0}^{\infty} M^k$ where matrix product is defined in terms of the two operations $\oplus$ and $\otimes$. ($M^0$ is the identity matrix with diagonal elements $I_\otimes$). $M^*_{ij}$ gives the "sum" of the weights of all directed paths from $i$ to $j$ where the weight of a path is the "product" of the weights of the arcs.

The APP is a problem of major importance in a wide variety of areas and has been extensively studied in recent years. Some examples of instances of the APP are the following:

| Problem | $S$ | $\oplus$ | $\otimes$ | $I_\oplus$ | $I_\otimes$ |
|---|---|---|---|---|---|
| Connectivity (trans. closure) [271] | $\{false, true\}$ | *or* | *and* | *false* | *true* |
| Generation of regular language | $\{words\}$ | $\cup$ | $\cdot$ | $\emptyset$ | *empty word* |
| Max. capacity path | $\mathbb{R}^+ \cup \infty$ | *max* | *min* | 0 | $\infty$ |
| Path with min. number of arcs | $\mathbb{N} \cup \infty$ | *min* | $+$ | $\infty$ | 0 |
| Shortest paths [82] | $\mathbb{R} \cup \infty$ | *min* | $+$ | $\infty$ | 0 |
| Max. reliability path | $\{a \mid 0 \leq a \leq 1\}$ | *max* | $*$ | 0 | 1 |
| Min. cost spanning tree [181] | $\mathbb{R}^+ \cup \infty$ | *min* | *max* | $\infty$ | 0 |

The APP also finds application in areas such as parsing and logic programming, and can be used as the basis of fast parallel algorithms for matrix inversion. Like the prefix sums computation, it is a remarkably versatile second-order function.

Noting that node $i$ is connected to node $j$ by a directed path if and only if it is connected by a directed path of length $\leq n - 1$, we have

$$\begin{aligned} M^* &= \bigoplus_{k=0}^{n-1} M^k \\ &= (M^0 \oplus M)^{n-1} \\ &= (M^0 \oplus M)^{2^l} \quad \text{for} \quad 2^l \geq n - 1 \end{aligned}$$

Therefore, to obtain an efficient shared memory parallel algorithm for the computation of the APP on matrix $M$ we need only set the main diagonal to $I_\otimes$ and repeatedly square the resulting matrix until we have a sufficiently large power. For an $n \times n$ matrix $M$, this method yields a circuit of depth $O(log^2 n)$ or, equivalently, a PRAM algorithm of time complexity $O(log^2 n)$ (Number of processors $= O(n^3)$ if we use the standard matrix multiplication algorithm).

Consider the problem of topologically ordering the $n$ vertices of a directed acyclic graph, i.e. assigning a number to each of the vertices such that there is no path from a vertex to a lower numbered one. There are several efficient sequential algorithms for this problem in which one successively numbers vertices from 1 to $n$. It is perhaps not immediately clear how one would shortcut this iterative process to achieve an algorithm which produces such an ordering in $o(n)$ parallel time. However, this problem can be solved in a straightforward way using the fast parallel APP algorithm. We can simply compute the connectivity or transitive closure of the dag using the parallel APP algorithm with $(or, and)$, and then sort the vertices by their in-degree in the closure. If there is a directed path from $i$ to $j$ in the dag, then $j$ will have a higher in-degree than $i$ in the closure.

The minimum cost spanning tree problem is another one which has simple, efficient sequential algorithms, e.g. the algorithms of Kruskal and Prim [66], but for which it is not immediately clear that it has a fast parallel algorithm. As noted above, Maggs and Plotkin [181] observed that if the weights of the edges in the graph are distinct, then the minimum cost spanning tree is precisely the set of edges $<i,j>$ in the graph for which $M_{ij} = M^*_{ij}$ where we compute the APP with $(min, max)$. [If the distinct weights property does not apply, then a simple modification can be made to the weights to achieve it.]

## 2.13. Expression Evaluation

Consider the problem of evaluating a given (binary tree) expression for a particular set of values of the arguments. We have seen previously that particular expressions, such as Horner expressions corresponding to polynomials, can be evaluated by a PRAM or circuit in a number of steps logarithmic in the size (number of leaves) of the expression. What about the evaluation of an arbitrary tree expression? A basic technique in the field of circuit complexity is *tree restructuring* [48, 77, 157, 196, 214, 272]. The technique was developed to show that every function (Boolean or arithmetic) which can be represented by an expression of size $s$ can also be represented by a circuit of depth $O(log\ s)$. One disadvantage of using this tree restructuring approach as a means of fast parallel expression evaluation on a PRAM is that it requires the calculation of appropriate points for tree splitting. When the cost of performing this tree splitting is taken into account the parallel complexity becomes $O(log^2 s)$ rather than $O(log\ s)$. If instead one uses *parallel tree contraction* [3, 30, 88, 190, 191, 192], then the $O(log\ s)$ parallel time bound can be achieved. Fast parallel tree contraction on a PRAM is another technique of very wide applicability.

## 2.14. The Class $\mathcal{NC}$

We have described a number of parallel algorithms for the PRAM, circuit and comparison network models. The circuits and comparison networks given can be directly transformed into corresponding PRAM algorithms. These idealised models have provided a robust framework for the investigation of parallel algorithms and their complexity [26, 54, 61, 62, 63, 80, 87, 89, 149, 223, 224, 266, 270]. One outcome of this work has been the development of an extensive set of results concerning $\mathcal{NC}$, the class of computational problems which can be solved on a PRAM by a deterministic algorithm in polylogarithmic time using only a polynomial number of processors. A major open problem in theoretical computer science is to determine whether $\mathcal{P}$, the class of polynomial time computable problems, is contained in $\mathcal{NC}$. If this were shown to be true then it would imply that every problem which had a fast (polynomial time) sequential algorithm also had a fast (polylogarithmic time), efficient (polynomial number of processors) parallel algorithm. Over the last decade, a large number of important problems in $\mathcal{P}$ have been shown to also lie in $\mathcal{NC}$. The following list of such problems is by no means complete.

> **Evaluation of expressions and programs** [87, 88, 89, 103, 106, 149, 158, 185, 189, 193, 224, 261]: Tree restructuring, tree contraction, expression evaluation, evaluation of straight-line algebraic programs of polynomial degree over a commutative semiring, evaluation of straight-line programs corresponding to dynamic programming algorithms, context-free recognition, parallel simulation of finite state automata, circuit value problem for planar monotone circuits, evaluation of set expressions.

**Logic** [79, 143, 185, 224, 251]: Term matching, term equivalence, evaluation of DATALOG logic programs with the polynomial-fringe property.

**Sets** [27, 60, 149, 166, 224, 264]: Sorting, selection, set operations, constructing Huffman trees.

**Sequences and strings** [57, 80, 87, 149, 216, 224, 229]: Prefix sums, merging, sequence comparison (string edit problem), string matching, recognising shuffle of two strings , longest common substring, finding squares in a string, pattern matching for $d$-dimensional patterns.

**Lists** [22, 23, 55, 56, 57, 80, 89, 149, 224]: List ranking.

**Arithmetic** [17, 18, 33, 77, 149, 213, 270, 272] : $n$-bit integer arithmetic (addition, multiplication, division), linear recurrences, polynomial arithmetic (evaluation, multiplication, division, GCD), evaluation of elementary functions (exp, ln, sin, etc.).

**Matrices** [47, 67, 80, 149, 224, 270]: Matrix multiplication, determinant, rank, inverse, solution of linear system, Cholesky factorisation.

**Graphs** [28, 58, 68, 80, 89, 92, 99, 101, 133, 149, 151, 162, 178, 224, 230, 240, 248]: Algebraic path problem (transitive closure, shortest paths, minimum cost spanning tree, topological ordering of dag), transitive reduction, connected components, biconnectivity, triconnectivity, Euler tours, ear decomposition, maximal independent set, symmetry breaking, lowest common ancestors, planarity, tree isomorphism, bipartite perfect matching, minimal elimination ordering.

**Combinatorial optimisation** [15]: Fixed dimension linear programming.

**Geometry** [8, 224, 278]: Convex hull in two and three dimensions, Voronoi diagrams and proximity problems, detecting segment intersections, triangulating a polygon, point location.

A number of interesting and important randomised $\mathcal{NC}$ algorithms have also been produced [5, 91, 150, 176, 197, 217, 225, 226] for graph problems such as depth-first search, constructing a perfect matching, maximum cardinality matching, maximum $s-t$ flow, planar graph isomorphism, and subtree isomorphism, and for various problems in computational geometry.

An exciting development over the last few years has been the development of a large number of new deterministic [35, 36, 37] and randomised [96, 117, 266] parallel algorithms for important problems, which achieve nearly-constant time on a CRCW PRAM. The problems include hashing [31, 94, 96, 97, 183, 184], dictionary (insert, delete, query operations) [96], integer sorting [38, 115, 183, 184, 218, 219], integer chain sorting [96, 115, 116], space allocation [96, 116], linear approximate compaction [96, 105, 183], estimation [96, 116], load balancing [93, 96, 116], leaders election [95, 96, 116, 184], generation of random permutations [183], 2-ruling set [55, 96, 105, 183], all nearest-smaller-values [35], approximate sum [96], efficient simulation of Maximum PRAM model on Tolerant PRAM model [95, 96, 117, 184]. These new algorithmic techniques provide a theoretical framework for the future development of tools which would automate a number of tedious aspects of practical parallel computation, such as processor allocation [94, 96, 105, 183] and memory allocation [147].

## 2.15. $\mathcal{P}$-Completeness

Using techniques analogous to those used in sequential computation to establish $\mathcal{NP}$-completeness, a number of problems in $\mathcal{P}$ have been shown to be $\mathcal{P}$-complete. A computational problem $\Pi$ is $\mathcal{P}$-complete if and only if $\Pi \in \mathcal{P}$ and ($\Pi \in \mathcal{NC} \Rightarrow \mathcal{P} \subseteq \mathcal{NC}$). The $\mathcal{P}$-complete problems are, in a sense, those in $\mathcal{P}$ for which it is hardest to obtain a fast, efficient PRAM algorithm. Showing that any one of them was in $\mathcal{NC}$ would imply that all problems in $\mathcal{P}$ had fast, efficient PRAM algorithms. The first $\mathcal{P}$-complete problems were established in the early 1970s [136, 137, 164]. Two recently published lists of such problems [110, 194] together contain around 250 problems. Those interested in $\mathcal{P}$-completeness results are strongly encouraged to consult [110]. Two very simple $\mathcal{P}$-complete problems are the following.

**Subset Closure**
**Given**: A finite set $X$, a binary operation $\circ$ on $X$, a subset $S \subseteq X$, and an element $x \in X$.
**To determine**: Whether $x$ is contained in the smallest subset of $X$ which contains $S$ and is closed under $\circ$.

**Monotone Circuit Value Problem** [102]
**Given**: A single-output Boolean circuit with $\{\wedge, \vee\}$ gates, and a set of values for the inputs.
**To determine**: The output.

Some other examples of $\mathcal{P}$-complete problems are:

**Evaluation of expressions and programs** [102, 182, 245]: Planar circuit value problem, arithmetic circuit value problem, type inference, deadlock detection.

**Logic** [78, 143, 279]: Unification, propositional Horn clause satisfiability, path systems, context-free grammar membership.

**Algebra**: Finite algebra, generalised word problem, subgroup equality, subgroup isomorphism, group rank.

**Arithmetic** [146]: Iterated mod.

**Matrices** [262]: Gaussian elimination with partial pivoting.

**Graphs** [104]: Maximum flow, lexicographically first maximal independent set, lexicographically first maximal path, lexicographically first depth-first search ordering, high degree subgraph, minimum degree elimination order.

**Combinatorial optimisation** [75, 154]: Linear programming, linear inequalities, first fit decreasing bin packing, nearest neighbour travelling salesman heuristic, two-player game.

**Geometry** [25]: Plane sweep triangulation, visibility layers.

Some simple examples of problems in $\mathcal{P}$ which are not currently known to be in $\mathcal{NC}$ or to be $\mathcal{P}$-complete are the following:

**Integer GCD**
**Given**: Two $n$-bit positive integers $a, b$.
**To determine**: $GCD(a, b)$

**Relative Primeness**
**Given**: Two $n$-bit positive integers $a, b$.
**To determine**: Whether $a, b$ are relatively prime.

**Stable Marriage**
**Given**: $n$ men and $n$ women plus a list of marital preferences for each person.
**To determine**: $n$ marriages that will stand the test of time.

**Ray Tracing**
**Given**: A set of $n$ mirrors of lengths $l_1, l_2, \ldots, l_n$ and their placements, a source $S$ and the trajectory of a single beam emitted from $S$, a designated mirror $M$.
**To determine**: If $M$ is hit by the beam. At the mirrors the angle of incidence of the beam equals the angle of reflection.

Serna and Spirakis [237, 238, 239] have investigated the extent to which solutions to important $\mathcal{P}$-complete problems such as linear programming, maximum flow and high degree subgraph can be approximated by fast, efficient PRAM algorithms if $\mathcal{P} \neq \mathcal{NC}$.

## 2.16. Parallel Efficiency

Matrix multiplication and the algebraic path problem are two fundamental computational problems which have fast $\mathcal{NC}$ algorithms. A large number of important problems in $\mathcal{P}$ can be shown to be in $\mathcal{NC}$ by a reduction to one of these two problems. (In the case of reductions to the APP, these are usually reductions to the transitive closure instance of that problem.) Unfortunately, many of the parallel algorithms so produced are extremely inefficient in terms of the number of processors required. For example, many problems on graphs with $v$ vertices and $e$ edges can be solved sequentially in time $O(v + e)$ or $O((v\ log\ v)\ +\ e)$. For a number of these problems, one can obtain a parallel algorithm with time complexity $O(log^2 v)$, but the algorithm requires $M(v)$ processors, where $M(v)$ is the sequential complexity of $v \times v$ matrix multiplication. As noted earlier, the best known upper bound on $M(v)$ is $O(v^{2{\cdot}376})$ [65] and we know that it cannot be less than proportional to $v^2$. Thus we have a number of fast PRAM algorithms for which the processor-time product is much greater than the time required to solve the problem by a sequential algorithm. To produce fast practical parallel algorithms for such problems we must avoid the brute force use of matrix multiplication and transitive closure on dense matrices. This difficulty has come to be known as the *matrix multiplication / transitive closure bottleneck*. It is particularly serious in applications where one is dealing with highly sparse matrices or graphs. In such cases, by embedding the problem in one involving dense matrices one may produce a theoretically fast algorithm, but it is unlikely to be of much practical value. In the last few years, substantial progress has been made on overcoming this bottleneck. A number of important new PRAM algorithms for sparse matrix and graph problems have been developed which are very efficient in their use of processors [68, 98, 99, 100, 114, 132, 144, 160, 162, 203, 244, 252].

As we have seen, the robustness of the PRAM model and the class $\mathcal{NC}$ has permitted the development of a rich theory of parallel algorithms and their complexity. However, the above considerations show that a naive preoccupation with $\mathcal{NC}$ may not result in efficient parallel algorithms for practical implementation. Probing $\mathcal{NC}$ further, it is not even clear that the class captures the informal notion of "problems which are amenable to parallel solution". Vitter and Simons [268] have shown that some $\mathcal{P}$-complete problems may be solved by parallel algorithms which are in a very reasonable sense, efficient. On the other hand, a problem such as searching an ordered list, which runs in logarithmic sequential time, is in $\mathcal{NC}$, irrespective of the existence of efficient parallel algorithms for that problem. In fact, searching does not admit efficient parallel algorithms.

As noted in [263], efficiency is a prime consideration in the design of parallel algorithms: one would like to solve a problem roughly $p$ times faster when using $p$ processors. This consideration is missing from the definition of $\mathcal{NC}$, instead the emphasis of $\mathcal{NC}$ theory is simply on the development of parallel algorithms which have polylogarithmic time complexity. In [161], Kruskal, Rudolph and Snir develop an alternative set of complexity classes for PRAM computations, and demonstrate that they provide an equally robust framework for studying parallel algorithms and complexity, but one which is more relevant in the context of practical parallel computing. Their emphasis is much more on the performance of a parallel algorithm relative to the best known sequential algorithm for the same problem. In describing the approach of [161] we will use the following notation. For a given problem, $S(n)$ will denote the sequential running time, $T(n)$ the parallel running time, and $P(n)$ the number of processors.

One very weak requirement, in terms of parallel performance, would be that the parallel algorithm demonstrate some unbounded speedup, i.e. $lim_{n\rightarrow\infty}T(n)/S(n) = 0$. A parallel algorithm for which $T(n) = O(n/loglog\ n)$ when $S(n) = O(n)$ would satisfy this condition. That kind of small improvement is unlikely to be sufficient in many cases. We are more likely to want to claim that a significant reduction in running time can be achieved through the use of parallelism, i.e. that $T(n)$ is a fast decreasing function of $S(n)$. The two obvious choices for such a function are captured in the following definition.

**Definition 2.1** *A parallel algorithm is* polynomially fast *if* $T(n) = O(S(n)^{\epsilon})$ *for some* $\epsilon < 1$, *and it is* polylogarithmically fast *if* $T(n) = O(log^k S(n))$ *for some fixed* $k$.

Reduction in running time has a cost. The number of processors must increase as fast as the speedup; generally it increases faster. The *inefficiency* of a parallel algorithm is the ratio $T(n) * P(n)/S(n)$, i.e. the ratio between the time-processor product for the parallel algorithm and the number of operations performed by the sequential algorithm.

**Definition 2.2** *A parallel algorithm has* constant inefficiency *if* $T(n) * P(n) = O(S(n))$, *it has* polylogarithmically bounded inefficiency *if* $T(n) * P(n) = O(S(n)\ log^k S(n))$ *for some fixed* $k$, *and it has* polynomially bounded inefficiency *if* $T(n) * P(n) = O(S(n)^k)$ *for some fixed* $k$.

Six interesting classes can be obtained by combining the two requirements on speedup with these three constraints on inefficiency.

| | Polylog Fast | Poly Fast |
|---|---|---|
| Constant Ineff. | **ENC** (Efficient, NC fast) | **EP** (Efficient, Parallel) |
| Polylog Ineff. | **ANC** (Almost efficient, NC fast) | **AP** (Almost efficient, Parallel) |
| Poly Ineff. | **SNC** (Semi efficient, NC fast) | **SP** (Semi efficient, Parallel) |

Kruskal, Rudolph and Snir [161] classify a large number of important problems within this framework.

> **ENC**: Sorting, merging, selection, prefix sums, polynomial evaluation, expression evaluation, fast Fourier transform, connected and biconnected components of dense graphs.
>
> **EP**: Various dense graph computations (strongly connected components, single source shortest paths, minimum cost spanning tree, directed graph reachability), monotone circuit value problem.
>
> **ANC**: Connectivity and biconnectivity of sparse graphs with $e = O(v)$ edges.
>
> **SP**: Depth-first search of undirected graphs, weighted bipartite matching, flows in $0-1$ networks.

## 2.17. Communication Complexity

In discussing the complexity of PRAM algorithms we have used the two standard complexity measures, namely parallel time complexity and number of processors. For the circuit model, parallel time complexity corresponds to the depth of the directed acyclic graph (dag), and for the comparison network model it corresponds to the number of levels in the network. In this section we consider the communication complexity of PRAM algorithms in a simplified setting first proposed by Papadimitriou and Ullman [204].

We model the computational problem to be solved as a dag, with nodes corresponding to the functions computed and arcs corresponding to functional dependencies. In most practical situations, the development of an appropriate dag is a major part of the algorithm design process but, for simplicity, we will assume it is fixed and given. Our problem is to efficiently schedule the dag on a $p$ processor parallel system which may have a large local memory at each processor, i.e. to assign each node of the dag to one or more processors in the system which will compute that node. (As we shall see, allowing more than one processor to compute the same node can sometimes save communication at no expense in terms of parallel time.) A schedule must satisfy the constraint that a node can only be computed at a given time step if its predecessors have been computed in previous time steps. We will use $t$ to denote the total number of time steps required for a schedule.

Communication complexity is captured in the following way. If node $v$ depends on node $u$, i.e. there is an arc from $u$ to $v$ in the dag, and $u, v$ are computed in distinct processors then that arc is said to be a communication arc. The communication complexity $c$ of a given schedule is simply the number of communication arcs in the dag. This measure captures an important practical cost in the implementation of parallel algorithms on a multiprocessor system, i.e. the total message traffic generated.

**Example.** Consider the ($2 \times 2$ diamond) dag on vertices $\{v_{0,0}, v_{0,1}, v_{1,0}, v_{1,1}\}$ which has the four arcs $\{<v_{0,0}, v_{0,1}>, <v_{0,1}, v_{1,1}>, <v_{0,0}, v_{1,0}>, <v_{1,0}, v_{1,1}>\}$. This can be scheduled in the following way on two processors $P_0, P_1$.

| $t$ | $P_0$ | $P_1$ |
|---|---|---|
| 1 | $v_{0,0}$ | |
| 2 | $v_{0,1}$ | $v_{1,0}$ |
| 3 | $v_{1,1}$ | |

giving a schedule with $c = 2, t = 3$. The alternative schedule

| $t$ | $P_0$ | $P_1$ |
|---|---|---|
| 1 | $v_{0,0}$ | $v_{0,0}$ |
| 2 | $v_{0,1}$ | $v_{1,0}$ |
| 3 | $v_{1,1}$ | |

yields $c = 1, t = 3$ and gives an example showing that allowing more than one processor to compute a node can reduce communication

Our main interest is in the tradeoff between parallel time and communication. First we note that for some simple dags there is no real issue concerning the best way to trade communication for time. For example, consider a complete binary tree on $n$ nodes. If we schedule the tree on $p$ processors, $p > 1$, then we must have $t \geq n/p$. As each processor, other than the one which computes the root, must compute a node value required by some other processor, we have also $c \geq p - 1$. Thus, we have $ct = \Omega(n)$. This lower bound for the communication-time product can be easily achieved by a schedule.

A more interesting dag is the diamond. The *diamond dag* is an $n \times n$ square mesh rotated 45 degrees, where the children of each node are the nodes immediately to the southeast and southwest, if they exist. More formally, it is the dag on $n^2$ nodes $\{v_{i,j} \mid 0 \leq i, j \leq n-1\}$ where there is an arc from $v_{i,j}$ to $v_{i,j+1}$ and from $v_{i,j}$ to $v_{i+1,j}$, where those nodes exist. The diamond dag arises in a number of important dynamic programming algorithms. It turns out that the diamond does not share with the binary tree the nice property that the best lower bounds on time and communication can be simultaneously achieved. Rather, there is a lower bound on the product $ct$ that is stronger than what is implied by the best lower bounds on $c$ and $t$ individually.

If we schedule an $n \times n$ diamond on $p$ processors, $p \leq n$, then we must have $t \geq n^2/p$. This lower bound on parallel time is easily matched by a fast "stripes" schedule in which processor $k$, $0 \leq k \leq p-1$, computes nodes $v_{i,j}$, for all $kn/p \leq i \leq ((k+1)n/p) - 1$ and $0 \leq j \leq n-1$. For this schedule we have $c = O(np)$. Now, let us consider a lower bound on communication. If we divide the nodes of the dag evenly among the $p > 1$ processors, then each will compute $n^2/p$ nodes, and it is not hard to show that there must be among the arcs on the nodes computed by any one of these $p$ processors at least $(n^2/p)^{1/2} = n/p^{1/2}$ communication arcs. Thus, we have $c = \Omega(np^{1/2})$. This lower bound on communication is easily matched by a schedule in which each processor computes a contiguous $(n/p^{1/2}) \times (n/p^{1/2})$ subdiamond of the dag, but for this method we have a rather higher parallel time of $O(n^2/p^{1/2})$. We have described two distinct parallel schedules for the $n \times n$ diamond, and have shown that one of them optimises time, while the other optimises communication. For both of these schedules we have $ct = O(n^3)$. Papadimitriou and Ullman [204] have shown that this bound on the communication-time product is, in fact, optimal to within a constant factor. This result demonstrates that there is an important tradeoff between communication and parallel time for the scheduling of the diamond dag on a multiprocessor. Papadimitriou and Ullman also studied a tradeoff between parallel time and communication delay for the diamond dag. The *communication delay* $d$ of a scheduled dag is defined to be the maximum number of communication arcs on any directed path in the dag. In [204] it is shown that for the $n \times n$ diamond, $(d+1)t = \Omega(n^2)$.

A number of other interesting communication-time tradeoff results have been obtained. Klawe and Paterson (see [204]) have shown a $ct = \Omega(n^4)$ lower bound for the dag where the children of node $v_{i,j}$ are all the nodes $v_{k,j}$ where $i < k$, and all the nodes $v_{i,l}$ where $j < l$. This dag arises in many important dynamic programming problems, and can be computed in $O(n^3)$ sequential time. Andivahis [24] has improved this tradeoff result to $c^2t = \Omega(n^7)$. Jayasimha and Loui [131] show a $ct = \Omega(n^3)$ tradeoff for a dag that corresponds to the solution of a triangular system of linear equations. Afrati et al. [4] show that the problem of finding, for a given dag, a schedule which minimises time, and also minimises communication as a secondary criterion, is $\mathcal{NP}$-complete, even if the dag is a tree, and the number of processors to be used is either given or open (the problem can be solved in polynomial time if the amount of communication is fixed.) Papadimitriou and Yannakakis [205] develop a polynomial algorithm which, for any dag, calculates within a factor of two the optimum weighted sum $t + \tau d$, for any $\tau$, when no bound on the number of processors is specified. In [140], it is shown that if $\tau$ is a fixed integer, then a dynamic programming approach can be used to obtain a polynomial time algorithm for solving the scheduling problem exactly.

Aggarwal, Chandra and Snir [7] have studied a model called the *local memory PRAM*, or *LPRAM*, which also captures both the communication and computation requirements of PRAM algorithms in a convenient way. An LPRAM is a CREW PRAM in which each processor is provided with an unlimited amount of local memory. Processors can simultaneously read from the same location in the global memory, but two or more are not allowed to simultaneously write into the same location. The input variables are initially available in the global memory, and the outputs must also be eventually there. The multiprocessor is a synchronous MIMD machine. In order to model the communication delay and computation time, it is convenient to restrict the machine such that, at every time step, the processors do one of the following:

> In one communication step, a processor can write, and then read a word from global memory.
>
> In a computation step, a processor can perform a simple operation on at most two values that are present in its local memory.

A computation is represented as a dag, and a schedule for a dag consists of a sequence of computation steps and communication steps. At a computation step each processor may evaluate a node of the dag; this evaluation can only take place at a processor when its local memory contains the values corresponding to all of the incoming arcs. After the computation step is completed the values for the outgoing arcs are held in the local memory. At a communication step, any processor may write into the global memory any value that is presently in its local memory, and then it may read into its local memory a value from the global memory.

**Example.** Consider the nine-node dag with arcs $\{<a,c>, <a,d>, <a,e>, <a,f>, <b,c>, <b,d>, <b,e>, <b,f>, <c,g>, <d,g>, <e,h>, <f,h><g,r>, <h,r>\}$. This can be scheduled to run on a four processor LPRAM in five communication steps and three computation steps as follows.

Comm. step 1 : $P_1, P_2, P_3, P_4$ read $a$.
Comm. step 2 : $P_1, P_2, P_3, P_4$ read $b$.
Comp. step 1 : $P_1, P_2, P_3, P_4$ compute $c, d, e, f$ respectively.
Comm. step 3 : $P_2, P_4$ write $d, f$ respectively. $P_1, P_3$ read $d, f$ respectively.

Comp. step 2 : $P_1$ computes $g$, $P_3$ computes $h$.
Comm. step 4 : $P_3$ writes $h$, $P_1$ reads it.
Comp. step 3 : $P_1$ computes $r$.
Comm. step 5 : $P_1$ writes $r$.

The dag can also, for example, be computed on a two processor LPRAM in four communication steps and four computation steps, or on a single processor LPRAM in three communication steps and seven computation steps. Note that the minimum communication delay (number of communication steps) and the minimum computation time (number of computation steps) may not be achievable by the same schedule.

Aggarwal, Chandra and Snir [7] show that two $n \times n$ matrices can be multiplied in $O(n^3/p)$ computation time and $O(n^2/p^{2/3})$ communication delay using $p$ processors, for $p \leq n^3/log^{3/2}n$, and that these bounds are optimal. They also show that any algorithm which uses only binary comparisons and sorts $n$ elements requires a communication delay of $\Omega((n\ log\ n)/(p\ log\ (n/p)))$ for $1 \leq p \leq n$, and also that this lower bound can be achieved by an algorithm with $O((n\ log\ n)/p)$ computation time. Other problems considered include computing an $n$-point FFT graph, computing binary trees, and computing the diamond dag.

## 3. Special Purpose Parallel Computing

We noted in the introduction to this paper that no single model of parallel computation had yet come to dominate developments in parallel computing in the way the von Neumann model has dominated sequential computing [84, 258, 259]. Instead we have a variety of models such as VLSI systems, systolic arrays and distributed memory multicomputers, in which the careful exploitation of network locality is crucial for algorithmic efficiency. In the different types of system it manifests itself in different ways. In a VLSI system, a design with good network locality will have short wires, and hence will require less area. An efficient systolic algorithm will have a simple, regular structure and use only nearest neighbour communication. An efficient multicomputer algorithm will be one which minimises the distance that messages have to travel in the network by careful mapping of the virtual process structure onto the physical processor architecture. Of course, an efficient algorithm for, say, a hypercube multicomputer will not necessarily perform well when run on, for example, a 2D array multicomputer with the same number of processors. We will use the generic term "special purpose" to refer to this type of parallel computing.

In a related paper [187], we describe a number of aspects of the work which has been done in recent years on the design, analysis, implementation and verification of special purpose parallel computing systems. The volume of published material on these topics is huge. A long, but by no means complete, bibliography is given at the end of [187]. Special purpose parallel systems are particularly appropriate in application areas where the goal of achieving the best possible performance is much more important than that of achieving an architecture-independent design. Some examples of such areas are: digital signal processing (filtering, transforms), image processing, computer vision, mobile robot control, particle simulation, cellular automata / lattice gas computations, dense matrix computations, communications and cryptography, speech recognition, computer graphics, game playing. For more examples, see [187].

The range of possible technologies for the development of such systems is extensive, varied, and growing. The following is a representative sample of those in use today.

VLSI systems (custom VLSI chips, field-programmable gate arrays)

Systolic architectures (application specific arrays, programmable systolic architectures)

Cellular automata machines

Multicomputers (2D and 3D arrays, pyramids, fat trees, hypercubes, butterflies etc.)

Neural systems (VLSI neural networks, analog VLSI systems)

On the technological horizon we have the prospects of various exotic, massively parallel systems such as lattice gas machines, quantum dot arrays, and various types of optical and holographic systems.

We will not pursue the fascinating world of special purpose parallel computing further in this paper. We simply note that, in the next few years, we will probably start to see a much sharper distinction between the two fields of special purpose parallel computing and general purpose parallel computing than at present. Those primarily concerned with achieving the maximum possible performance for a specific application (at any cost?) are likely to move more and more towards highly specialised architectures and technologies in the pursuit of performance gains. In contrast, those for whom it is important to achieve architecture-independence and portability in their designs, will increasingly seek a robust and lasting framework within which to develop their designs. In the remainder of this paper we will describe some ideas which have as their goal, the development of such a robust framework. As we shall see, the use of advanced technologies such as optics may also have an important role to play in achieving the communications performance required for efficient general purpose parallel computing systems. In this setting, the model is the central driving force and the optical hardware is simply one possible means of implementing the model. This can be contrasted with work on the development of special purpose optical parallel systems, where one directly exploits the capabilities of optical technology at the algorithmic level to produce, for example, very high performance image processing systems, signal processors, or neural networks.

## 4. General Purpose Parallel Computing

We have seen that an idealised model of parallel computation such as the PRAM can provide a robust framework within which to develop techniques for the design, analysis and comparison of parallel algorithms. A major issue in theoretical computer science since the late 1970s has been to determine the extent to which the PRAM and related models can be efficiently implemented on physically realistic distributed memory architectures. A number of new routing and memory management techniques have been developed which show that efficient implementation is indeed possible in many cases. In this section we describe some of the results which have been obtained, and discuss their significance for the future of general purpose parallel computing. Before doing so, we will give an *informal* description of what we mean by a general purpose parallel computer (GPPC).

A GPPC consists of a set of general purpose microprocessors connected by a communications network. The memory is fully distributed, with each processor having its own physically local memory module. The GPPC supports a single address space across all processors by allocating a part of each module to a common global memory system. Each processor thus has access to its own private address space of local variables, and to the global memory system. The purpose of the communications network is simply to support non-local memory accesses in a *uniformly efficient* way through message routing. By uniformly efficient, we mean that the time taken for a processor to read from, or write to, a non-local memory element in another processor-memory pair should be independent of which physical memory module the value is held in. The algorithm designer / programmer should not be aware of any hierarchical memory organisation based on the particular physical interconnect structure currently used in the communications network. Instead, performance of the communications network should be described only in terms of its global properties, e.g. the maximum time required to perform a non-local memory operation, and the maximum number of such operations which can simultaneously be in the network at any time.

A GPPC, as described, differs from a PRAM in that it has a two-level memory organisation. Each processor has its own physically local memory; all other memory is non-local and accessible at a uniform rate. In contrast, the PRAM has a one-level memory organisation; all memory in a PRAM is non-local. The GPPC and PRAM models are similar to the extent that they both have no notion of network locality. The GPPC differs from most current distributed memory multicomputers, e.g. hypercubes, in having no exploitable network locality, but is similar in that it is constructed as a network of processor-memory pairs. One formal model which would correspond reasonably closely to the (informally defined) GPPC would be a distributed memory multicomputer with full connectivity, i.e. with an interconnection structure corresponding to the complete graph [147]. Another is the bulk-synchronous parallel computer [258] which we will describe later in the paper.

Our purpose in giving this informal description of what we mean by a general purpose parallel computer is simply to describe, in very general terms, the main characteristics that we would expect any such computer to have. We have placed emphasis on the communication / memory organisation of such a machine. However, to support a coherent global memory it would also be necessary to provide support for features such as processor synchronisation.

The efficient implementation of a single address space on a distributed memory architecture requires an efficient method for the distributed routing of read and write requests, and of the replies to read requests, through the network of processors. In the following section we show how the idea of randomising can be used to produce an efficient distributed routing scheme for this problem.

## 4.1. Randomised Routing

A distributed memory multicomputer can be thought of as having $p$ processor-memory pairs located at distinct nodes of a $p$-node graph. Each processor can send packets to, and receive packets from, processors at adjacent nodes in the graph. Each edge of the graph can transmit one packet of information in unit time, and has a queue for storing packets that have to be transmitted along it.

A large number of graphs have been proposed as interconnection networks for such multicomputers. Two important parameters of any such graph are its *degree*, i.e. the maximum number of edges incident at any node, and its *diameter*, i.e. the maximum distance between any pair of nodes, where the distance between two nodes is the length of a shortest path between them. If implemented using conventional VLSI technology, a graph with low degree is likely to have advantages in terms of physical packaging. The advantage of using a graph with small diameter is, of course, that it will permit a packet to be sent quickly between any two nodes in the network. (The constraints of VLSI technology and, more generally, of three dimensional space, imply that some low diameter networks cannot be realised without having long wires. We will return to this wiring problem later in the paper and show how optics may provide a solution.) Some of the graphs which have been proposed as interconnection networks on $p$ nodes are listed in the following table, together with their degree and diameter.

| Name | Degree | Diameter |
|---|---|---|
| 1D array (ring) | 2 | $p/2$ |
| Shuffle-exchange | 3 | $2\ log\ p$ |
| Cube-connected-cycles | 3 | $(5/2)\ log\ p$ |
| 2D mesh of trees | 3 | $2\ log\ p$ |
| 3D mesh of trees | 3 | $2\ log\ p$ |
| 2D array (toroidal) | 4 | $\Theta(p^{1/2})$ |
| Butterfly (wrapped) | 4 | $2\ log\ p$ |
| de Bruijn graph | 4 | $log\ p$ |
| 3D array (toroidal) | 6 | $\Theta(p^{1/3})$ |
| Pyramid | 9 | $log\ p$ |
| Hypercube | $log\ p$ | $log\ p$ |

A large amount of work has been done in recent years on the development of efficient routing methods for such networks [167, 168, 169, 171, 259], on the efficient embedding of one network in another [122, 134, 167, 168, 169, 227], and on the demonstration of work-preserving emulations of one network by another [156, 167, 168, 231]. We will focus our attention here on the routing problem.

We consider the problem of packet routing on a $p$-processor network. Let *$h$-relation* denote the routing problem where each processor has at most $h$ packets to send to various points in the network, and where each processor is also due to receive at most $h$ packets from other processors. We are interested in the development of distributed routing methods in which the routing decisions made at a node at some point in time are based only on information concerning the packets that have already passed through the node at that time. In the nondistributed case where global information is available everywhere, the problem of routing is easier and well understood.

Let us first consider deterministic methods for distributed routing. We define a routing method to be *oblivious* if the path taken by each packet is entirely determined by its source and destination. (Note that the use of the term oblivious here is slightly different from its use in the context of comparison networks or circuits.) It is known [45, 142] that, for a 1-relation no deterministic oblivious routing method can do better than $\Omega(p^{1/2}/d)$ time steps, in the worst case, for any degree $d$ graph. The most obvious examples of deterministic oblivious approaches are greedy methods in which one sends all packets to their destination by a shortest path through the network. For 1-relations, the performance of greedy routing on a butterfly can be summarised as follows. All 1-relations can be realised in $O(p^{1/2})$ steps, which, as we have observed, is an optimal worst case bound for any such fixed degree network. A large number of 1-relations which arise in practical parallel computation, e.g. the bit-reversal permutation and the transpose permutation, provably require $\Theta(p^{1/2})$ steps. What about the "average case"? Define a *random 1-mapping* to be the routing problem where each processor has a single packet which is to be sent to a random destination. Greedy routing of a random 1-mapping on a butterfly will terminate in $O(log\ p)$ steps. Moreover, the fraction of all random 1-mappings which do not finish in $O(log\ p)$ steps is incredibly small, despite the fact that most of the 1-relations which seem to arise in practice do not finish in this time. We can probably conclude from these results that "typical" routing problems, in a practical sense, is a rather different concept from "typical" routing problems in a mathematical sense. The performance of greedy routing on a hypercube is very similar to the case of the butterfly. All 1-relations can be realised in $O(p^{1/2}/log\ p)$ steps, which is an optimal worst case bound for any *log p* degree network. For the average case, where each packet has a random destination, greedy routing will terminate in $O(log\ p)$ steps. In the case of the hypercube, there are exponentially many shortest paths for a greedy method to choose from, but even randomising among these choices still gives no better than $O(p^\alpha)$, $\alpha > 0$, steps for many 1-relations.

Another possible approach to deterministic routing is to use a sorting network of low depth. For example, from Batcher's odd-even merge sorting network [32, 155] we can obtain an interconnection network by associating each line in the sorting network with a node in the interconnection network, and each comparison element in the sorting network with an edge in the interconnection network. In this way, we can obtain a $p$-node graph of degree $O(log^2 p)$ and diameter $O(log^2 p)$. Any 1-relation routing problem can be realised in $O(log^2 p)$ steps by using the associated sorting network as a means of "sorting" the packet addresses. Note that this method is deterministic and requires no queueing. By using the sorting network of Ajtai, Komlos and Szemeredi [11], or its refinement by Paterson [207], we can improve this result to obtain degree, diameter, and number of time steps $O(log\ p)$. However, the complex structure of the networks involved, and the very large constant factors hidden in the $O(log\ p)$ bounds, rule out this approach as a practical option, at least for the present time.

We have seen that for the butterfly and hypercube, the performance of greedy routing on random 1-mappings is much better than on "worst case 1-relations", such as the bit-reversal permutation in the case of the butterfly. Around 1980, Valiant made the simple and striking observation that one could achieve efficient distributed routing, in terms of worst case performance, if one could reduce a 1-relation to something like the composition of two random 1-mappings. The resulting technique which emerged from this observation has come to be known as *two-phase randomised routing* [12, 167, 212, 253, 256, 257, 259, 260]. Using this approach, a 1-relation is realised by initially sending each packet to a random node in the network, using a greedy method. From there it is forwarded to the desired destination, again by a greedy method. Both phases of the routing correspond closely to the realisation of a random 1-mapping. Extensive investigation of this method, in terms of the number of steps required, size of buffers required etc., has shown that it performs extremely well, both in theory and in practice. The main theoretical results which follow from the use of randomised routing are summarised in the following two theorems.

**Theorem 4.1** *With high probability, every* 1*-relation can be realised on a p processor cube-connected-cycles, butterfly, 2D array and hypercube in a number of steps proportional to the diameter of the network.*

For the fixed degree networks in Theorem 4.1, this result is essentially optimal. For the (*log p*)-degree hypercube, the following stronger result can be obtained.

**Theorem 4.2** *With high probability, every $(\log p)$-relation can be realised on a $p$ processor hypercube in $O(\log p)$ steps.*

Proofs of Theorems 4.1 and 4.2 can be found in [259]. Randomised routing can also be used to achieve good worst case performance on networks such as the shuffle-exchange graph [12] and fat trees [109, 172].

An interesting alternative to using randomised routing on a standard, well defined network such as a butterfly, is to use *deterministic routing on a randomly wired network.* In [170, 254] it is shown that a simple deterministic routing algorithm can be used to realise a 1-relation in $O(\log p)$ steps on a randomly wired, bounded degree network known as a multibutterfly. An important feature of multibutterflies is that they have powerful expansion properties. In addition to permitting fast deterministic routing, such expander graphs [13, 14, 16, 141, 177, 255] also have very strong fault tolerance properties.

The techniques and results that we have described for various types of randomised routing show convincingly that for the problem of routing $h$-relations at least, there are a variety of theoretically and practically efficient methods which can be used. In order to show that we can efficiently simulate a shared address space on a distributed memory architecture we also need to show that we can deal with the problem of "hot spots", i.e. where a large number of processors simultaneously try to access the same memory module.

## 4.2. Hashing

In theoretical terms, one very effective method of uniformly distributing memory references is to hash the single address space. The hash function has, of course, to be efficiently computable. Hash functions for this purpose have been proposed and analysed by Mehlhorn and Vishkin [188]. They suggest using an elegant class of functions with some provably desirable properties: the class of polynomials of degree $O(\log p)$ in arithmetic modulo $m$, where $p$ is the number of processors and $m$ is the total number of words in the shared address space. As in the case of randomised routing, the idea of hashing the address space in this way has been subjected to extensive scrutiny in terms of both its theoretical and its practical performance. All of the available evidence suggests that it works extremely well in both respects. In fact, even constant degree polynomial hash functions, e.g. degree two, seem to work well in practice. (Recent results [73, 74] show that linear hash functions have certain limitations, at least in a theoretical sense, but that cubic hash functions work well. The results also suggest that quadratic hash functions may have some shortcomings.) One additional advantage of using a hashed address space is that we do not need then to resort to randomising to avoid bottlenecks in packet routing, simple deterministic methods will suffice.

Detailed technical accounts of the role of hashing in achieving efficient general purpose parallel computing can be found in [145, 147, 220, 221, 242, 258, 259]. We will mention only the following two results which demonstrate that distributed memory architectures can efficiently simulate PRAMs. Let $EPRAM(p,t)$ [ $CPRAM(p,t)$, $HYPERCUBE(p,t)$, $COMPLETE(p,t)$ ] denote the class of problems which can be solved on a $p$ processor EREW PRAM [ CRCW PRAM, hypercube, completely connected network, respectively ] in $t$ time steps.

**Theorem 4.3 (Valiant [259])**
*With high probability, $EPRAM(p\ log\ p, t/log\ p) \subseteq HYPERCUBE(p, t)$.*

**Theorem 4.4 (Karp, Luby and Meyer auf der Heide [147])**
*With high probability, $CPRAM(p\ log\ log\ p\ log^{*}p, t) \subseteq COMPLETE(p, t\ log\ log\ p\ log^{*}p)$.*

Theorems 4.3 and 4.4 show that PRAM algorithms with a degree of parallel slackness can be implemented on distributed memory architectures in a way which is optimal in terms of the processor-time product.

**Definition 4.5** *An $m$ processor algorithm, when implemented on an $n$ processor machine, where $n \leq m$, is said to have a* parallel slackness *factor of $m/n$ for that machine.*

Parallel slackness is an idea of fundamental importance in the area of general purpose parallel computing. If parallel algorithms and programs are designed so that they have more parallelism than is available in the machine, then the available parallel slackness can be effectively exploited to hide the kind of network latencies one finds in distributed memory architectures. The only requirement is that the processors provide efficient support for multithreading and fast context switching [44, 228]. Latency tolerance via multithreading is likely to be more effective on large scale general purpose parallel computing systems than the use of complex caching schemes for latency reduction [111, 175].

The idea of exploiting parallel slackness can even be carried over into the area of sequential computing. Much effort in recent years has been devoted to the development of complex heuristic techniques for the efficient prefetching of values from memory in sequential computations [50, 139, 195]. A radical alternative to this approach is, instead, to design parallel algorithms for implementation on sequential machines. The parallel slackness of the algorithm can then be exploited to achieve efficient prefetching. For more on this topic, see [265].

We have seen then that by achieving a degree of parallel slackness in program designs one can provide significant opportunities for the effective scheduling of those programs, by the programmer or by a compiler, to hide the various kinds of latencies which arise in both sequential and parallel computing. This idea of exploiting parallel slackness, or overdecomposition, is not new. It was, for example, a central idea in the early HEP parallel architecture [243], and has been used in its successors, Horizon and Tera [19]. In recent years it has come to be recognised as crucial, not only for efficient implementation of PRAM like models [147, 258, 259], but also for dataflow models [201]. The prospects for "autoparallelising" sequential code, which may be regarded as the extreme opposite of this approach, appear very bleak indeed.

The above analysis strongly supports the general principle that one should aim, at all times, to produce algorithms and programs which have more parallelism in them than is available in the machine (subject, of course, to the kinds of constraints on parallel efficiency which were discussed earlier in the paper.) In the future we can expect to see the development of a variety of programming languages for general purpose parallel computing. A clear message from our discussion is that such languages must permit, and indeed encourage, the development of programs which demonstrate a high degree of fine grain concurrency.

## 4.3. Combining

In the previous sections we have been concerned with the problems of implementing the weakest PRAM model, the EREW PRAM, on a distributed memory architecture. In practical parallel programming it is often convenient to permit concurrent access to a memory location, as in the CRCW PRAM model. An important practical case is that of broadcasting, where all processors simultaneously require the value of a single memory location.

One approach to the implementation of concurrent memory access is to use combining networks [107], i.e. networks that can combine and replicate messages in addition to delivering them in a point-to-point manner. The Fluent machine of Ranade [220, 221] provides an excellent example of how a CRCW PRAM can be efficiently implemented on a distributed memory architecture equipped with a combining network. The interconnection network of the Fluent machine is a butterfly. Let $CBUTTERFLY(p, t)$ denote the class of problems which can be solved on a $p$ processor butterfly with a combining network in $t$ time steps.

**Theorem 4.6 (Ranade [221])**
*With high probability,* $CPRAM(p, t) \subseteq CBUTTERFLY(p, t \; log \; p)$.

The theorem is established by showing that a $p$-processor Fluent machine can emulate a $p$-processor CRCW PRAM with only a slowdown of $O(log \; p)$, i.e. each parallel step of the PRAM requires at most $O(log \; p)$ steps on the Fluent machine, with high probability. The size of buffers required at each node of the network is constant. The following is a very brief account of the main ideas used in the emulation. The address space of the PRAM is hashed onto the memory modules of the Fluent machine using a hashing function chosen at random from a $(log \; n)$-universal class of hash functions [53, 273]. Suppose several processors wish to read the same memory location at the same time. Each one sends a message to the appropriate memory module along some path in the network. These paths will intersect to form a tree and there is, therefore, no need to send more than one request along any branch of the tree. A request simply waits at each node until (i) another request to the same destination arrives on the other input to the node, in which case the node combines the two and forwards the result along the tree, or (ii) the node determines that no future requests arriving on the other input will have the same destination. By always transmitting messages in sorted order of their destinations, and using "ghost messages" where necessary, one can achieve the above emulation result.

Concurrent access to shared variables in the Fluent machine is based on the multiprefix primitive. It has the form $MP(A, v, \oplus)$ where $A$ is a shared variable, $v$ is a value, and $\oplus$ is a binary associative operator. At any time step a processor can execute a multiprefix operation, with the constraint that if processors $P_i$ and $P_j$ execute $MP(A, v_i, \oplus_i)$ and $MP(A, v_j, \oplus_j)$, then $\oplus_i = \oplus_j$. The semantics of the multiprefix operation is defined as follows.

**Definition 4.7** *At time step $T$, let $P_A = \{p_1, p_2, \ldots, p_k\}$ be the set of processors referring to variable $A$, such that $p_1 < p_2 < \cdots < p_k$. Suppose that $p_i \in P_A$ executes instruction $MP(A, v_i, \oplus)$. Let $a_0$ be the value of $A$ at the start of time step $T$. Then, at the end of time step $T$, processor $p_i$ will receive the value $a_0 \oplus v_1 \oplus \cdots \oplus v_{i-1}$ and the value of variable $A$ will be $a_0 \oplus v_1 \oplus \cdots \oplus v_k$.*

Thus, when a set of processors perform a multiprefix operation on a common variable, the result is the same as if a single prefix operation were performed with the processors ordered by their index.

A parallel architecture which is very close in design to the Fluent machine is currently under construction at the University of Saarbrücken [1, 2].

Valiant [258] has investigated the extent to which concurrent access to shared variables can be provided without the use of combining networks. Working with the BSP model, he has shown that if one has enough parallel slackness, then one can support concurrent accesses in software on networks which have only point-to-point communication, with only constant slowdown, by using fast integer sorting methods. In the next section we describe the BSP model of parallel computation proposed by Valiant.

## 4.4. The BSP Model

For a detailed account of the model, and of the various routing and hashing results which can be obtained for it, the reader is referred to [258]. We concentrate here on presenting a view of (i) how a bulk-synchronous parallel architecture would be described, and (ii) how it would be used.

A *bulk-synchronous parallel (BSP) computer* consists of the following:

a set of processor-memory pairs

a communications network that delivers messages in a point-to-point manner

a mechanism for the efficient barrier synchronisation of all, or a subset, of the processors

There are no specialised combining, replication or broadcasting facilities. If we define a time step to be the time required for a single local operation, i.e. a basic operation on locally held data values, then the performance of any BSP computer can be characterised by the following four parameters:

$p$ = number of processors

$s$ = processor speed, i.e. number of time steps per second

$l$ = synchronisation periodicity, i.e. minimal number of time steps between successive synchronisation operations

$g$ = (total number of local operations performed by all processors in one second) / (total number of words delivered by the communications network in one second)

The parameter $l$ is related to the network latency, i.e. to the time required for a non-local memory access in a situation of continuous message traffic. The parameter $g$ corresponds to the frequency with which non-local memory accesses can be made; in a machine with a higher value of $g$ one must make non-local memory accesses less frequently. More formally, $g$ is related to the time required to realise $h$-relations in a situation of continuous message traffic; $g$ is the value such that an $h$-relation can be performed in $gh$ steps.

A BSP computer operates in the following way. A computation consists of a sequence of parallel *supersteps*, where each superstep is a sequence of steps, followed by a barrier synchronisation at which point any memory accesses take effect. During a superstep, each processor has a set of programs or threads which it has to carry out, and it can do the following:

perform a number of computation steps, from its set of threads, on values held locally at the start of the superstep

send and receive a number of messages corresponding to non-local read and write requests

The complexity of a superstep $S$ in a BSP algorithm is determined as follows. Let $L$ be the maximum number of local computation steps executed by any process or during $S$, $h_1$ be the maximum number of messages sent by any processor during $S$, and $h_2$ be the maximum number of messages received by any processor during $S$. The cost of $S$ is then $max\{l, L, gh_1, gh_2\}$ time steps. (An alternative is to charge $max\{l, L + gh_1, L + gh_2\}$ time steps for superstep $S$. The difference between these two costs will not, in general, be significant.)

The use of the parameters $l$ and $g$ to characterise the communications performance of the BSP computer contrasts sharply with the way in which communications performance is described for most distributed memory architectures on the market today. We are normally told many details about local network properties, e.g. the number of communications channels per node, the speed of those channels, the graph structure of the network etc. The way in which such descriptions emphasise local properties of the network, rather than its global properties, reflects the fact that most of those machines are designed to be used in a way where network locality is to be exploited. Those customers who have highly irregular problems, for which such exploitation is much more difficult, are often much less impressed by such machines when they are told about the global performance of the network in situations where network locality is not exploited.

A major feature of the BSP model is that it lifts considerations of network performance from the local level to the global level. We are thus no longer particularly interested in whether the network is a 2D array, a butterfly or a hypercube, or whether it is implemented in VLSI or in some optical technology. Our interest is in global parameters of the network, such as $l$ and $g$, which describe its ability to support non-local memory accesses in a uniformly efficient manner. As an aside, we note that it might be an interesting and instructive exercise to benchmark the various parallel architectures available today, in terms of such global parameters.

In the design and implementation of a BSP computer, the values of $l$ and $g$ which can be achieved will depend on (i) the capabilities of the available technology, and (ii) the amount of money that one is willing to spend on the communications network. As the computational performance of machines, i.e. the performance captured by $p$ and $s$, continues to grow, we will find that to keep $l$ and $g$ low it will be necessary to continually increase our investment in the communications hardware as a percentage of the total cost of the machine. A central thesis of the BSP and PRAM approaches to general purpose parallel computing is that if these costs are paid, then parallel machines of a new level of efficiency, flexibility, and programmability can be obtained.

On the basis of Theorems 4.1 and 4.2 we might expect to be able to achieve the following values of $l$ and $g$ for a $p$ processor BSP computer, by using the network shown.

| Network | $l$ | $g$ |
|---|---|---|
| 2D Array | $O(p^{1/2})$ | $O(p^{1/2})$ |
| Butterfly | $O(log\ p)$ | $O(log\ p)$ |
| Hypercube | $O(log\ p)$ | $O(1)$ |

These estimates are based entirely on the asymptotic degree and diameter properties of the graph. In a practical setting, the use of techniques such as wormhole routing [69, 167, 234, 235], rather than store and forward routing, would also have a significant impact on the values of $l$ and $g$ which could be achieved.

When $g$ is small, e.g. $g = 1$, the BSP computer corresponds closely to a PRAM, with $l$ determining the degree of parallel slackness required to achieve optimal efficiency. For a BSP computer of this kind, i.e. with a low $g$ value, we can use hashing to achieve efficient memory management [258]. The case $l = g = 1$ corresponds to the idealised PRAM, where no parallel slackness is required.

In designing algorithms for a BSP computer with a high $g$ value, we need to achieve a measure of *communication slackness* by exploiting thread locality in the two-level memory, i.e. we must ensure that for every non-local memory access we request, we are able to perform approximately $g$ operations on local data. To achieve architecture independence in the BSP model, it is therefore appropriate to design parallel algorithms which are parameterised not only by $n$, the size of the problem, and $p$, the number of processors, but also by $l$ and $g$. The following example of such an algorithm appears in [258]. The problem is the multiplication of two $n \times n$ matrices $A, B$ on $p \leq n^2$ processors. The standard $O(n^3)$ sequential algorithm is adapted to run on $p$ processors as follows. Each processor computes an $(n/p^{1/2}) \times (n/p^{1/2})$ submatrix of $C = A.B$. To do so it will require $n^2/p^{1/2}$ elements from $A$ and the same number from $B$. For each processor we thus have a computation requirement of $O(n^3/p)$ operations, since each inner product requires $O(n)$ operations, and a communications requirement of $O(n^3/p)$ for the number of non-local reads, since $p \leq n^2$. If we assume that both $A$ and $B$ are distributed uniformly amongst the $p$ processors, with each processor receiving $O(n^2/p)$ of the elements from each matrix, then the processors can simply replicate and send the approprate elements from $A$ and $B$ to the $2p^{1/2}$ processors requiring them. Therefore, we also have a communications requirement of approximately $n^2/p^{1/2} = O(n^3/p)$ for messages sent. We thus have a total parallel time complexity of $O(n^3/p)$, provided $l = O(n^3/p)$ and $g = O(n/p^{1/2})$. An alternative algorithm, given in [7], that requires fewer messages altogether, can be implemented to give the same optimal runtime, with $g$ as large as $O(n/p^{1/3})$ but with $l$ slightly smaller at $O(n^3/p \ log \ n)$.

The BSP model can be regarded as a generalisation of the PRAM model which permits the frequency of barrier synchronisation to be controlled. By capturing the network performance of a BSP computer in global terms using the values $l$ and $g$, the model enables us to design algorithms and programs which are parameterised by those values, and which can therefore be efficiently implemented on a range of BSP architectures with widely differing $l$ and $g$ values. It therefore provides a solution to the problems posed at the start of the paper. We have a simple and robust model which permits both scalable parallel performance and a high degree of architecture independence in software. Its simplicity also offers the prospect of our being able to develop a coherent framework for the design and analysis of parallel algorithms.

## 5. Optical Communication

Simple arguments can be used to show that various low diameter networks, such as the butterfly and the hypercube, cannot be implemented using VLSI technology without having long wires for some of the edges in the network. This has led some to conclude that such networks should be replaced by networks such as fat trees [172, 173] which are more efficient, in the VLSI model [174, 250, 267], in terms of their use of area or volume.

In this section we show that optical communication systems [34, 81] offer the prospect of a dramatic improvement in the efficiency with which non-local communication can be achieved. We show that a simple (and possibly cheap) optical interconnection architecture based on wavelength division multiplexing can be used (a) to solve the above mentioned VLSI wiring problem, and (b) to implement an extremely simple and efficient form of randomised optical routing. The results presented in this section are due, in this form, to Rao [222], although most of them are reinterpretations, in terms of optical communication, of known results in the theory of parallel computation. The power of optical communication has also been investigated in [21, 86, 118, 209, 258, 259].

Rao [222] considers various routing problems on a $p^{1/2} \times p^{1/2}$ 2D array of processors, where the processors on each row of the array are connected by an optical bus, and the processors on each column of the array are also connected by such a bus. We thus have $2p^{1/2}$ buses, each of length $p^{1/2}$. Each bus uses *wavelength division multiplexing (WDM)* [76] to support simultaneous communication between many disjoint pairs of processors on the same bus. This communications architecture will be referred to as the $p$ processor *mesh of buses (MOB)*. The idea of using a mesh of buses for interconnection was proposed by Wittie [275], and as a basis for optical interconnection networks by Dowd [76].

## 5.1. Solving The VLSI Wiring Problem

To solve the VLSI wiring problem we need only consider the simplest MOB in which the optical buses have fixed transmitters and receivers at each processor, i.e. where the transmitters and receivers are initially set (off-line) to achieve a certain communication pattern, e.g. a hypercube, and then remain fixed as that pattern is used. The first result shows that all networks can be emulated on the MOB with an efficiency related to the degree of the network.

**Theorem 5.1** *Any $p$ processor network $N$ of degree $d$ can be emulated on a $p$ processor MOB so that (i) there are $O(d)$ transmitters/receivers per processor, and (ii) each edge in $N$ is realised by a path of length at most three in the MOB.*

By an edge in the MOB we mean a channel on one of the buses, and by a path we mean a sequence of such edges. Theorem 5.1 is a consequence of the following well known results: (a) any such degree $d$ network has $O(d)$ perfect matchings, (b) each perfect matching corresponds to a 1-relation, and (c) by Hall's Theorem, all 1-relations can be routed (off-line) in a 2D array by permuting the packets of the rows, then permuting the packets of the columns, and then finally permuting the packets of the rows again. For details of the proof of part (c), see [167]. For networks such as the cube-connected-cycles and the hypercube we can do even better.

**Theorem 5.2** *A $p$ processor cube-connected-cycles network can be emulated on a $p$ processor MOB so that (i) there are $O(1)$ transmitters/receivers per processor, and (ii) each edge in the cube-connected-cycles is realised by a single edge in the MOB.*

**Theorem 5.3** *A $p$ processor hypercube network can be emulated on a $p$ processor MOB so that (i) there are $O(log\ p)$ transmitters/receivers per processor, and (ii) each edge in the hypercube is realised by a single edge in the MOB.*

Theorems 5.2 and 5.3 follow directly from standard VLSI layouts of those networks.

## 5.2. Randomised Optical Routing

We now consider a model of optical communication corresponding to the case where the buses in the MOB have tunable transmitters at each processor. Unlike the previous model, we are now able to dynamically change (on-line) the destination on the bus to which a processor can send a message. The model of computation is as follows. At any step, a processor can send a message directly to any other processor on the same bus. A message is successfully received if it was the only message sent to that destination in the step. A processor which successfully receives a message sends back an acknowledgement. (A single bus version of this optical model was investigated in [21].) We now describe a very simple randomised method for routing 1-relations on a MOB with tunable transmitters, which Rao [222] credits to Leighton and Maggs. The method proceeds in rounds, where each round consists of the following sequence of steps.

1. Each processor with a message sends it to a randomly selected position in its row bus.
2. Each processor that successfully received a message in step 1 forwards it using its column bus to the correct destination row for that message.
3. Each processor that successfully receives a message from step 2 forwards it using its row bus to the correct destination for that message.
4. For each message that was successfully received in step 3, an acknowledgement is sent to its source along the path it took. (It is easy to show that an acknowledgement gets back to any processor whose message was successfully sent to its destination.)
5. When a processor receives an acknowledgement, it does not send the message in later rounds.

**Theorem 5.4** *With high probability, the above method will route any* 1*-relation in* $O(log\ log\ p)$ *steps on a MOB.*

The $O(log\ log\ p)$ upper bound is easily established as follows. Each round takes six communication steps. The probability of a given message colliding with another in the first round is less than $1/e$. If we have $p/l$ messages left at the start of some round, then no more than about $p/l^2$ of them will be unsuccessful in that round. Therefore, after $k$ rounds we will have no more than about $p/e^{2^k}$ of them left, and thus $O(log\ log\ p)$ rounds will be sufficient. On a completely connected optical network with the same collision rules, the problem of routing a 1-relation can be trivially completed in one step, whereas the above method requires $O(log\ log\ p)$ steps on the MOB. For the problem of routing $h$-relations, where $h$ is at least logarithmic in $p$, Rao [222] has established the following powerful result.

**Theorem 5.5** *With high probability, any* $h$*-relation, with* $h \geq log\ p$*, can be routed in* $O(h)$ *steps on a MOB.*

Therefore, for such larger $h$-relations, the MOB is as powerful as a completely connected network! For the case where $1 < h < log\ p$, much less is known. In particular, no constant time method is known for routing a 2-relation on a completely connected optical network.

## 6. Challenges

In the previous sections we have seen that there are a variety of theoretically and practically efficient solutions to the problem of supporting a single address space on a distributed memory architecture. We have also seen that there are a large number of efficient, practical shared memory algorithms for important problems. In this section we briefly describe some of the main issues which need to be addressed in the future in order to continue the development of this framework for general purpose parallel computing based on fine grain concurrency in a shared address space.

### 6.1. Architecture

Most distributed memory architectures are based on conventional microprocessors [119, 120, 208]. We need alternative processor designs which can support a very large number of lightweight threads simultaneously, and can provide fast context switching, message handling, address translation, hashing etc. [44, 69, 71, 130, 274]. If such designs are not produced then we may find that the processors, and not the communications network, will be the bottleneck in the system.

We need to continue to develop improved networks for communication [69, 76, 170, 209, 222] and synchronisation [40, 138, 159]. There is currently great emphasis in parallel computing on various "Grand Challenge" applications in science and engineering. While not doubting the importance of these applications, we would suggest that perhaps the most important challenge for parallel architectures at the present time is to develop systems for which global "inefficiency parameters", such as $l$ and $g$ in the BSP model, are as low as possible. We have observed that the use of optical technologies may prove to be extremely important in this respect. In focusing our attention on the reduction of global parameters such as $l$ and $g$, we should note that it may not necessarily be cost-effective to try to obtain the extreme case of the PRAM, where $l$ and $g$ are both 1. At any given point in time, the capabilities and economics of the technologies available will determine the most cost-effective values of such parameters. An important advantage of the BSP model [258] over the PRAM [1, 2, 221] is that it provides an architecture-independent framework which allows us to take full advantage of whichever values of $l$ and $g$ are the most cost-effective at a given point in time.

Large general purpose parallel computer systems will inevitably suffer hardware faults of various kinds during their operation. We need to develop efficient techniques which can provide a degree of fault tolerance for processors, memories, and communications links. An interesting approach to this problem is to use the idea of information dispersal [179, 180, 215], where a space efficient redundant encoding of data is used to provide secure and reliable storage of information, and efficient fault tolerant routing of messages. Other approaches to the problems of fault tolerance are described in [152, 153, 241].

### 6.2. Algorithms

Although the potential for automating memory management via hashing is a major advantage of the BSP model, the BSP algorithm designer may wish to retain control of memory management in the two-level memory to achieve higher efficiency, e.g. on a BSP computer with a high value of $g$. A systematic study of bulk-synchronous algorithms remains to be done. Some first steps in this direction are described in [85, 258].

We need to continue to investigate the effects of hashing on the algorithmic performance achievable for important computational problems. There is already some theoretical evidence to confirm what one might intuitively expect, namely that any negative effects of hashing on algorithmic performance are much less dramatic on problems involving complex sparse irregular data structures, than they are on problems involving simple regular data structures. It is likely that future developments in most scientific and engineering applications will involve problems on large sparse irregular matrices and graphs, much more than on dense matrices and graphs. It is, therefore, likely that the advantages, in terms of programming simplicity, of using hashing will increasingly outweigh any disadvantages. We need to continue to develop fast, processor efficient parallel algorithms for sparse matrix and graph computations.

## 6.3. Languages and Software

The PRAM model was developed to facilitate the study of parallel algorithms and their complexity. In that context it has proved to be extremely useful. However, as we have pursued the design and implementation of parallel architectures based on the PRAM model, it has become clear that we have no well developed framework for the programming of such architectures. This can be contrasted with other approaches to general purpose parallel computing such as the actor and dataflow models, where there has been an intensive effort to develop a programming framework, although rather less on the investigation of parallel algorithms and their complexity. It is vital for the success of the approach described in this paper that we develop programming languages and methodologies for the kinds of parallel architectures proposed. Of the various challenges mentioned, this is perhaps the most important, and in many respects the most difficult one. The apparent unwillingness of many programmers of parallel machines to use anything other than minor variants of the sequential languages FORTRAN and C is widely perceived to be a major impediment to the continuing development of parallel computing. Another impediment is, of course, the “dusty decks” of old FORTRAN codes which many organisations are unwilling, or unable, to abandon. Many new parallel programming languages have been proposed and rejected over the last decade or so. Nevertheless, we must continue to seek a programming model which will provide a means of achieving the architecture-independence sought, while permitting scalable parallel performance on the kinds of architectures described. Some preliminary work in this direction can be found in [186]. It is to be hoped that as such a programming framework is developed we will also be able to provide a strategy for the migration of the dusty decks to the new architectures.

## 7. Other Approaches

A large number of approaches are currently being proposed as the basis of a framework for general purpose parallel computing. In this paper, the case for the BSP/PRAM approach has been presented. In this section we will briefly mention some of the other approaches.

Perhaps the most conservative of the alternatives is SIMD or data parallelism. Although a number of interesting algorithms have been developed for such architectures [41, 42, 43, 123, 246] , the model does not appear to be sufficiently general, even when extended to its SPMD form.

Another conservative approach is simply to continue with architectures based on message passing across a fixed set of channels [124, 125, 135]. Although such a model is adequate for the development of many special purpose parallel systems, and for low level systems programming, it does not appear to offer enough in terms of architecture-independence.

An approach related to message passing which appears to be more attractive is the actor model [9], which we might think of as message passing using names rather than a fixed set of channels. The names are first class objects and can be passed in messages. The graph of possible interactions between actors can thus change dynamically. The actor model provides a convenient framework for concurrent object-oriented programming [10]. Dally has developed an interesting parallel architecture, called the J-Machine [69, 70, 72], which supports the actor model.

The dataflow model has evolved considerably over the last decade. Modern designs for dataflow architectures [128, 129, 200, 201, 206] emphasise the importance of ideas such as efficient multithreading and the exploitation of parallel slackness, in the same way as the PRAM architectures do. There are, of course, major differences between the two approaches in terms of synchronisation control, scheduling control etc. It is not yet clear whether the freedom which the dataflow model offers the programmer has a cost to be paid in terms of scalable parallel performance.

Other approaches to general purpose parallel computing which have been suggested in recent years include asynchronous PRAMs [59, 90], block PRAMs [6], hierarchical PRAMs [121], tuple space [51, 52], graph reduction [210, 211], rewriting, and shared virtual memory.

## 8. Conclusion

The goals of general purpose parallel computing are to achieve both scalable parallel performance and architecture-independent parallel software. Despite much effort to find them, no serious theoretical impediments to the achievement of these goals have yet been found. We have argued that the bulk-synchronous parallel computer is a robust model of parallel computation which offers the prospect of achieving both requirements. The main challenge at the present time is to develop an appropriate programming framework for the BSP model.

Two other models which appear to offer the required architecture-independence are the actor and dataflow models. Unlike the BSP and PRAM models, which have global barrier synchronisation as the basic mechanism, these two models have at their core the idea of local, usually pairwise, synchronisation events. It is not yet clear whether these two models can offer the same scalability in parallel performance as we have demonstrated can be obtained for the BSP model. It is also unclear at present whether they can offer a convenient framework for the investigation of parallel algorithms and their complexity. Nevertheless, by virtue of their attractiveness in programming terms, they merit serious consideration.

Although we have stressed the differences between these various approaches, there is reason to believe that, at the architectural level, the BSP, PRAM, actor and dataflow models will require a number of similar mechanisms for efficient implementation; in particular, high performance global communications, uniform memory access, and multithreading to hide network latencies.

## Acknowledgements

I would like to thank Les Valiant for the numerous discussions we have had, over the last few years, on the development of a framework for general purpose parallel computing. The idea of exploiting bulk synchrony in parallel computation is due to him. I would also like to thank Les Valiant and Bill Gear for providing the opportunity for me to spend my sabbatical

leave from Oxford at NEC Research Institute, and to my colleagues at NECI for providing such a stimulating environment in which to work.

## References

[1] F Abolhassan, J Keller, and W J Paul. On the cost-effectiveness and realization of the theoretical PRAM model. Research Report 09/1991, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1991.

[2] F Abolhassan, J Keller, and W J Paul. On the cost-effectiveness of PRAMs. In *Proc. 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9, 1991.

[3] K Abrahamson, N Dadoun, D G Kirkpatrick, and T Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[4] F Afrati, C H Papadimitriou, and G Papageorgiou. Scheduling DAGs to minimize time and communication. In J H Reif, editor, *VLSI Algorithms and Architectures. 3rd Aegean Workshop on Computing (AWOC 88). LNCS Vol.319*, pages 134–138. Springer-Verlag, 1988.

[5] A Aggarwal and R J Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988.

[6] A Aggarwal, A K Chandra, and M Snir. On communication latency in PRAM computations. In *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[7] A Aggarwal, A K Chandra, and M Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.

[8] A Aggarwal, B Chazelle, L Guibas, C O'Dunlaing, and C Yap. Parallel computational geometry. In *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 468–477, 1985.

[9] G Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[10] G Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[11] M Ajtai, J Komlós, and E Szemerédi. Sorting in C log N parallel steps. *Combinatorica*, 3:1–19, 1983.

[12] R Aleliunas. Randomized parallel communication. In *Proc. 1st Annual ACM Symposium on Principles of Distributed Computing*, pages 60–72, 1982.

[13] N Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[14] N Alon. Eigenvalues, geometric expanders, sorting in rounds, and Ramsey Theory. *Combinatorica*, 6:207–219, 1986.

[15] N Alon and N Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. In *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 574–582, 1990.

[16] N Alon and V D Milman. Eigenvalues, expanders and superconcentrators. In *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 320–322, 1984.

[17] H Alt. Comparing the combinational complexities of arithmetic functions. *Journal of the ACM*, 35(2):447–460, April 1988.

[18] H Alt. On the efficient parallel evaluation of elementary functions. Technical Report B 90-03, Institut für Informatik, Freie Universität Berlin, March 1990.

[19] R Alverson, D Callahan, D Cummings, B Koblenz, A Porterfield, and B Smith. The Tera computer system. In *Proc. International Conference on Supercomputing. ACM SIGARCH Computer Architecture News, Vol.18 No.3*, pages 1–6. ACM Press, 1990.

[20] G M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conference 30*, pages 483–485, 1967.

[21] R J Anderson and G L Miller. Optical communication for pointer based algorithms. Technical Report CRI 88-14, University of Southern California, 1988.

[22] R J Anderson and G L Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[23] R J Anderson and G L Miller. Deterministic parallel list ranking. *Algorithmica*, 6(6):859–868, 1991.

[24] D Andivahis. Time-communication tradeoff in multiprocessing systems. Diploma Dissertation, National Technical University of Athens, Athens, Greece, 1984. (In Greek).

[25] M J Atallah, P Callahan, and M T Goodrich. P-complete geometric problems. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 317–326, 1990.

[26] M J Atallah, R Cole, and M T Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. In *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–160, 1987.

[27] M J Atallah, M T Goodrich, and S R Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. In J H Reif, editor, *VLSI Algorithms and Architectures (3rd Aegean Workshop on Computing, AWOC 88), LNCS Vol. 319*, pages 1–10. Springer-Verlag, 1988.

[28] M J Atallah and U Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29(3):330–337, July 1984.

[29] W C Athas and C L Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 12(8):9–24, August 1988.

[30] I Bar-On and U Vishkin. Optimal parallel generation of a computation tree form. *ACM Transactions on Programming Languages and Systems*, 7, No.2:348–357, April 1985.

[31] H Bast and T Hagerup. Fast and reliable parallel hashing. In *Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 50–61, 1991.

[32] K E Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[33] P W Beame, S A Cook, and H J Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15(4):994–1003, November 1986.

[34] T E Bell. Optical computing: A field in flux. *IEEE Spectrum*, 23(8):34–57, August 1986.

[35] O Berkman, D Breslauer, Z Galil, B Schieber, and U Vishkin. Highly parallelizable problems. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 309–319, 1989.

[36] O Berkman, J Ja'Ja', S Krishnamurthy, R Thurimella, and U Vishkin. Some triply-logarithmic parallel algorithms. In *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 871–881, 1990.

[37] O Berkman and U Vishkin. Recursive *-tree parallel data-structure. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 196–202, 1989.

[38] P C P Bhatt, K Diks, T Hagerup, V C Prasad, T Radzik, and S Saxena. Improved deterministic parallel integer sorting. *Information and Computation*, 94(1):29–47, September 1991.

[39] R S Bird and P Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[40] Y Birk, P B Gibbons, J L C Sanz, and D Soroker. A simple mechanism for efficient barrier synchronization in MIMD machines. Research Report RJ 7078, IBM Research, October 1989. Also appears in Proc. 1990 IEEE International Conference on Parallel Processing, Volume II Software, pages 195-198.

[41] G E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[42] G E Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[43] G E Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[44] B Boothe and A Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proc. 19th Annual International Symposium on Computer Architecture*, 1992. To appear.

[45] A Borodin and J E Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–145, February 1985.

[46] A Borodin and I J Munro. *The Computational Complexity of Algebraic and Numeric Problems.* American Elsevier, 1975.

[47] A Borodin, J von zur Gathen, and J E Hopcroft. Fast parallel matrix and GCD computations. *Information and Computation*, 52:241–256, 1982.

[48] R P Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21, No.2:201–206, April 1974.

[49] A W Burks, H H Goldstine, and J von Neumann. *Preliminary discussion of the logical design of an electronic computing instrument. Part 1, Volume 1.* The Institute of Advanced Study, Princeton, First edition, 28 June 1946. Second edition, 2 September 1947. Report to the U.S. Army Ordnance Department. Also appears in *Papers of John von Neumann on Computing and Computer Theory*, W Aspray and A Burks, editors. Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987, 97-142.

[50] D Callahan, K Kennedy, and A Porterfield. Software prefetching. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[51] N Carriero and D Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.

[52] N Carriero and D Gelernter. *How to write parallel programs: a first course.* MIT Press, 1990.

[53] J L Carter and M N Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[54] R Cole and U Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proc. 18th Annual ACM Symposium on Theory of Computing*, pages 206–219, 1986.

[55] R Cole and U Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Computation*, 70(1):32–53, 1986.

[56] R Cole and U Vishkin. Approximate parallel scheduling. Part 1: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17:128–142, 1988.

[57] R Cole and U Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, June 1989.

[58] R Cole and U Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, 1991.

[59] R Cole and O Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

[60] R J Cole. An optimally efficient selection algorithm. *Information Processing Letters*, 26(6):295–299, January 1988.

[61] S A Cook. The classification of problems which have fast parallel algorithms. In M Karpinski, editor, *Proc. Conference on Foundations of Computation Theory, LNCS Vol. 158*, pages 78–93, 1983.

[62] S A Cook. An overview of computational complexity. *Communications of the ACM*, 26(6):400–408, 1983.

[63] S A Cook. A taxonomy of problems with fast parallel algorithms. *Information and Computation*, 64((1-3)):2–22, 1985.

[64] S A Cook and C Dwork. Bounds on the time for parallel RAM's to compute simple functions. In *Proc. 14th Annual ACM Symposium on Theory of Computing*, pages 231–233, 1982.

[65] D Coppersmith and S Winograd. Matrix multiplication via arithmetic progressions. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 1–6, 1987.

[66] T H Cormen, C E Leiserson, and R L Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[67] L Csanky. Fast parallel matrix inversion algorithms. *SIAM Journal on Computing*, 5(4):618–623, December 1976.

[68] E Dahlhaus and M Karpinski. An efficient parallel algorithm for the minimal elimination ordering (MEO) of an arbitrary graph. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 454–459, 1989.

[69] W J Dally. Network and processor architecture for message-driven computers. In R Suaya and G Birtwistle, editors, *VLSI and Parallel Computation*, pages 140–222. Morgan Kaufmann, San Mateo, CA, 1990.

[70] W J Dally, A Chien, S Fiske, W Horwat, J Keen, M Larivee, R Lethin, P Nuth, and S Wills. The J-Machine: A fine-grain concurrent computer. In G X Ritter, editor, *Proc. Information Processing 89*, pages 1147–1153. Elsevier Science Publishers, B. V., 1989.

[71] W J Dally et al. The Message-Driven Processor. In *Proc. Hot Chips III Symposium*, 1991.

[72] W J Dally and D S Wills. Universal mechanisms for concurrency. In E Odijk, M Rem, and J-C Syre, editors, *Proc. PARLE 89: Parallel Architectures and Languages Europe. LNCS Vol.365*, pages 19–33. Springer-Verlag, 1989.

[73] M Dietzfelbinger. On limitations of the performance of universal hashing with linear functions. Forschungsbereich Nr. 84, Fachbereich 17, Universität-GH Paderborn, June 1991.

[74] M Dietzfelbinger, J Gil, Y Matias, and N Pippenger. Polynomial hash functions are reliable. In *Proc. 19th International Colloquium on Automata, Languages and Programming, LNCS.* Springer-Verlag, 1992.

[75] D Dobkin, R J Lipton, and S Reiss. Linear programming is log-space hard for P. *Information Processing Letters*, 8:96–97, 1979.

[76] P W Dowd. High performance interprocessor communication through optical wavelength division multiple access channels. In *Proc. 18th Annual International Symposium on Computer Architecture*, pages 96–105, 1991.

[77] P E Dunne. *The Complexity of Boolean Networks.* Academic Press, 1988.

[78] C Dwork, P C Kanellakis, and J Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[79] C Dwork, P C Kanellakis, and L J Stockmeyer. Parallel algorithms for term matching. *SIAM Journal on Computing*, 17(4):711–731, August 1988.

[80] D Eppstein and Z Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.*, 3:233–283, 1988.

[81] D G Feitelson. *Optical Computing. A Survey for Computer Scientists.* MIT Press, Cambridge, MA, 1988.

[82] R W Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[83] S Fortune and J Wyllie. Parallelism in random access machines. In *Proc. 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[84] C W Gear, editor. *Computation and Cognition. Proceedings of the First NEC Research Symposium.* SIAM Press, 1991. Panel Session - The Future of Parallelism, pages 153-168.

[85] A V Gerbessiotis and L G Valiant. Direct bulk-synchronous parallel algorithms. Technical Report TR-10-92, Aiken Computation Laboratory, Harvard University, 1992. To appear in Proc. 3rd Scandinavian Workshop on Algorithm Theory, July 8-10, 1992. LNCS, Springer-Verlag.

[86] M Gereb-Graus and T Tsantilas. Efficient optical communication in parallel computers. In *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992 (to appear).

[87] A M Gibbons and W Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, UK, 1988.

[88] A M Gibbons and W Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Information and Computation*, 81:32–45, 1989.

[89] A M Gibbons and P Spirakis, editors. *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation.* Cambridge International Series on Parallel Computation. Cambridge University Press, Cambridge, UK, 1992.

[90] P B Gibbons. A more practical PRAM model. In *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[91] P B Gibbons, R M Karp, G L Miller, and D Soroker. Subtree isomorphism is in random NC. In J H Reif, editor, *VLSI Algorithms and Architectures. 3rd Aegean Workshop on Computing (AWOC 88). LNCS Vol.319*, pages 43–52. Springer-Verlag, 1988.

[92] P B Gibbons, R M Karp, V Ramachandran, D Soroker, and R E Tarjan. Transitive reduction in parallel via branchings. Technical Report CS-TR-171-88, Department of Computer Science, Princeton University, July 1988.

[93] J Gil. Fast load balancing on a PRAM. In *Proc. 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 10–17, 1991.

[94] J Gil and Y Matias. Fast hashing on a PRAM - designing by expectation. In *Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1991.

[95] J Gil and Y Matias. Leaders election without a conflict resolution rule - Fast and efficient randomized simulations among CRCW PRAMs. In *Proc. 1st Latin American Informatics Symposium*, 1991.

[96] J Gil, Y Matias, and U Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 698–710, 1991.

[97] J Gil, F Meyer auf der Heide, and A Wigderson. Not all keys can be hashed in constant time. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 244–253, 1990.

[98] J R Gilbert and H Hafsteinsson. Parallel solution of sparse linear systems. In R Karlsson and A Lingas, editors, *Proc. 1st Scandinavian Workshop on Algorithm Theory. LNCS Vol. 318*, pages 145–153. Springer-Verlag, 1988.

[99] A V Goldberg, S A Plotkin, and G E Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal of Discrete Mathematics*, 1(4):434–446, November 1988.

[100] M Goldberg and T Spencer. Constructing a maximal independent set in parallel. *SIAM Journal of Discrete Mathematics*, 2(3):322–328, August 1989.

[101] M Goldberg and T Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 18(2):419–427, April 1989.

[102] L M Goldschlager. The monotone and planar circuit value problems are log space complete for P. *ACM Sigact News*, 9(2):25–29, 1977.

[103] L M Goldschlager. A space efficient algorithm for the monotone planar circuit value problem. *Information Processing Letters*, 10(1):25–27, February 1980.

[104] L M Goldschlager, R A Shaw, and J Staples. The maximum flow problem is log space complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[105] M T Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 711–722, 1991.

[106] M T Goodrich and S R Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 190–196, 1989.

[107] A Gottlieb, R Grishman, C P Kruskal, K P McAuliffe, L Rudolph, and M Snir. The NYU Ultracomputer - Designing an MIMD, shared-memory parallel machine. *IEEE Transactions on Computers*, 32:75–89, 1983.

[108] A C Greenberg, R E Ladner, M S Paterson, and Z Galil. Efficient parallel algorithms for linear recurrence computation. *Information Processing Letters*, 15(1):31–35, August 1982.

[109] R I Greenberg and C E Leiserson. Randomized routing on fat-trees. In *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 241–249, 1985.

[110] R Greenlaw, H J Hoover, and W L Ruzzo. A compendium of problems complete for P. Technical Report TR 91-05-01, Department of Computer Science, University of Washington, June 1991.

[111] A Gupta, J Hennessy, K Gharachorloo, T Mowry, and W-D Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proc. 18th Annual International Symposium on Computer Architecture*, pages 254–263, 1991.

[112] J L Gustafson, G R Montry, and R E Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, July 1988.

[113] J L Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[114] T Hagerup. Optimal parallel algorithms on planar graphs. *Information and Computation*, 84(1):71–96, January 1990.

[115] T Hagerup. Constant-time parallel integer sorting. In *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 299–306, 1991.

[116] T Hagerup. Fast parallel space allocation, estimation and integer sorting. Technical Report 03/1991, Max-Planck-Institut für Informatik, Saarbrücken, April 1991.

[117] T Hagerup. The log-star revolution. In *Proc. Annual Symposium on Theoretical Aspects of Computer Science*, 1992.

[118] A Hartmann and S Redfield. Design sketches for optical crossbar switches intended for large scale parallel processing applications. *Optical Engineering*, 29(3):315–327, 1989.

[119] J Hennessy. VLSI processor architecture. *IEEE Transactions on Computers*, C-33(11):1221–1246, December 1984.

[120] J L Hennessy and D A Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[121] T H Heywood. A practical hierarchical model of parallel computation. Technical Report SU-CIS-91-39, School of Computer and Information Science, Syracuse University, November 1991.

[122] P A J Hilbers. *Processor Networks and Aspects of the Mapping Problem*, volume 2 of *Cambridge International Series on Parallel Computation.* Cambridge University Press, Cambridge, UK, 1991.

[123] W D Hillis and G L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[124] C A R Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[125] C A R Hoare. The transputer and occam: A personal story. *Concurrency: Practice and Experience*, 3(4):249–264, August 1991.

[126] P Hudak. Concept, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[127] P Hudak, S Peyton Jones, and P Wadler, editors. Report on the Programming Language Haskell - A Non-Strict, Purely Functional Language. Version 1.1 , 1991.

[128] R A Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 131–140, 1988.

[129] R A Iannucci. *Parallel Machines: Parallel Machine Languages.* Kluwer Academic Publishers, 1990.

[130] INMOS Limited. *Transputer Reference Manual.* Prentice Hall, 1988.

[131] D N Jayasimha and M C Loui. The communication complexity of parallel algorithms. Technical Report CSRD 629, University of Illinois at Urbana-Champaign, 1986.

[132] D B Johnson and P Metaxas. Connected components in $O(lg^{3/2} \mid V \mid)$ parallel time for the CREW PRAM. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 688–697, 1991.

[133] D B Johnson and P Metaxas. Optimal algorithms for the vertex updating problem of a minimum spanning tree. In *Proc. 6th International Parallel Processing Symposium*, pages 306–314. IEEE Press, 1992.

[134] S L Johnsson. Communication in network architectures. In R Suaya and G Birtwistle, editors, *VLSI and Parallel Computaion*, pages 223–389. Morgan Kaufmann, San Mateo, CA, 1990.

[135] G Jones and M Goldsmith. *Programming in occam 2.* Prentice Hall, 1988.

[136] N D Jones and W T Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3:105–117, 1977.

[137] N D Jones, Y E Lien, and W T Laaser. New problems complete for nondeterministic log space. *Mathematical Systems Theory*, 10:1–17, 1976.

[138] H F Jordan. A special purpose architecture for finite element analysis. In *Proc. IEEE International Conference on Parallel Processing*, pages 263–266, 1978.

[139] N P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[140] H Jung, L Kirousis, and P Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays. In *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 254–264, 1989.

[141] N Kahale. Better expansion for Ramanujan graphs. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 398–404, 1991.

[142] C Kaklamanis, D Krizanc, and T Tsantilas. Tight bounds for oblivious routing in the hypercube. *Mathematical Systems Theory*, 24:223–232, 1991.

[143] P C Kanellakis. Logic programming and parallel complexity. Technical Report CS-86-23, Department of Computer Science, Brown University, October 1986.

[144] M Y Kao and P N Klein. Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 181–192, 1990.

[145] A Karlin and E Upfal. Parallel hashing - An efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, 1988.

[146] H J Karloff and W L Ruzzo. The iterated mod problem. *Information and Computation*, 80(3):193–204, March 1989.

[147] R M Karp, M Luby, and F Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proc. 24th Annual ACM Symposium on Theory of Computing*, 1992. To appear.

[148] R M Karp, R E Miller, and S Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[149] R M Karp and V Ramachandran. Parallel algorithms for shared-memory machines. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity*, pages 869–941. North Holland, 1990.

[150] R M Karp, E Upfal, and A Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6:35–48, 1986.

[151] R M Karp and A Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM*, 32(4):762–773, October 1985.

[152] Z M Kedem, K V Palem, A Raghunathan, and P G Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 381–390, 1991.

[153] Z M Kedem, K V Palem, and P G Spirakis. Efficient robust parallel computations. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 138–148, 1990.

[154] G A P Kindervater, J K Lenstra, and D B Shmoys. The parallel complexity of TSP heuristics. *Journal of Algorithms*, 10:249–270, 1989.

[155] D E Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[156] R Koch, F T Leighton, B M Maggs, S B Rao, and A L Rosenberg. Work-preserving emulations of fixed-connection networks. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 227–240, 1989.

[157] S R Kosaraju. Parallel evaluation of division-free arithmetic expressions. In *Proc. 18th Annual ACM Symposium on Theory of Computing*, pages 231–239, 1986.

[158] S R Kosaraju. On the parallel evaluation of classes of circuits. In K V Nori and C E Veni Madhavan, editors, *Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science. LNCS Vol.472*, pages 232–237. Springer-Verlag, 1990.

[159] C P Kruskal, L Rudolph, and M Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, 1988.

[160] C P Kruskal, L Rudolph, and M Snir. Techniques for parallel manipulation of sparse matrices. *Theoretical Computer Science*, 64:135–157, 1989.

[161] C P Kruskal, L Rudolph, and M Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[162] C P Kruskal, L Rudolph, and M Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5:43–64, 1990.

[163] H T Kung. New opportunities in multicomputers. In C W Gear, editor, *Computation and Cognition. Proceedings of the First NEC Research Symposium*, pages 1–21. SIAM Press, 1991.

[164] R E Ladner. The circuit value problem is log space complete for P. *ACM Sigact News*, 7(1):18–20, 1975.

[165] R E Ladner and M J Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[166] L L Larmore and T M Przytycka. Parallel construction of trees with optimal weighted path length. In *Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 71–80, 1991.

[167] F T Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.

[168] F T Leighton and C E Leiserson. Advanced parallel and VLSI computation. Research Seminar Series MIT/LCS/RSS 7, Laboratory for Computer Science, Massachusetts Institute of Technology, December 1989.

[169] F T Leighton, C E Leiserson, and M Klugerman. Theory of parallel and VLSI computation. Research Seminar Series MIT/LCS/RSS 10, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

[170] F T Leighton and B M Maggs. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 384–389, 1989.

[171] F T Leighton, B M Maggs, and S B Rao. Universal packet routing algorithms. In *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[172] C E Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C–34(10):892–901, October 1985.

[173] C E Leiserson and B M Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

[174] T Lengauer. VLSI theory. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity*, pages 835–868. North Holland, 1990.

[175] D Lenoski, J Laudon, K Gharachorloo, W-D Weber, A Gupta, J Hennessy, M Horowitz, and M S Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[176] A Lingas and M Karpinski. Subtree isomorphism is NC reducible to bipartite perfect matching. *Information Processing Letters*, 30(1):27–32, January 1989.

[177] A Lubotzky, R Phillips, and P Sarnak. Ramanujan graphs. *Combinatorica*, 8(3):261–277, 1988.

[178] M Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, November 1986.

[179] Y-D Lyuu. Fast fault-tolerant parallel communication and on-line maintenance using information dispersal. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 378–387, 1990.

[180] Y-D Lyuu. *Information Dispersal and Parallel Computation.* Cambridge International Series on Parallel Computation. Cambridge University Press, Cambridge, UK, 1992. To appear.

[181] B M Maggs and S A Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26:291–293, 1988.

[182] H G Mairson. Type inference for the simply-typed lambda calculus is complete for PTIME: Proof by computer program, 1991. Article on TYPES mailing list.

[183] Y Matias and U Vishkin. Converting high probability into nearly-constant time, with applications to parallel hashing. In *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 307–316, 1991.

[184] Y Matias and U Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12:573–606, 1991.

[185] E W Mayr. Theoretical aspects of parallel computation. In R Suaya and G Birtwistle, editors, *VLSI and Parallel Computaion*, pages 85–139. Morgan Kaufmann, San Mateo, CA, 1990.

[186] W F McColl. Phase: A programming language for general purpose parallel computing. Technical report, NEC Research Institute, Princeton, 1992. (In preparation).

[187] W F McColl. Special purpose parallel computing. In Gibbons and Spirakis [89].

[188] K Mehlhorn and U Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[189] G L Miller, V Ramachandran, and E Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal on Computing*, 17(4):687–695, August 1988.

[190] G L Miller and J H Reif. Parallel tree contraction and its application. In *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[191] G L Miller and J H Reif. Parallel tree contraction. Part I: Fundamentals. In S Micali, editor, *Randomness and Computation. Vol.5*, pages 47–72. JAI Press, 1989.

[192] G L Miller and J H Reif. Parallel tree contraction. Part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, December 1991.

[193] G L Miller and S H Teng. Dynamic parallel complexity of computational circuits. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 254–263, 1987.

[194] S Miyano, S Shiraishi, and T Shoudai. A list of P-Complete problems. RIFIS Technical Report RIFIS-TR-CS-17, Research Institute of Fundamental Information Science, Kyushu University, October 1989.

[195] T Mowry and A Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[196] D E Muller and F P Preparata. Restructuring of arithmetic expressions for parallel evaluation. *Journal of the ACM*, 23:534–543, 1976.

[197] K Mulmuley, U V Vazirani, and V V Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.

[198] I J Munro and M S Paterson. Optimal algorithms for parallel polynomial evaluation. *Journal of Computer and System Sciences*, pages 189–198, 1973.

[199] R S Nikhil. Id - Language Reference Manual. Version 90.1. Computation Structures Group Memo 284-2, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1991.

[200] R S Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. 16th Annual International Symposium on Computer Architecture*, pages 262–272, 1989.

[201] R S Nikhil, G M Papadopolous, and Arvind. *t: A killer micro for a brave new world. Computation Structures Group Memo 325, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1991.

[202] V Pan. *How to Multiply Matrices Faster. LNCS Vol.179.* Springer-Verlag, 1984.

[203] G E Pantziou, P G Spirakis, and C D Zaroliagis. Efficient parallel algorithms for shortest paths in planar graphs. In J R Gilbert and R Karlsson, editors, *Proc. 2nd Scandinavian Workshop on Algorithm Theory. LNCS Vol.447*, pages 288–300. Springer-Verlag, 1990.

[204] C H Papadimitriou and J D Ullman. A communication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, August 1987.

[205] C H Papadimitriou and M Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[206] G M Papadopolous and K R Traub. Multithreading: A revisionist view of dataflow architectures. In *Proc. 18th Annual International Symposium on Computer Architecture*, pages 342–351, 1991.

[207] M S Paterson. Improved sorting networks with $O(log\ N)$ depth. *Algorithmica*, 5(1):75–92, 1990.

[208] D A Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[209] R Paturi, D-T Lu, J E Ford, S C Esener, and S H Lee. Parallel algorithms based on expander graphs for optical computing. *Applied Optics*, 30(8):917–927, March 1991.

[210] S Peyton-Jones and D Lester. *Implementing Functional Languages.* Prentice Hall, 1992.

[211] S L Peyton Jones. Parallel implementation of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[212] N J Pippenger. Parallel communication with limited buffers. In *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 127–136, 1984.

[213] N J Pippenger. The complexity of computations by networks. *IBM Journal of Research and Development*, 31(2):235–243, March 1987.

[214] F P Preparata and D E Muller. Efficient parallel evaluation of Boolean expressions. *IEEE Transactions on Computers*, 25:548–549, 1976.

[215] M O Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.

[216] M O Rabin. Optimal parallel pattern matching through randomization. In *Proc. Sequences 91 Conference*, 1991.

[217] S Rajasekaran and J H Reif. Randomized parallel computation. In S K Tewksbury, B W Dickinson, and S C Schwartz, editors, *Concurrent Computations - Algorithms, Architecture, and Technology. Proc. 1987 Princeton Workshop on Algorithms, Architecture and Technology Issues for Models of Concurrent Computation*, pages 181–202. Plenum Press, 1988.

[218] R Raman. The power of collision - Randomized parallel algorithms for chaining and integer sorting. In *Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science, LNCS Vol.472*, pages 161–175. Springer-Verlag, 1990.

[219] R Raman. Optimal sub-logarithmic time integer sorting on a CRCW PRAM. Technical report, 1991. Submitted for publication.

[220] A G Ranade. How to emulate shared memory. In *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[221] A G Ranade. Fluent parallel computation. Ph.D. Thesis, Department of Computer Science, Yale University, May 1989.

[222] S B Rao. Properties of an interconnection architecture based on wavelength division multiplexing. Technical Report TR-92-009-3-0054-2, NEC Research Institute, Princeton, January 1992.

[223] J H Reif. Depth first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[224] J H Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, San Mateo, CA, 1992. To appear.

[225] J H Reif and S Sen. Randomization in parallel algorithms and its impact on computational geometry. In H Djidjev, editor, *Proc. International Symposium on Optimal Algorithms. LNCS Vol.401*, pages 1–8. Springer-Verlag, 1989.

[226] J H Reif and S Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, 7(1):91–117, 1992.

[227] A L Rosenberg. On validating parallel architectures via graph embeddings. In S K Tewksbury, B W Dickinson, and S C Schwartz, editors, *Concurrent Computations - Algorithms, Architecture, and Technology*, pages 99–115. (Proc. 1987 Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation), Plenum Press, 1988.

[228] R H Saavedra-Barrera, D E Culler, and T von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1990.

[229] A Saoudi, M Nivat, C Pandu Rangan, R Sundaram, and G D S Ramkumar. A parallel algorithm for recognizing the shuffle of two strings. In *Proc. 6th International Parallel Processing Symposium*, pages 112–115, 1992.

[230] B Schieber and U Vishkin. On finding lowest common ancestors: Simplification and parallelization. In J H Reif, editor, *VLSI Algorithms and Architectures. 3rd Aegean Workshop on Computing (AWOC 88). LNCS Vol.319*, pages 111–123. Springer-Verlag, 1988.

[231] E Schwabe. On the computational equivalence of hypercube-derived networks. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 388–397, 1990.

[232] C L Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computers*, 33:1247–1265, 1984.

[233] C L Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[234] C L Seitz. Concurrent architectures. In R Suaya and G Birtwistle, editors, *VLSI and Parallel Computation*, pages 1–84. Morgan Kaufmann, San Mateo, CA, 1990.

[235] C L Seitz. Multicomputers. In C A R Hoare, editor, *Developments in Concurrency and Communication*, University of Texas Year of Programming Institute on Concurrent Programming, pages 131–200. Addison-Wesley, 1990.

[236] C L Seitz, W C Athas, W J Dally, R Faucette, A S Martin, S Mattison, C D Steele, and W-K Su. *Message-Passing Concurrent Computers: Their Architecture & Programming.* Addison-Wesley, 1989.

[237] M Serna. Approximating linear programming is log-space complete for P. *Information Processing Letters*, 37(4):233–236, February 1991.

[238] M Serna and P Spirakis. The approximability of problems complete for P. In H Djidjev, editor, *Proc. International Symposium on Optimal Algorithms. LNCS Vol.401*, pages 193–204. Springer-Verlag, 1989.

[239] M Serna and P Spirakis. *The Parallel Approximability of Hard Problems.* Cambridge International Series on Parallel Computation. Cambridge University Press, Cambridge, UK, 1992. To appear.

[240] Y Shiloach and U Vishkin. An $O(log\, n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[241] A A Shvartsman. Achieving optimal CRCW PRAM fault-tolerance. *Information Processing Letters*, 39(2):59–66, 1991.

[242] A Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 20–25, 1989.

[243] B J Smith. Architecture and applications of the HEP multiprocessor computer system. In T F Tao, editor, *SPIE (Real Time Signal Processing IV)*, volume 298, pages 241–248. Society of Photo-Optical Instrumentation Engineers, August 1981.

[244] T H Spencer. More time-work tradeoffs for parallel graph algorithms. In *Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 81–93, 1991.

[245] P Spirakis. The parallel complexity of deadlock detection. *Theoretical Computer Science*, 52((1,2)):155–163, 1987.

[246] G L Steele Jr and W D Hillis. Connection Machine Lisp: Fine-grained parallel symbolic processing. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 279–297, 1986.

[247] V Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[248] R E Tarjan and U Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[249] A M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Series 2*, 42:230–265, 1936. Corrections, *ibid.*, 43 (1937), 544-546.

[250] J D Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[251] J D Ullman and A van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

[252] J D Ullman and M Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, February 1991.

[253] E Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31:507–517, 1984.

[254] E Upfal. An O(log N) deterministic packet-routing scheme. *Journal of the ACM*, 39(1):55–70, January 1992.

[255] L G Valiant. Graph-theoretic properties in computational complexity. *Journal of Computer and System Sciences*, 13:278–285, 1976.

[256] L G Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.

[257] L G Valiant. Optimality of a two-phase strategy for routing in interconnection networks. *IEEE Transactions on Computers*, c-32(9):861–863, September 1983.

[258] L G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[259] L G Valiant. General purpose parallel architectures. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity*, pages 943–971. North Holland, 1990.

[260] L G Valiant and G J Brebner. Universal schemes for parallel communication. In *Proc. 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, 1981.

[261] L G Valiant, S Skyum, S Berkowitz, and C Rackoff. Fast parallel computation of polynomials using few processors. *SIAM Journal on Computing*, 12(4):641–644, November 1983.

[262] S Vavasis. Gaussian elimination with pivoting is P-complete. *SIAM Journal of Discrete Mathematics*, 2(3):413–423, 1989.

[263] U Vishkin. Synchronous parallel computation - a survey. Technical Report 71, Courant Institute, New York University, April 1983.

[264] U Vishkin. An optimal parallel algorithm for selection. *Advances in Computing Research*, 4:79–86, 1987.

[265] U Vishkin. Can parallel algorithms enhance serial implementation? Technical Report UMIACS-TR-91-145, Institute for Advanced Computer Studies, University of Maryland, 1991.

[266] U Vishkin. Structural parallel algorithmics. In J Leach Albert, B Monien, and M Rodriguez Artalejo, editors, *Proc. 18th International Colloquium on Automata, Languages and Programming, LNCS Vol.510*, pages 363–380. Springer-Verlag, 1991.

[267] P M B Vitanyi. A modest proposal for communication costs in multicomputers. In S K Tewksbury, B W Dickinson, and S C Schwartz, editors, *Concurrent Computations - Algorithms, Architecture, and Technology*, pages 203–216. (Proc. 1987 Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation), Plenum Press, 1988.

[268] J S Vitter and R A Simons. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Transactions on Computers*, C-35(5):403–418, May 1986.

[269] J von Neumann. *First draft of a report on the EDVAC.* Moore School of Electrical Engineering, University of Pennsylvania, 30 June 1945. Contract No. W-670-ORD-4926 between the United States Army Ordnance Department and the University of Pennsylvania. Reprinted in *Papers of John von Neumann on Computing and Computer Theory*, W Aspray and A Burks, editors. Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987, 17-82.

[270] J von zur Gathen. Parallel arithmetic computations: A survey. In *Proc. Mathematical Foundations of Computer Science1986, LNCS Vol. 233*, pages 93–112. Springer-Verlag, 1986.

[271] S Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[272] I Wegener. *The Complexity of Boolean Functions.* John Wiley and Sons, 1987.

[273] M N Wegman and J L Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.

[274] C Whitby-Strevens. The transputer. In *Proc. 12th Annual International Symposium on Computer Architecture*, pages 292–300, 1985.

[275] L D Wittie. Communication structures for large networks of microcomputers. *IEEE Transactions on Computers*, C-30(4):264–273, April 1981.

[276] J C Wyllie. The complexity of parallel computations. Ph.D. Thesis, Department of Computer Science, Cornell University, 1981.

[277] A C-C Yao. Bounds on selection networks. *SIAM Journal on Computing*, 9(3):566–582, August 1980.

[278] C K Yap. What can be parallelized in computational geometry? In A Albrecht, H Jung, and K Mehlhorn, editors, *Parallel Algorithms and Architectures, LNCS Vol. 269*, pages 184–195, 1987.

[279] H Yasuura. On parallel computational complexity of unification. In *Proc. International Conference on Fifth Generation Computers*, pages 235–243, 1984.