Complexity Issues in General Purpose Parallel Computing

Douglas Andrew Chin
St. Catherine's College

Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford

Trinity Term 1991

Complexity Issues in General Purpose Parallel Computing

Abstract

In recent years, powerful theoretical techniques have been developed for supporting communication, synchronization and fault tolerance in general purpose parallel computing. The proposition of this thesis is that different techniques should be used to support different algorithms. The determining factor is *granularity,* or the extent to which an algorithm uses long blocks for communication between processors.

We consider the *Block PRAM model* of Aggarwal, Chandra and Snir, a synchronous model of parallel computation in which the processors communicate by accessing a shared memory. In the Block PRAM model, there is a time cost for each access by a processor to a block of locations in the shared memory. This feature of the model encourages the use of long blocks for communication.

In the thesis we present Block PRAM algorithms and lower bounds for specific problems on arrays, lists, expression trees, graphs, strings, binary trees and butterflies. These results introduce useful basic techniques for parallel computation in practice, and provide a classification of problems and algorithms according to their granularity. Also presented are optimal algorithms for universal hashing and skewing, which are techniques for supporting conflict-free memory access in general- and special-purpose parallel computations, respectively.

We explore the Block PRAM model as a theoretical basis for the design of scalable general purpose parallel computers. Several simulation results are presented which show the Block PRAM model to be comparable to, and competitive with, other models that have been proposed for this role. Two major advantages of machines based on the Block PRAM model is that they are able to preserve the granularity properties of individual algorithms and can efficiently incorporate a significant degree of fault tolerance.

The thesis also discusses methods for the design of algorithms that do not use synchronization. We apply these methods to define fast circuits for several fundamental Boolean functions.

For my family

Acknowledgments

During my four years in Oxford, I have been fortunate to have received excellent supervision from two very nice men. I would especially like to thank William F. McColl, my supervisor since June 1989, for the many hours he has spent introducing me to research topics, reading my work and listening to my ideas. His time and talents in all of these capacities have never failed to motivate and guide my work, and he has had the greatest single influence on the ideas expressed in this thesis (but, alas, no influence on my American spelling). I would also like to thank J. A. Dominic Welsh for his caring patience in helping me overcome my rough start in Oxford. In my long search for a qualifying thesis topic, I could not have asked for better help than his broad base of expertise and his incisive sense of quality in mathematical research.

I have enjoyed the company of a stimulating community of combinatorial mathematicians and theoretical computer scientists in Oxford. Apart from my supervisors, Colin McDiarmid and Abdon Sanchez-Arroyo have been especially generous with their time and ideas. Over the years, I have also benefited from many discussions with James Annan, Gaetan Hains, Quentin Miller, Jorge Ramirez-Alfonsin, Ken Regan, Mikkel Thorup, Dirk Vertigan and Xiaohong Xie.

Thanks to the generous support of the Mathematical Institute, the National Science Foundation, the Rhodes Trust, the University of Bonn and the University of Catania, I have been able to attend at least four conferences in each of my years in Oxford, allowing me to make many out-of-town contacts and friends. I have been especially inspired by the small but active community of British complexity theorists, including Meurig Beynon, Ben Dessau, Paul Dunne, Alan Gibbons, Leslie Henderson, Mark Jerrum, Mike Paterson, S. Ravindram, Alistair Sinclair and Uri Zwick.

I have received helpful comments on the ideas in this thesis from Andy Jones, David May, Shanmugavelayutham Muthuk, Naomi Nishimura, Ian Parberry, Greg Plaxton, Vijaya Ramachandran and Desh Ranjan. I am also grateful to Helmut Alt, Philip Gibbons, Torben Hagerup, Charles Martel, Bojana Obrenic, Jeffrey Vitter and Ofer Zajicek for responding promptly and generously to my requests for reprints.

¿From the start, I was warned that the Oxford D. Phil. course would exact a small toll on my savings and sanity. Thanks to the fellowship support of the National Science Foundation and the Rhodes Trust, I remain solvent as of this writing. My peace of mind is greatly indebted to preserving circles of friends in the John Wesley Society and the college bridge team, to Chris Wong, who will find his way back; to Assad Mehmood, who made teaching its own reward; and to George Berzsenyi, who teaches me still.

Finally, I would like to thank Andrew Newman and Geraint Jones for rescuing my files after a crash on the final weekend.

D. Andrew Chin

Oxford
July 1991

Were I to await perfection, my book would never be complete
—Chinese proverb

Contents

Chapter 1

Introduction

To a research scientist, no development can be more satisfying than the union, at long last, between theory and practice. Real-world applications can immediately demonstrate a theory's practical value and empirical truth. This thesis proclaims the union between the way we solve many problems abstractly and the way we can—and should—solve them in practice.

1.1 Theory and practice

Eventually, our most powerful problem-solving machines will be parallel computers.[1] In a parallel machine, many computations can take place simultaneously, saving time. Miniaturized hardware and mass production will provide an irresistible economy of scale in favor of parallel computing. Already, parallel computers containing more than 65,000 processors have been built and sold commercially, and we may expect even larger machines to appear before very long.

Parallel complexity theory describes the difficulty of solving problems on parallel computers. To date, most of this theory has focused on the *Parallel Random Access Machine* (PRAM) model of computation. A PRAM consists of a number of processors each with a local memory, together with a shared memory. In unit time, each processor may execute a basic operation on data in local memory, or it may read or write to a location in the shared memory. (The PRAM is formally introduced in Chapter 2.)

[1]This may already be true: see [?].

It is widely agreed that the PRAM model has significant value in the development of a theoretical basis for general purpose parallel computing. We give some reasons for this.

• *Ease of use:* The existing body of PRAM research provides many immediate applications and demonstrates the ease with which new algorithms can be designed and specified. A popular, user-friendly model of computation is essential to stimulate the widespread development of parallel software.

• *Portability:* Real parallel computers, or *multiprocessors,* consist of processors connected in a network. The topology, or architecture, of that network is ignored in the description of a PRAM. Architecture independence allows the performance of parallel computers to be discussed despite the present lack of consensus about designing interprocessor networks. PRAM algorithms do not need to take into account memory organization, network topology or other hardware design attributes of real parallel computers. Architecture independence is essential if parallel software is to be *portable,* or broadly applicable, across a range of computer designs and technological developments. See [135, 189] for a detailed discussion of these issues.

• *Scalability:* PRAM algorithms are designed to solve problems, which can occur as problem instances of varying size. Typically the amount of *parallelism,* the number of parallel processors which can be used to speed up a PRAM algorithm, increases as the problem size increases. The power of a PRAM to solve the problem quickly is then said to *scale* upwards with the number of processors. A PRAM continues to match the computation and communication behavior of the algorithm, time step for time step, even when arbitrarily many processors are used. This ideal performance makes the PRAM model a standard for evaluating the *scalability* of real parallel architectures [149]. The design of scalable software, such as PRAM algorithms, encourages the development of parallel machines with more processors, so that bigger problems can be solved quickly.

For specific problems, theoretical computer scientists have designed *algorithms,* or procedures for finding solutions, designed to run on the PRAM model. They also have proved *lower bounds,* demonstrating that to solve problems of a given size requires a minimum length of time and/or a minimum number of processors. These results provide abstract solutions to problems and a means for evaluating them, thereby contributing to both computer science and pure mathematics. (For a survey, see [11, 75, 104, 154].) Until recently, however, unrealistic characteristics of the PRAM model have limited the value of this research program.

• PRAM processors can read and write into locations of a shared memory without requiring extra time, or *communication latency,* to transfer data through an interconnection network.

• PRAM processors can have unrestricted patterns of access to a shared memory without causing competition, or *contention,* for limited memory resources, and without causing local accumulations of traffic, or *congestion,* in an interconnection network.

• The operations on each PRAM processor are performed in unison, or *synchronously.*

• PRAM processors compute and communicate without errors, or *failures.*

The PRAM is an ideal parallel complexity model. For the foreseeable future, we will be unable to build a real parallel machine having any of these qualities. Consequently we cannot expect the PRAM complexity results to describe the performance of algorithms in practice.

1.2 Thesis

Needing to support an ideal model in the real world, we can treat the unrealistic characteristics of the PRAM in two complementary ways. In practical work, we can design machines which simulate the ideal conditions required by the PRAM model. In theoretical work, we can implement and simulate PRAM algorithms on alternative architecture-independent parallel complexity models which more accurately describe real machines.

By taking this theoretical route, this thesis enters an active discussion. Since 1987, at least 10 "realistic" parallel complexity models have been introduced, and many more variants of these models have been proposed. Each model accounts for some difficulties in parallel computing but not others, leaving open questions about the impact of unexamined factors.

The goal of this thesis is to bring unity and coherence to the discussion of practical issues in parallel computation. We study a single architecture-independent complexity model, the *Block PRAM,* which has very general application to the major practical issues in parallel computation described above. This approach enables us to exhibit a rich correspondence between practical issues, complexity models and algorithmic structures in parallel computation. Many issues are related quantitatively for the first time in the complexity theory literature, and many optimal bounds are proved. The final product is a fourfold prescription for parallel computation, described in terms of the complexity of parallel algorithms.

- General simulation results show how PRAM algorithms can be implemented efficiently in practice.
- Algorithms and lower bounds for specific problems indicate when it is possible to improve upon the performance of the general simulation.
- Complexity tradeoffs between these general and specific approaches lead to designs with which we can simultaneously achieve most of the benefits of both approaches.
- Opportunities for improving performance and reliability identify important primitive operations which should be supported in hardware or low-level software.

A further important (if esoteric) achievement of this thesis is the rehabilitation of the Block PRAM as a theoretical model of parallel complexity. Since its first appearance in [5], the Block PRAM model has been essentially abandoned by the research community, despite the rapidly growing interest in "practical" and "realistic" PRAM models. We identify the model's failure to account for network congestion as a key reason for its premature demise. We then address this problem by showing that to a great extent, the effect of network congestion on the Block PRAM model can be either minimized (probabilistically) or modeled (deterministically). Moreover, we demonstrate advantages to using the Block PRAM model as a basis for general purpose parallel computation. Based on the results in this thesis, we are able to conclude that algorithms designed for the Block PRAM model will have a significant impact on the design of portable and scalable software for general purpose parallel computers.

1.3 Overview

The remainder of this thesis is organized as follows.

• Chapter 2 gives a formal introduction to the PRAM model and its resulting complexity theory.

• Chapter 3 reviews the main realistic obstacles to general purpose parallel computation that have been recognized to date, and some of the theoretical solutions which have been advanced to cope with them.

• Chapter 4 gives a detailed and critical description of the Block PRAM model, which is the main model studied in this thesis.

• Chapter 5 gives Block PRAM algorithms and lower bounds for specific problems. The parallel complexities of the two most fundamental PRAM algorithms, prefix sums computation and list ranking, are separated for the first time, justifying a claim of Gazit, Miller and Teng [71]. New optimal Block PRAM algorithms are given for prefix sums computation and string matching. New Block PRAM lower bounds are proved for list ranking and tree contraction. Existing Block PRAM algorithms for permuting data and performing the Fast Fourier Transform are generalized. Other new Block PRAM algorithms for integer sorting and graph connectivity are also given.

• Chapter 6 proves tight bounds on the Block PRAM complexity of hashing and skewing, two techniques which are used to allow for memory accesses during parallel computation. By showing that hashing does not necessarily destroy locality, these results address open questions of Valiant [189] and Gibbons [76] and contradict a comment of Heywood and Ranka [89].

• Chapter 7 discusses the applicability of the Block PRAM model and complexity theory to real parallel computers. The notion of locality-preserving hash functions is introduced, which allows Block PRAM algorithms to be implemented in real parallel computers which use hashing. The chapter also discusses some relationships between the Block PRAM model and multitasking and multigauging architectures.

• Chapter 8 surveys relationships between a variant of the Block PRAM model, the arbitrary-Block PRAM, and two asynchronous parallel complexity models, the XPRAM [190] and the EREW Phase LPRAM [76]. Several new simulation results are proved, showing that the models are pairwise comparable in power. It is shown for oblivious computations that the XPRAM and EREW Phase LPRAM models are equivalent, and a connection is drawn between architectural support for arbitrary pipelining and support for pointer chasing, thereby addressing a concern of Miller [142].

- Chapter 9 discusses methods for the design and analysis of efficient asynchronous algorithms, and applies these methods to derive faster circuits for several fundamental Boolean functions.
- Chapter 10 shows how EREW PRAM and Block PRAM algorithms can run correctly even when there are occasional failures in interprocessor communication. A variant of the Block PRAM model, the Faulty Block PRAM, is introduced. Using this model, tight simultaneous bounds on complexity and fault tolerance are proved for list ranking and tree contraction, and simulation results are given for EREW PRAM and certain Block PRAM computations.
- Chapter 11 gives conclusions and open questions.

1.4 Notations

Throughout this thesis, logarithms are base 2. **N** denotes the set of positive integers, **Z** the set of integers, and **R** the set of real numbers.

We use the "big-oh" notation to describe the asymptotic growth of nonnegative real-valued functions f, g as $n \to +\infty$:

- $f(n) = O(g(n))$ if there is a constant $c > 0$ such that $0 \leq \lim f(n)/g(n) \leq c$.
- $f(n) = \Omega(g(n))$ if there is a constant $c > 0$ such that $\lim f(n)/g(n) \geq c$.
- $f(n) = \Theta(g(n))$ if there are constants $c_1, c_2 \geq 0$ such that $c_1 \leq \lim f(n)/g(n) \leq c_2$.
- $f(n) = o(g(n))$ if $\lim f(n)/g(n) = 0$.
- $f(n) = \omega(g(n))$ if $\lim f(n)/g(n) = +\infty$.

If $f(n) = n^{O(1)}$ we say $f(n)$ is *polynomial* in n; if $f(n) = \log^{O(1)} n$ we say $f(n)$ is *polylogarithmic* in n; if $f(n) = O(1)$ we say $f(n)$ is *constant.* Unless otherwise specified, a complexity bound is *optimal* if it can be improved only by a constant factor.

For integers a, n with $n > 0$, by a mod n we will mean the unique integer in the range $[0 \ldots n-1]$ which is congruent to a modulo n.

For real $x > 1$, by $\log^* n$ we will mean the least integer k such that $\log^{(k)} n \leq 1$, where $\log^{(k)}$ is the logarithm function iterated k times.

Throughout this thesis, we will use standard terms from *graph theory.* A *graph* is a mathematical structure representing a set of objects together with connections between pairs of those objects. Informally, a graph is any kind of network. Graph theory is one of the richest and most rapidly growing fields in pure mathematics. For a survey of graph theory, see e.g. [33, 85]. For a survey of sequential graph algorithms, see e.g. [61]. For a survey of parallel graph algorithms, see e.g. [104].

Chapter 2

The PRAM model

Abstract models describing general classes of computing machines are defined in order to facilitate the development of coherent theories of computational complexity. Algorithms designed for these models can then be viewed as progress towards solving problems in practice, and lower bounds proved for these models represent fundamental limits to such progress. The interpretation of complexity-theoretic results for computations on a given machine depends on the extent to which the specific characteristics of the machine have been ignored by the abstract model.

The PRAM model has played a prominent role in research on parallel algorithms. By using a powerful, idealized model of computation, algorithm designers can focus on the logical, algorithmic and mathematical issues of solving problems in parallel. This can make complexity results easier to obtain, and it is not surprising that results for more realistic models of parallel computation are often based on corresponding results for the PRAM. This chapter provides a brief introduction to the PRAM model of parallel computation and its complexity theory.

2.1 Definition of the model

We formally describe a *Parallel Random Access Machine* (PRAM) as a set of p processors sharing a common memory of unbounded size. (The number p is the *parameter* for the model.) Each processor is a unit-cost RAM [7] with a restricted arithmetic instruction set [154] (see Section 4.2.8). The instruction set includes indirect loads and stores to the shared memory. The word size is $\Theta(\log p)$.

The processors are assigned different identifying numbers $1, \ldots, p$. Each processor has a local memory of unbounded size, in which is stored its identifying number and its program. All processors synchronously execute the same program, performing one instruction in each time step. Depending on their identifying numbers or other local conditions, some processors may take part in an instruction, while the other processors remain idle. (That is, the processors may branch, but the branches must be executed sequentially.) The inputs to the program reside initially in shared memory, and the outputs are also stored there.

Variants of the PRAM model are defined according to whether several processors can access the same memory location in the same time step. The possible protocols include Exclusive Read, Exclusive Write (EREW), Concurrent Read, Exclusive Write (CREW), and Concurrent Read, Concurrent Write (CRCW). In the CRCW case, various rules for resolving write conflicts may be specified:

COLLISION: A special collision symbol is written into the contested location.

COMMON: All processors writing to the contested location must be writing the same value.

ARBITRARY: Nondeterministically, one of the processors writing to the contested location succeeds.

PRIORITY: The lowest-numbered processor writing to the contested location succeeds.

Our definition differs from standard descriptions of the PRAM in the literature [75, 104] in two respects: the (asymptotically) fixed logarithmic word size, and the restriction of the arithmetic operations included in the instruction set. These specifications have not always been necessary in the PRAM theory, but they are essential in order to develop a rigorous and meaningful Block PRAM theory, as we discuss in Chapter 4. Since the Block PRAM model is based on the EREW PRAM, we have included these specifications in our definition of the PRAM model.

2.2 Speedup and efficiency

As with sequential complexity, the time complexity of a parallel algorithm is given in terms of its worst-case running time for problem instances of size n, together with the parameters of the complexity model. For example, for a given PRAM algorithm $\mathcal{A}$, the parallel time complexity $t^{\mathcal{A}}_{\mathrm{PRAM}}$ is a function of the problem size n and the number of processors p. (In this section, where the algorithm and problem we refer to is understood, the superscript $\mathcal{A}$ is omitted for clarity.)

For a given problem, let t_{RAM} represent the best known upper bound on the sequential complexity of a problem of size n. The *speedup* of an algorithm for a parallel complexity model M is $t_{\mathrm{RAM}}(n)/t_{\mathrm{M}}(n)$, a function of n and the parameters for M. For example, the speedup of a PRAM algorithm is $t_{\mathrm{RAM}}(n)/t_{\mathrm{PRAM}}(n,p)$.

The *work* w used by a parallel algorithm is the parallel time complexity multiplied by the number of processors. Since p parallel operations can be simulated in p steps on a single processor, $t_{\mathrm{RAM}}(n) \leq w_{\mathrm{M}}(n)$ for any parallel complexity model M and for any choice of parameters for M. For example, $t_{\mathrm{RAM}} \leq w_{\mathrm{PRAM}}(n,p) = p \cdot t_{\mathrm{PRAM}}(n,p)$ for all $n, p \in \mathbf{N}$.

A fundamental result of Brent [35] establishes that in a synchronous parallel model of computation such as the PRAM, an algorithm using q processors can be simulated by $p \leq q$ processors, conserving work.

Theorem 2.1. [35]

Any synchronous parallel algorithm $\mathcal{A}$ running in time t that consists of a total of x basic instructions can be implemented by p processors in time $O(x/p + t)$.

Proof.

Let x_i be the number of instructions performed in the i-th step of $\mathcal{A}$. Using p processors, this step can be simulated in $O(x_i/p + 1)$ time. Hence $\mathcal{A}$ can be simulated in $\sum_{i=1}^{t}(x_i/p + 1) = O((\sum_{i=1}^{t} x_i)/p + t) = O(x/p + t)$.$\square$

Brent's Theorem allows us to use the quantity of work in expressing asymptotic complexity bounds throughout this thesis. For example, an algorithm which runs in $O(\log n)$ time on $n/\log n$ processors will be said to run in $O(n/p + \log n)$ time.

Work as a complexity measure has practical importance, as typically many applications share the computational resources of a parallel computer. The power of a parallel computer can be expressed as the amount of work it can do in a given amount of time. A parallel algorithm for model M uses this power *efficiently* (within a constant factor) if $w_{\mathrm{M}}(n)/t_{\mathrm{RAM}}(n) = O(1)$ for some range of parameters for M. For example, if $t_{\mathrm{PRAM}}(n,p) = O(t_{\mathrm{RAM}}(n)/p + f(n))$, then the algorithm is efficient for $p = O(t_{\mathrm{RAM}}(n)/f(n))$.

Speedup and efficiency, then, are central objectives in the design of parallel algorithms. As Kruskal, Rudolph and Snir have emphasized recently in [113], the search for fast, efficient parallel algorithms constitutes the basis of parallel complexity theory.

2.3 Open problems

Research in parallel complexity theory is concerned with the question of which problems can be solved efficiently using massive parallelism. We mention here two of the most important and challenging open areas of study relating to PRAM algorithms.

$\mathcal{P} =? \mathcal{NC}$: A problem of size n is in the class $\mathcal{P}$ if it can be solved by a RAM algorithm in $n^{O(1)}$ time, and in the class $\mathcal{NC}$ if it can be solved by a PRAM algorithm in $n^{O(1)}/p + \log^{O(1)} n$ time. Clearly $\mathcal{NC} \subseteq \mathcal{P}$; it is unknown whether $\mathcal{P} = \mathcal{NC}$. The classes $\mathcal{P}$ and $\mathcal{NC}$ are robust across a large number of sequential and parallel models of computation, respectively (see e.g. [60]), and $\mathcal{P} =? \mathcal{NC}$ is a fundamental structural question in computational complexity theory. There is a large class of *$\mathcal{P}$-complete* problems in $\mathcal{P}$, such that an $\mathcal{NC}$ algorithm for any one of them implies $\mathcal{P} = \mathcal{NC}$. There are also many interesting problems known to be in $\mathcal{P}$ but not known to be $\mathcal{P}$-complete or in $\mathcal{NC}$, including computing the greatest common divisor of two integers, finding a perfect matching of a graph, and estimating all of the (real and complex) roots of an n-th degree polynomial. For a review of current research on the $\mathcal{P} =? \mathcal{NC}$ question, see e.g. [79, 80, 144].

The transitive closure bottleneck: Many problems on directed graphs can be reduced to computing the *transitive closure* $X^* = \oplus_{k=0}^{\infty} X^k$ of a square matrix X. Let $(S, \oplus, \otimes, I_\oplus, I_\otimes)$ be a closed semiring ($\oplus$ is closed, associative

Problem	S	$\oplus$	$\otimes$	$I_\oplus$	$I_\otimes$
Strong components	$\{0,1\}$	$\vee$	$\wedge$	0	1
Finding a cycle	$\{0,1\}$	$\vee$	$\wedge$	0	1
Minimum cost spanning tree	$\mathbf{R}^+ \cup \infty$	min	max	∞	0
Topological sorting	$\mathbf{R} \cup \infty$	max	$+$	$-\infty$	0
Shortest paths	$\mathbf{R} \cup \infty$	min	$+$	∞	0
Longest paths	$\mathbf{R} \cup \infty$	max	$+$	$-\infty$	0
Maximum reliability paths	$[0,1]$	max	$\times$	0	1

Table 2.1: Directed graph problems reducible to transitive closure

and commutative; $\otimes$ is closed, associative and right- and left-distributive over $\oplus$; $I_\oplus$ is a zero with respect to $\otimes$). Let $G = (V, A)$ be a directed graph with $V = \{1, \ldots, n\}$, and each of the m edges $(i,j) \in A$ has a label $x_{i,j} \in S$. Define $x_{ij} = I_\oplus$ for $(i,j) \in V^2 \backslash A$. With appropriate choices for $\oplus$ and $\otimes$, each of the problems in Table 2.1 can be solved by the transitive closure computation.

Unfortunately, this technique does not lead to any practical, efficient parallel algorithms. Each of the first four problems in Table 2.1 can be solved sequentially in time either $O(m+n)$ and the last three can be solved in time $O((m+n)\log n)$. However, the best known upper bound on the complexity of performing transitive closure using a polynomial number of processors is $O(M(n)/p + \log^2 n)$, where $M(n) = O(n^{2.376})$ is the sequential complexity of matrix multiplication. Because the only known NC algorithms for these problems involve transitive closure as a subroutine, this gap in complexity is known as the *transitive closure bottleneck* [104]. This bottleneck is particularly troublesome for computations on sparse, irregular graphs, where $m = o(n^2)$, since the transitive closure algorithm requires an input of size $\Theta(n^2)$. Some progress towards overcoming the bottleneck for planar graphs has been reported recently in [100].

Chapter 3

Realistic issues

Computers are not fast enough. The clearest evidence for this is the fact that humans can still solve many problems much more rapidly than our most powerful computers. For example, human capabilities for real-time visual and auditory processing remain far beyond the frontiers of supercomputing. "What the human mind does almost effortlessly," writes Hillis [91], "would take the fastest existing computers many days. These electronic giants that so outmatch us in adding columns of numbers are equally outmatched by us in the processes of symbolic thought."

Demands for greater computational performance are coming from an economy where information is increasingly a source of employment, capital and production, from a research community where data is collected more rapidly than it can be processed, and from a design community where models and prototypes are too complex for human intuition and reasoning.[1] High-performance computing has applications throughout the natural and social sciences, and in medicine, engineering, business and communication. Massive computational power will be necessary to achieve many of the aspirations of humanity, from exploring space and managing the earth's resources, to mapping the human genome and modeling molecules.

Large-scale parallel computers will bear the final responsibility for improving computational performance. Since von Neumann's post-World War II computer designs [200], the technological advances which have provided

[1]Whether or not we welcome these developments, we can hope that computer science will aim to offer solutions that will benefit humanity.

faster sequential processors have been increasingly difficult to achieve and are ultimately limited by the speed of light. The speed of a sequential processor limits, in turn, the extent to which a large memory can be utilized efficiently. We are, therefore, approaching a time when it is no longer cost-effective to build bigger sequential computers to achieve high performance. The inherent difficulty of efficiently improving the performance of sequential computers is known as the *von Neumann bottleneck* [25].

Parallel computation offers an alternative approach to achieving high performance. Instead of building one big sequential computer, we can build a machine consisting of many small computers running in parallel. Provided that we can use the computing power of such a machine efficiently, we have overcome the von Neumann bottleneck.

The purpose of parallel complexity theory is to explore the extent to which efficient and fast parallel computation is possible. By showing that many important computational problems can be solved quickly and efficiently in an ideal parallel complexity model, the PRAM theory represents important progress toward high-performance computing. Before we can apply this theory, however, the gap between the ideal PRAM and the realities of parallel computation must be bridged.

3.1 Bridging models for parallel computation

In recent years, the complexity theory research community has made substantial progress towards demonstrating that there are no theoretical obstacles to building massive general purpose parallel computers. The gap between real parallel architectures and the ideal PRAM model has been bridged in two discernible stages.

From realistic machines to realistic shared-memory models: A realistic *distributed memory* machine, consisting of a set of processor-memory elements connected by a physical network, can simulate shared memory in the following way. The aggregate of the local memory elements is taken to be the shared memory. To write into a shared memory location, a processor sends the data to the location's host processor through the network. To read a shared memory location, a processor sends a request to the location's host processor, which sends back a reply.

The research community has responded to the practical difficulties posed by this simulation by identifying key problem areas (see Section 3.2) and developing many promising techniques (see Section 3.3). Despite the achievements of this research program, it should be kept in mind that simulation techniques are methods for coping with reality, not changing reality. In particular, at present we must continue to view the PRAM as an unrealistic model of parallel computation.

• *From realistic shared-memory models to the ideal PRAM:* These developments have led recently to the introduction of *realistic shared-memory models.* Since 1989, models have been developed which take into account communication latency [5, 6], asynchrony [52, 76, 138, 148, 190] and/or component failures [43, 99, 106]. Such models are necessarily weaker than the PRAM model. However, most of the new models have been introduced together with *simulation results,* demonstrating the extent to which PRAM algorithms can be implemented with little or no asymptotic loss in efficiency (see Chapter 8).

This proliferation of realistic shared-memory models can be largely justified by their potential role in bridging the gap between hardware and software. If a model of parallel computation (a) is realizable in hardware (e.g., using the distributed-memory simulation described above) and (b) allows efficient simulations of PRAM software, then it can be used as a *bridging model.* Computer manufacturers can produce machine realizations of the bridging model with different sizes and technologies, while software designers can write algorithms based on the PRAM model which will run efficiently on any such machine. As Valiant writes in his paper introducing the idea of a bridging model for parallel computation [189]: "A major purpose of such a model is simply to act as a standard on which people can agree...The adoption of such a standard can be expected to insulate software and hardware development from one another and make possible both general purpose machines and transportable software."

It is likely that the adoption of a bridging model for parallel computation will have long-term implications for computer manufacture, and so it is important to make the right choice. In [189], Valiant proposed the Bulk Synchronous Parallel (BSP) model of computation (see Chapter 8) as such a bridging model, and programming interfaces have been developed for future

Problem	**Solution**	
Defined	*Described*	*Evaluated*
Communication latency (3.2.1)	Pipelining (3.3.1) Parallel slackness (3.3.1)	Chapter 5 Chapter 7
Contention (3.2.2)	Hashing (3.3.2) Skewing (3.3.2) Combining (3.3.2)	Chapter 6 Chapter 6
Congestion (3.2.2)	Randomized routing (3.3.3) Information dispersal (3.3.5)	Chapter 7 Chapter 10
Asynchrony (3.2.3)	Bulk synchrony (3.3.4) Parallel slackness (3.3.1) Robust computation (3.3.6)	Chapter 8 Chapter 8
Network failure (3.2.4)	Information dispersal (3.3.5) Parallel slackness (3.3.1)	Chapter 10 Chapter 10
Processor failure (3.2.4)	Robust computation (3.3.6)	

Table 3.1: Cross-reference: problems and solutions

machines based on the BSP model [14]. I n Chapter 7, we recommend the Block PRAM model as a valid alternative, with important consequences for both hardware and software design.

3.2 Realistic constraints to parallel computation

We turn now to a survey of the difficulties involved in simulating shared memory on massively parallel distributed-memory architectures. Table 3.1 provides a cross-reference between the problems described in this section, the simulation techniques described in Section 3.3, and the chapters in the thesis, if any, in which they are discussed in the context of the Block PRAM model.

3.2.1 Communication latency

Distinct processors working together on the same problem will need to communicate from time to time. In present-day machines, this communication must take place through a physical network and is subject to considerable delay. "It is becoming abundantly clear," write Aggarwal, Chandra and Snir [6], "that much of the complexity in parallel computing is due to the difficulty in communication itself." Although optical networks have been studied extensively as a long-term prospect [137], an access to a non-local memory location will continue to take longer—and be more difficult to control—than a local computation.

The topology of the network connecting the processors is a basic design choice. The central tradeoff is between *communication latency,* the transit time that must be allowed for a single message, and *connectivity,* the number of wires and ports represented by the network. In graph-theoretic terms, we may speak of the tradeoff between diameter and maximum degree. A signal may have to traverse a path as long as the diameter of the graph, passing through routing hardware at each vertex or *node*; the diameter is, therefore, an asymptotic lower bound on communication latency in the network. Each wire between processors, or *link,* must be represented by an edge in the graph; the maximum degree is, therefore, an asymptotic lower bound on the number of ports that must be available to at least one of the processors.

We give some important examples in Table 3.2. For illustration, we describe the networks for approximately $p = 2048$ processors.

If the network is to be used for general problem solving, another graph property becomes important: *vertex-transitivity.* A graph is vertex-transitive if for every pair of vertices v, w, there is an automorphism σ on the graph such that $\sigma(v) = w$. Informally, a vertex-transitive graph looks the same when viewed from any of its vertices. The star and binary tree graphs are not vertex-transitive, and they are bad choices for a general purpose network, because too much traffic goes through the root, causing *congestion* (see Section 3.2.2). *Randomized routing* techniques for keeping network traffic uniform exist for the other vertex-transitive networks in the Table 3.2 (see Section 3.3.3).

Ultimately, any network of processors must be constructed in a three-dimensional world, and the physical diameter of any network of p processors

[2] $p = 2047$
[3] $p = 2197$; with wraparound
[4] $p = 2025$; with wraparound

Graph	Diameter		Max degree	
	p processors	$p = 2048$	$p = p$	$p = 2048$
Complete graph	1	1	$p-1$	2047
Star	2	2	$p-1$	2047
Hypercube	$\log p$	11	$\log p$	11
Butterfly	$2\log p - O(\log\log p)$	16	4	4
Shuffle-exchange	$2\log p$	22	3	3
Binary tree[2]	$2\log(p+1) - 2$	20	3	3
Cube-conn. cycles	$2.5\log p - O(\log\log p)$	20	3	3
Cube mesh[3]	$1.5p^{1/3}$	18	6	6
Square mesh[4]	$p^{1/2}$	44	4	4
Ring	$p/2$	1024	2	2

Table 3.2: Graph properties of interprocessor networks

must be $\Omega(p^{1/3})$. Any network based on a graph with a diameter of $o(p^{1/3})$ must therefore contain some long wires. Currently, routing of signals at the nodes of an interprocessor network takes much longer than propagation of signals along the wires. For this reason, we tend to speak of $O(\log p)$ latency for the hypercube-type networks currently being built, even though this cannot be justified when the number of processors grows without limit.

We have discussed the problem of communication latency in some detail because we believe it represents the most fundamental practical obstacle to large-scale computing. If communication latency did not exist, each of the other obstacles described in this section would be considerably simpler, if not trivial.

Communication has been long recognized as a key obstacle to parallel computation [131]. The complexity of interprocessor communication in parallel computing has been studied for more than a decade.

• Two-processor communication complexity, as formalized by Yao [207], has been used as a tool for proving complexity bounds for distributed computer networks, VLSI circuits, Boolean circuits (see Section 9.2) and test trees, and as a structural measure in its own right; the various topics are surveyed in [39, 120, 132].

- Sequential computations with parallel accesses to disks and other external secondary storage devices, with corresponding communication delays, have been studied in [197, 198, 199], extending work in [2, 4].
- Computations on data-dependency graphs have been mapped onto specific networks of processors [62, 73, 81, 109, 123, 163] and general sets of communicating processors [6, 151, 152] to minimize overall computation and communication time.
- Communication latency has been introduced as a parameter in shared-memory models of parallel computation [5, 6, 76, 152]. Algorithms have been designed for these models, and techniques have been developed for simulations of PRAMs and computations on data-dependency graphs.

Since we wish to study communication latency as an obstacle to realizing the PRAM theory, our work contributes to the last of these research areas. In Section 3.3.1, we describe proposed methods for coping with latency. In Chapter 5, we apply and evaluate these methods by showing how PRAM algorithms can best be implemented to run quickly and efficiently on the Block PRAM, a particular shared-memory complexity model which accounts for latency.

3.2.2 Contention and congestion

In current large-scale parallel computers, the memory resource is contained in *modules,* or *banks.* Each module contains many memory locations, and each is capable of serving only a constant number of requests during each time step. The modules are connected to the processors by way of a physical network, and each link in the network is capable of transmitting only a constant amount of data during each time step. (For example, the machine may consist of a network of processor-memory units, or *processing elements.*)

Contention results when the accesses during a time step require a module to serve too many requests, and *congestion* occurs when a link or node is required to carry too much data. Excessive accesses and transmissions must be *serialized,* or staggered to take place over several time steps. Unless contention and congestion are managed, they may well dominate the actual running time of a theoretically efficient algorithm [159].

As explained by Kuck in a groundbreaking 1977 paper [116], contention is especially likely to occur when there are a number of different but systematic patterns of access to shared memory, e.g. in multidimensional arrays. Congestion has been observed in on-line deterministic routing of certain "hard" permutations on the torus and butterfly networks, e.g. the bit-reversal permutation [130, 167]. For a greedy routing strategy on the $n \times n$ torus, tight lower bounds on congestion have been proved for specially constructed "blocking" permutations [88].

(It should be noted that congestion is not an inherent obstacle to deterministic routing of permutations on static interconnection networks, since routing can be performed by specialized sorting networks such as the Ajtai-Komlós-Szemerédi network [10]. However, practical sorting networks have not yet been developed for this purpose.)

The PRAM model does not restrict memory access patterns; this is especially true of the CREW and CRCW variants. Contention and congestion, therefore, remain significant obstacles to realizing the PRAM model. *Bandwidth,* the capacity of a memory module for inputting and outputting data and the capacity of a link for carrying data, can be increased through improved technology. However, these improvements are soon exhausted, and inevitably we must find ways of smoothing out demands on the memory modules and traffic in the network. Techniques for overcoming contention are described in Section 3.3.2 and evaluated in Chapter 6, and methods for overcoming congestion are described in Section 3.3.3 and evaluated in Chapters 7 and 10.

3.2.3 Asynchrony

It is difficult, and sometimes impossible, to predict accurately the time required to execute a process in a parallel algorithm. Evidence of this difficulty is the fact that experimentation, benchmarking and fine-tuning are the predominant activities of researchers concerned with implementing large-scale parallel applications. This situation is unlikely to change in the foreseeable future. It has long been observed in parallel computation [118, 204] that machine specifications provide very limited information about machine behavior in practice. Intricate characteristics of the input data and the real-time activities of the operating system can cause great variations in processing speed. The machine specifications themselves may be subjected to complicated kinds of alteration, as components are replaced and upgraded.

Asynchrony is the unpredictable variation of processor speeds in parallel computers. In particular, asynchrony means that we do not know the speeds of processors relative to one another. This necessarily makes communication more difficult. In traditional specifications of parallel algorithms [94], communication between two processors occurs when one processor performs an action (e.g. writes into a shared memory location, sends a message) that the other later detects (e.g. reads the shared memory location, receives the message): the first event must occur before the second. To guarantee this ordering of events, there must be an *interaction point*[5] in real time, made known to both processors, which occurs after the first event but before the second [118]. This entails some *synchronization* control mechanism; see [76] for a review of hardware and operating system support for providing interprocessor synchronization. An interaction point common to all processors is called a *synchronization barrier* [76, 189].

This is not to say that communication is impossible without synchronization. If, at runtime, communication events happen to occur in the correct order even without synchronization, then certainly the communications have taken place and the computation is valid. However, synchronization is necessary in the specification of an algorithm to guarantee—through the ordering of communication events—that the specified communications always take place. Algorithms so specified are called *semi-synchronous.*[6] A significant advantage of semi-synchronous algorithms from the standpoint of debugging is that such algorithms are *repeatable*:[7] for a given input, every execution gives the same intermediate and final results. Semi-synchronous algorithms are surveyed in [76, 118, 189].

Algorithms that can be specified without synchronization are called *asynchronous.* In asynchronous algorithms, there is no explicit dependency between processes. Processors never wait for data, but instead proceed with their computations based on the data available to them at the time. Asynchronous algorithms are surveyed in [118, 138, 148, 208]; also see Chapter 9. Since communication events do not take place in any determined order, such algorithms are in general not repeatable. For a given input, the only result

[5]Also called "condition synchronization" [21] and "activity synchronization" [72].
[6]Or "synchronized" [118].
[7]Or "determinate" [102].

Semi-synchronous algorithms	**Asynchronous algorithms**
Use interaction points	No interaction points
Synchronization supported in hardware and operating system	No support for synchronization
Deterministic (repeatable results)	Nondeterministic
Easier to debug and analyze	Potentially more efficient

Table 3.3: Semi-synchronous vs. asynchronous algorithms

guaranteed to recur in every execution is the final one. This can make debugging very difficult [76]. Asynchronous algorithms also seem to be more difficult to analyze for complexity than semi-synchronous algorithms [52]. Asynchronous algorithms are potentially more efficient because no time is spent waiting for data. However, this time savings is soon surrendered if we need to implement synchronization algorithmically; i.e., through specially designated shared variables. These points for comparison are summarized in Table 3.3.

It should now be clear that the PRAM model does not adequately specify algorithms in the presence of asynchrony. PRAM algorithms give an explicit order of execution for all instructions, including accesses to the shared memory. A naive approach to implementing a PRAM algorithm on an asynchronous machine would be to guarantee this order by placing a synchronization barrier after every time step. However, taking the line of argument of Section 3.2.1, we cannot expect to implement a synchronization barrier in unit time: a synchronization barrier requires a considerable amount of interprocessor communication, and communication in real machines is subject to latency. More promising techniques for coping with asynchrony are described in Sections 3.3.4 and 3.3.6 and evaluated in Chapters 8.

3.2.4 Network and processor failures

It is a fundamental principle of engineering that the larger and more complex a system is, the more likely it is to fail. This follows from the reasonable assumption that each component of a system has some positive constant probability of failure. In a parallel computer with many thousands of processing elements and communication paths, the possibility of failure is a serious obstacle to achieving the correctness and complexity specifications of any given parallel program. Clearly, we should not proceed with designing parallel algorithms and programs under the assumption that improvements in technology will eliminate the threat of failure.

Many kinds of failure can affect the performance of a multiprocessing network. *Network failures* are disruptions in the transmission of data along communication paths in the network. A communication path consists of links (e.g. wires) and nodes (e.g. switches). *Processor failures* are disruptions in the local execution of computations.

- *Link (node) interruption* occurs when a link (node) ceases to transmit data.
- *Link (node) corruption* occurs when a link (node) transmits incorrect data.
- *Processor interruption* occurs when a processor ceases to operate.
- *Processor corruption* occurs when a processor operates incorrectly.

Each of these failures can be *randomly-distributed* (each component fails with the same independent probability p) or *worst-case* (components can be failed by an omniscient adversary trying to disrupt the computation). Very pessimistic lower bounds have been proved for the problem of tolerating worst-case failures [28].

Further, these failures can be restricted to *transient* (occurring for only a few time steps) and *localized* (occurring only in a small part of the network) cases, but more general failures are possible.

The PRAM model does not account for the possibility of failures. To support the PRAM model in the real world, we must find ways to execute PRAM algorithms correctly and efficiently in the presence of failures. Methods for tolerating network failures are described in Section 3.3.5, and methods for tolerating processor failures are described in Section 3.3.6. Chapter 10 presents a scheme, based on the Block PRAM model, for tolerating transient, rando mly-distributed network corruption.

3.3 Techniques for simulating shared memory

3.3.1 Pipelining and parallel slackness

As we have seen in Section 3.2.1, communication through a multiprocessing network may be subject to a delay, or latency, proportional to the diameter of the network. Let l denote the latency of a network, as measured by the time steps of a given processor. (Recall that l is a function of the number of processors p.) Then an access by that processor to a single non-local memory location can, in general, take $\Theta(l)$ time steps to complete.

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

Currently, it is debatable which sequences of memory accesses by a processor can be pipelined. There is no theoretical evidence to suggest that any restrictions are necessary. Accordingly, Gibbons's Asynchronous PRAM model [76] allows pipelining of arbitrary sequences of accesses to the shared memory. There is as yet no practical evidence that *arbitrary pipelining* can be supported (i.e., with small constant factors) [76, 189]. Further evidence that arbitrary pipelining cannot be supported by current parallel computers is the observed difficulty of implementing pointer-based algorithms [142]; see Section 8.2.3.

We should note that this situation may change as "fine-grained" parallel architectures are introduced, with support for arbitrary pipelining as a stated objective [19]. Currently, however, more attention is being given to reducing the communication latency l rather than providing for arbitrary pipelining *per se* [57].

A more conservative alternative is offered by *block pipelining*. In block pipelining, only contiguous blocks of memory are pipelined. For several years, block pipelining has been achieved and exploited in real multiprocessor networks ([24, 36, 38, 68, 158, 171], cited in [5]). As Aggarwal, Chandra and Snir observe [5]: "Typically, it takes a substantial period of time to get the first word from global memory, but after that, subsequent words can be obtained quite rapidly—essentially at the clock speed of the machine...The size of a block that is transferred is typically correlated with communication latency."

As we will see in this thesis, this assumption is still strong enough to allow efficient *simulations* and implementations of PRAM algorithms (even when asynchrony is introduced). An efficient PRAM simulation is a procedure by which any PRAM algorithm on q processors can be executed on a more realistic model with p processors so that the running time is increased by a factor of $\Theta(q/p)$, provided that this factor q/p is big enough. The minimum number q/p of processes in the PRAM algorithm that must be mapped onto each processor in the realistic model is called the *parallel slackness* required by the simulation [188].

(A simulation that increases running time by a factor of $\Theta(sq/p)$ is said to have *slowdown* $s = s(p)$. Efficient simulations are those with constant slowdown.)

For example, we can use block pipelining to hide latency provided that we have sufficient parallel slackness so that although each processor may make many memory accesses during each simulation step, these accesses can be grouped into blocks of length $\Omega(l)$. Many PRAM algorithms can be restructured specifically to provide for block accesses using $O(l)$ parallel slackness, as we will see in Chapter 5. For PRAM algorithms in general, somewhat more parallel slackness ($lp^{\Omega(1)}$) is needed to guarantee with high probability that block accesses can be used; this result of [5] is formally presented in Chapter 7. Parallel slackness is also exploited in efficient PRAM simulations on models with asynchrony (Chapter 8) and transient, randomly distributed network corruption (Chapter 10).

3.3.2 Hashing, skewing and combining

Assuming that memory modules have been designed with sufficient bandwidth to provide normal service, contention should occur only when the demands on the modules are distributed very unevenly. This may happen, nevertheless, when the processors attempt to access the memory in systematic patterns, as discussed in Section 3.2.2.

• *Hashing:* One solution to this problem is to assign random logical addresses to the memory locations according to some *hash function.* The probability of each particular mapping of memory accesses to modules is exactly the same; so that with high probability, the physical distribution of memory accesses is nearly uniform, regardless of the pattern of addresses specified in the algorithm. However, hashing completely at random is not practical. Hash functions must be capable of being computed quickly and specified in relatively little space, so that a processor can efficiently evaluate the hash function for all of its memory accesses.

Universal families of hash functions were introduced by Carter and Wegman [37] as a practical way to use randomized hashing. The hash function to be used is chosen randomly from a universal family consisting of functions that are easy to specify and compute. For any set of h memory addresses, the probability that the function maps them to modules in a particular pattern is approximately the same as if the hashing had been done completely at random. The number h is a parameter of the family and is a measure of its *performance,* or *independence.* High-performance families can be used to avoid contention, with high probability, in PRAM simulations [5, 141, 190]. Lower-performance families (with $h = 2$) have been adequate so far in practice [189], although extra care in choosing parameters for the family can be necessary [145].

• *Skewing:* When the patterns of shared memory access are known in advance, the memory locations can sometimes be deterministically addressed, or *skewed,* to avoid contention. A standard result, first shown by Kuck, is that it is possible to assign the entries of an $N \times N$ array to N memory modules so that each row, column and long diagonal vector contains exactly one entry in each module, e.g. see [115, 121, 122]. Similar results can be obtained for accessing square subarrays and general diagonals of two-dimensional arrays [107] and vectors in multidimensional arrays [205].

Note that hashing and skewing differ only in intent and effect. Hashing is used to prevent contention in general purpose computation (e.g. PRAM simulation) and skewing is designed to avoid contention for specific patterns of memory access. We therefore tend to think of hashing as randomized addressing, and skewing as deterministic addressing. The two words describe the use of memory addressing in two very different contexts of contention avoidance.

• *Combining:* A special kind of contention occurs when two or more processors simultaneously attempt to access the same memory location. This may happen when running algorithms written for models which allow concurrent reads or writes, such as the CREW and CRCW PRAM models. Here, readdressing the memory cannot help; the demands on a single memory address will be the same no matter where it exists physically.

One solution, attributed to Gottlieb, Lubachevsky and Rudolph [78], is for multiple requests to a memory location to be transmitted through a *combining network.* The simplest example of a combining network is a binary tree. Requests enter the network at the leaves and move toward the memory location at the root. Requests that meet at internal nodes of the tree can be combined. With the appropriate bookkeeping at each node, replies from the memory location can be sent back through the tree to the requesting processors.

It would be impractical to provide a different combining network for each memory location, but Ranade [168] has shown that this is not necessary. The wires and switches in a butterfly network of processor-memory elements can serve as the combining network for every memory location, provided that the memory is hashed (using a high-performance universal family). This enables a CRCW PRAM algorithm to run on a (fault-free) butterfly network containing the same number p of (synchronous, fault-free) processors with a $O(\log p)$ factor increase in time (with high probability).

3.3.3 Randomized routing

As described in Section 3.3.2, congestion has been observed in the routing of specific permutations around interprocessor networks, when a deterministic algorithm is used. However, random permutations can be routed, with very low probability of congestion, simply by sending each packet of data along a randomly chosen shortest path to its destination. This suggests that randomization should be introduced in order to prevent congestion.

In two-phase *randomized routing,* a message from node i to node j is sent first from i to some randomly chosen node k, then on to j, taking shortest paths each time. Note that regardless of which specific permutation is required, each phase of the randomized algorithm performs a random permutation, taken from a uniform distribution over the set of all permutations. This key idea, that routing two random permutations is preferable to routing one specific permutation, was introduced by Valiant and Brebner [187, 191] and has survived many refinements. Randomized algorithms have now been developed for routing messages around meshes, butterflies, shuffle-exchange graphs and hypercubes [9, 125, 161, 168] so that with high probability, all messages arrive at their destinations in time proportional to the diameter of the network.

It should be noted that Ranade's PRAM simulation on the butterfly network uses deterministic routing; congestion is avoided by hashing the memory address space. This demonstrates that randomized routing is not always necessary to avoid congestion in general purpose parallel computing. Thus far, however, randomized routing has been applied to a wider range of architectures than randomization through hashing alone. We should therefore expect that models of parallel computation will generally be supported in real machines by the use of randomized routing to avoid congestion. In Section 4.2.6, we discuss the implications of randomized routing for our particular study.

Information dispersal, a method for avoiding congestion as well as tolerating link failures, is described in Section 3.3.5.

3.3.4 Bulk synchrony

As we have seen in Section 3.2.3, it is possible to implement a PRAM algorithm on an asynchronous network of processors by implementing a synchronization barrier after every step. To do this, however, requires at least as much time as must be allowed for communicating across the network. Let B denote the time required to implement a synchronization barrier, as measured by the time steps of a given processor. Then $B = \Omega(l)$, where B

and l are both functions of p.

Fortunately, it is often not necessary to synchronize after every time step. Each interaction point in a PRAM algorithm occurs precisely when a write to a shared memory location is followed some time later by a read to the same location. Provided that no interaction points occur, processors running asynchronously can execute B instructions each between synchronization barriers. The B instructions, followed by a synchronization barrier, together comprise a *phase* [76] or *superstep* [189] taking time $\Theta(B)$. In this way processors in an asynchronous machine can each perform B instructions between synchronization barriers not in time $\Theta(B^2)$, but optimally, in time $\Theta(B)$. The effect is that, within constant factors, all processors take the same time to perform each phase. The overhead associated with the requirement of synchronous operation is "tolerated" or "hidden." In this manner, synchrony is achievable in *bulk*, and a PRAM model with a parameter B to represent the cost of a synchronization barrier can be called a *semi-synchronous* [76] or *bulk-synchronous* model [189].

For example, we can use bulk synchrony to hide the synchronization overhead provided that we have sufficient parallel slackness so that although each processor makes many memory accesses during each simulation step, these accesses do not induce interaction points. Many PRAM algorithms can be restructured specifically to provide for bulk synchrony using parallel slackness $O(B)$ ([76]; see also Chapter 8). It is instructive to note the similarity in principle between bulk synchrony and block pipelining. Not surprisingly, we can observe a corresponding similarity between the restructuring of PRAM algorithms to provide for these techniques. This is formalized in Section 4.2.10 and Chapter 8.

We have informally described the Phase LPRAM of Gibbons [76] (the "L" is for latency; the model is named after the two parameters added to the PRAM model, B and l. The Phase LPRAM implements synchronization only by using barriers. The idea of bulk synchrony can be generalized to include synchronization steps other than barriers; i.e. *subset synchronization* steps. Important primitives and fundamental algorithms using subset synchronization are presented in [76]. Typically, these algorithms perform best when synchronizing subsets of $\Theta(l)$ processors. For our purposes, the inclusion of subset synchronization offers very small improvements in complexity for specific problems and no improvement for general simulations, as we show in Chapter 8. Therefore, we will only consider barrier synchronization in this thesis.

3.3.5 Information dispersal

As discussed in Section 3.2.4, the problem of network routing is complicated by the presence of faulty links and nodes. If a packet of data is routed along only one path in the network, then every link and node in that path must be reliable. If several paths are used, we can be more flexible. This observation suggests three possible approaches to routing in faulty networks.

- Route each data packet around failures *adaptively,* as they are encountered in the network.
- Route several copies of each data packet along different paths. With high probability, most of the paths will be reliable and the correct version of the packet can be determined at the destination node.
- Break each data packet up into smaller fragments which give a *redundant* encoding of the data, so that the packet can be reconstructed from any majority of its fragments. This technique is called *information dispersal* [23, 165].[8] Route the fragments of each packet along different paths. With high probability, most of the paths will be reliable and the packet can be correctly reconstructed at the destination node.

Adaptive routing for fault tolerance in multiprocessor networks has been extensively studied. The most significant result to date is an efficient routing algorithm for hypercube networks. For a p-processor network, $\Theta(p)$ randomly distributed node interruptions and $\Theta(p \log p)$ randomly distributed link interruptions can be tolerated efficiently with high probability [87].

The use of multiple paths to route a single packet is a comparatively naive approach. The method can be refined by information dispersal, so that space is used more efficiently; and by adaptive routing, so that more faulty links can be tolerated. All of the results to date concern routing on the hypercube. In Rabin's original scheme for a p-processor network, $\Theta(p/\log p)$ transient, randomly distributed link interruptions can be tolerated with a $\Theta(\log p)$ slowdown with high probability [165]. By introducing adaptive

[8]Information dispersal is one of many applications of the theory of *error-correcting codes* [192].

routing, [87, 133] improved the number of link interruptions tolerated to $\Theta(p)$, and the failures do not have to be transient. However, the slowdown is still $\Theta(\log p)$.

An advantage of information dispersal is that it can be used to avoid congestion. Data fragments blocked by traffic at a link can be considered lost due to link failure. Even if each link is capable of carrying only the equivalent of six packets, all of the routing results in the previous paragraph can be achieved.

In Chapter 10, we apply the information dispersal idea in the architecture-independent context of the Block PRAM model, and we show that given $lp^{\Omega(1)}$ parallel slackness, we are able to perform the routing efficiently by bundling the fragments together. However, having hidden the network structure, we are unable to exploit adaptive routing. Therefore, we are only able to match the fault tolerance properties of Rabin's routing algorithm [165] and not the later improvements.

3.3.6 Robust execution

In the theory of sequential computation, faulty processors are anomalies. Implicit in every in sequential complexity result is the hypothesis that the machine is working correctly. Analogously, the standard practical solution to processor failure in the sequential case is straightforward: fix the fault or use another computer.

In massively parallel computers such solutions are inappropriate. It is obviously wasteful to shut down, let alone discard, a 65,000-processor parallel computer for the sake of one faulty processor. Yet, as observed in Section 3.2.4, we expect such a machine to experience processor failures much more often than a sequential computer. These concerns demonstrate the need for an on-line scheme for *robust parallel computation,* so that reliable processors can proceed with a computation even when other processors fail to contribute to the task.

Such a scheme, originally suggested by Kanellakis and Shvartsman in [99], uses a system of pointers to keep track of progress during a given time step. The reliable processors perform the instructions that the faulty processors failed to do. Each reliable processor follows its pointer to find its next assignment, then updates the pointer. Using this technique, one step of a CRCW PRAM with p processors can be simulated robustly in $O(\log p)$ time on a CRCW PRAM with $\Theta(p)$ reliable processors and possibly some faulty processors.

This scheme is refined in [105, 106]. Notably, [105] shows that this simulation of one step cannot be improved, even if no processors are actually faulty. However, the simulation can be made efficient in the expected case by using the following analogy to parallel slackness. Several steps are simulated *tentatively,* each with high probability of correctness. Counters for each processor keep track of the number of steps which have been successfully performed. This work is then verified, or *audited,* using the counters. If there has been an error, the sequence of steps is *undone* and another attempt is made. Using this framework, [105] shows that any polylogarithmic-time CRCW PRAM algorithm can be executed efficiently and robustly on a CRCW PRAM in which the (independent) probability of a processor failing during a given time step is $p^{-\Omega(1)}$.

Robust computation can be used to tolerate asynchrony. If a processor does not execute its assigned instruction during a given unit of time, it may be considered to have failed during that time unit. Robust computation reassigns an instruction to working processors until it is successfully executed. This general scheme can be tailored to specific algorithms, and it is interesting to compare the techniques in [99, 106] and the asynchronous algorithms in [52, 53, 138] to see how this is done.

Chapter 4

The Block PRAM model

In the past ten years, and especially in the past two, there has been a flood of new parallel complexity models and measures in the literature [5, 6, 52, 76, 138, 141, 148, 189]. As Kruskal, Rudolph and Snir [113] observe, "it is easy to obtain hundreds of distinct models by systematically varying all the various parameters of model definitions." The typical pattern of theoretical research into practical issues in parallel complexity has been to introduce a new model for each issue under study.

In this thesis we try to take a more coherent approach. We demonstrate that a single model—the Block PRAM model of Aggarwal, Chandra and Snir [5]—is an appropriate context for introducing the problems of communication latency, asynchrony and fault tolerance, and for evaluating methods for coping with them. A single complexity measure can be applied to a wide range of algorithms and simulations, thereby providing comparisons between alternative techniques. In this chapter we define the Block PRAM model, justify its specific features, and indicate its place in, and contribution to, the current discussion on practical issues in parallel computing.

4.1 Definition of the model

A *Block PRAM* [5] is an EREW PRAM with the following modifications. It has two parameters, the number of processors p and the communication latency l. A processor may perform a local instruction, or access a location of its local memory, in unit time. It may also access a block of b consecutive locations in the global memory in time $l+b$. Since no read or write conflicts are allowed, concurrent accesses to overlapping blocks are serviced in some arbitrary order.

4.2 Evaluation of the model

The original purpose of the Block PRAM was to be a model widely applicable to the theoretical literature on parallel complexity that, nevertheless, accounted for the practical problem of communication latency. By allowing for block pipelining, the Block PRAM model also was able to account for another important practical phenomenon, *granularity,* which had largely eluded quantitative analysis in the parallel complexity theory.

As we shall see, the applicability of the Block PRAM model to practical issues in parallel computation extends far beyond the issues explicitly addressed in its design. In this thesis, we demonstrate that the Block PRAM is an appropriate model for assessing the impact of communication latency, contention, asynchrony and network failure on the implementation of PRAM algorithms. The prospect of unifying these issues clearly motivates the study of the Block PRAM model. However, we should first justify the specific characteristics of the model, and so we devote the remainder of this chapter to a detailed evaluation of the Block PRAM.

4.2.1 Latency, pipelining and granularity

The distinguishing features of the Block PRAM model are (1) the cost imposed for latency to shared memory and (2) the allowance for block pipelining. The result is a model that can quantify an issue which had not hitherto been considered in the PRAM complexity theory, namely that of *granularity.*

Before defining granularity, let us first make some basic observations about the model. The communication latency l, expressed as a multiple of time steps, is a function of p, the number of processors. Clearly, $t^{\mathcal{A}}_{\text{EREW}}(n,p) \leq t^{\mathcal{A}}_{\text{Block}}(n,p,l) \leq (l+1) \cdot t^{\mathcal{A}}_{\text{EREW}}(n,p)$ for all Block PRAM algorithms $\mathcal{A}$ and all parameters n, p, l. Note that EREW PRAM algorithms are also Block PRAM algorithms: we can choose to use the parameter l as part of the specification of $\mathcal{A}$.

Definition.

Let $\mathcal{A}$ be a Block PRAM algorithm. The *communication overhead* of $\mathcal{A}$ is defined by $Ov^{\mathcal{A}}(n,p,l) = t^{\mathcal{A}}_{\text{Block}}(n,p,l)/t^{\mathcal{A}}_{\text{EREW}}(n,p)$.

An access to shared memory may take up to $l+1$ times as long in the Block PRAM model as in the EREW PRAM model. On the other hand, if shared memory is always accessed in blocks of l consecutive locations, block pipelining can reduce the communication overhead to a constant factor. Such algorithms are said to have good *spatial* and *temporal locality* of reference [5]: each processor's shared memory accesses tend to be close together in space and in time. They may also be called *coarse-grained* algorithms, as we now explain.

The *granularity* of an algorithm [96, 114, 173] refers to the size of the *grains,* or subtasks, into which computations have been divided for parallel processing. Fine-grained algorithms consist of relatively many small or *lightweight* processes, each consisting of relatively few steps and communicating using relatively short messages or accesses to shared memory. Coarse-grained algorithms consist of longer sequential processes which use bigger blocks for communication.

Data granularity refers to the length of the messages or blocks accessed in shared memory by each process and corresponds to spatial locality. *Process granularity* refers to the number of sequential steps in each process and corresponds to temporal locality. From long experience in implementing signal processing algorithms on reconfigurable parallel architectures, Jamieson observed qualitatively distinct effects from the two kinds of granularity [96]. Data granularity primarily affected memory structure, pipelining, hashing and other issues related to communication latency; process granularity affected all of these issues as well as synchronization requirements. Our theory of granularity will quantify many of these relationships. However, it will not model the distinction between data and process granularity, but instead will model them together.

(Our description of granularity should be distinguished from the "problem of granularity in parallel memories" discussed in [116, 141, 186, 196]. This is actually the problem of contention in realizing the EREW PRAM on parallel machines with relatively slow memory modules. As will be seen in Chapter 6, hashing, a possible solution to this problem, has the apparent effect of reducing granularity, so there is an oblique connection between the two uses of the word. However, in our view, to refer to the problem as the "granularity problem" begs the question as to whether there are other solutions.

(A more closely related use of the word "granularity" has been to refer to the size of the individual processors and local memories in a parallel computer [12]. However, the effect of this "granularity" on the complexity theory still depends on the algorithm under study, and it is therefore more instructive to study granularity as an algorithmic issue.)

Until very recently, the literature on granularity as a complexity issue consisted of empirical performance analysis and specific case studies, notably on interchanging the nested loops in linear algebra algorithms [22, 56, 70, 204]. Typically the communication overhead $Ov^{\mathcal{A}}$, a value between 1 and $l+1$, varies with n and p as well as l, and an algorithm or problem may be fine or coarse for different ranges of parameters. With a three-dimensional space of parameters to explore for each algorithm, it is not surprising that granularity was studied so long in the absence of a theoretical model.

The Block PRAM allows the issue of granularity to be studied in the same frame of reference as the parallel complexity theory. In particular, we can now give quantitative, architecture-independent definitions of granularity.

Definitions.

Let $\mathcal{A}$ be a Block PRAM algorithm. Let the tuple $(\mathcal{A}, n, p, l)$ denote an *implementation* of $\mathcal{A}$ for a problem of size n on a Block PRAM with p processors and latency l. Then (A, n, p, l) is:

- *fine* if $Ov^{\mathcal{A}}(n,p,l) = \Omega(l)$;
- *almost fine* if $Ov^{\mathcal{A}}(n,p,l) = \Omega(l/\log l)$;
- *almost coarse* if $Ov^{\mathcal{A}}(n,p,l) = O(\log l)$; and
- *coarse* if $Ov^{\mathcal{A}}(n,p,l) = O(1)$.

We choose to give names to these particular ranges for $Ov^{\mathcal{A}}$ because they predominate in our analysis throughout this thesis, and because the resulting classification is simple and meaningful.

Definition.

Let Π be a problem for which there exists an efficient EREW PRAM algorithm and let $S \subseteq \mathbf{N}^3$. Then Π is:

- *fine (almost fine) for* $(n,p,l) \in S$ if for any efficient EREW PRAM algorithm $\mathcal{A}$ and for all $(n,p,l) \in S$, the implementation $(\mathcal{A},n,p,l)$ is fine (almost fine); and
- *coarse (almost coarse) for* $(n,p,l) \in S$ if there is an efficient EREW PRAM algorithm $\mathcal{A}$ such that for all $(n,p,l) \in S$, the implementation $(\mathcal{A},n,p,l)$ is coarse (almost coarse).

4.2.2 Exclusive memory access

The Block PRAM is essentially an EREW PRAM with adjusted costs for accesses to shared memory. Since the CREW and CRCW PRAM models also have an important place in the parallel complexity theory, we should defend the choice of the exclusive memory access protocol.

- *Empirical accuracy:* Because of memory contention, it is unlikely that block pipelining will continue to occur when block accesses overlap in arbitrary patterns. In other words, having accepted block pipelining in our model for whatever reason, we must commit to an EREW protocol.
- *Theoretical simplicity.* Algorithms and simulations which apply to an EREW protocol will run on real machines under an EREW, CREW or CRCW protocol.
- *Theoretical power.* The weak EREW protocol allows us to differentiate between EREW, CREW and CRCW PRAM algorithms when we try to implement them in the Block PRAM model. It also forces us to consider the problem of providing for memory contention alongside the complexity of algorithms.

4.2.3 Unbounded memory

As in the PRAM model, the Block PRAM shared memory and local memories are unrestricted in size. This allows us to focus on the structural aspects of efficient algorithms and lower bounds, resulting in a clearer presentation.

Most of our complexity results being asymptotic, we assume that memories will be built and managed to scale up with the number of processors and the space requirements of algorithms. We assume that long contiguous arrays will always be available to our algorithms, deferring the important and nontrivial problem of on-line parallel garbage collection and compaction to another study, e.g. [91]. Accordingly, from our point of view, limits on memory size will serve only to limit the range of applicability of our algorithms. Conversely, our lower bounds will apply to machines with memories of any size.

4.2.4 Explicit parallelism/explicit memory addressing

Parallel algorithm design is not yet automatic. It is often difficult, given a fast sequential algorithm, to produce an efficient parallel algorithm. *Parallelizing compilers* have been designed with the purpose of automatically finding and exploiting parallelism in sequential programs, so that they can run on parallel computers [13]. However, years of research have found many data structures and techniques in efficient parallel algorithms which do not appear in standard sequential algorithms [76]. Unless parallelism is specified in an algorithm, we cannot be certain that it will be found and exploited at runtime. For this reason, all theoretical models of parallel complexity require that parallelism be specified explicitly. The Block PRAM is no exception.

Similarly, *clustering,*[1] the combining and scheduling of primitive parallel operations together into subtasks according to spatial and temporal locality, is not yet automatic. *Vectorizing compilers* have been designed with the purpose of automatically finding and exploiting contiguous array data structures in parallel programs, so that they can run on vector computers [31, 117]. However, research on the Block PRAM model has found many data structures and techniques for clustering which do not appear in standard PRAM algorithms; see [5] and Chapter 5. Unless clustering is specified in an algorithm, we cannot be certain that it will be found and exploited at runtime. For this reason, any parallel complexity model that accounts for spatial and temporal locality must require that clustering is specified by the algorithm designer or programmer.

In the Block PRAM model, clustering occurs when a processor accesses a block of consecutive memory addresses, and so we require that the shared

[1]Also called "vectorizing" [31, 117] and "flattening" [31].

memory be addressed explicitly. Since the Block PRAM shared memory is unbounded and algorithms are designed to work on data structures of specified size, Block PRAM algorithms address the shared memory through user-defined arrays. This entails two assumptions:

- Disjoint arrays of arbitrary size consisting of locations with consecutive logical addresses can be allocated in the shared memory space.
- Shared memory locations can be addressed according to their positions in the relevant array.

By identifying frequently occurring structures and techniques for parallelizing (e.g. binary trees, shortcutting) and clustering (e.g. l-ary trees, rational permutations), parallel complexity models can contribute to the development of effective compilers. Because clustering is performed on programs that are already parallelized, clustering may be viewed as a "fine-tuning" compiling problem and is probably easier in general than parallelizing. Therefore, we may expect the development of general purpose clustering compilers, aided by a clear theory of Block PRAM algorithms and lower bounds, to precede the development of general purpose parallelizing compilers by many years. (See [136] for a discussion of clustering compilers in a programming context.)

4.2.5 Two-level block-hierarchical shared memory

The processors in the PRAM model communicate with each other, and with the user, through the shared memory. The alternative way of modeling interprocessor communication is *message passing.* Detailed comparisons between the shared-memory and message-passing paradigms can be found in [15, 76]. The general consensus is that software development is easier for shared-memory models, and that hardware implementation is easier for message-passing models. Since we are engaged in the bridging of theory and practice, the most significant point for our purposes is that the vast majority of parallel algorithms in the theory literature have been designed for shared-memory models. We wish to learn how to implement these algorithms in practice, and so it is appropriate for us to study a practical shared-memory model.[2]

[2]It should be noted that the Block PRAM model, by allowing block pipelining, offers the best representation in a shared-memory model of message-passing machines which

The Block PRAM shared memory is *hierarchical,* and a unique characteristic of the Block PRAM model is the formulation of that hierarchy. In a computation model with hierarchical memory [2, 3, 4], each processor views the memory as consisting of various levels, according to the difficulty of performing an access to each location; each processor may have a different point of view. For example, in a sequential model, the levels may range from computing registers through various levels of cache to main memory, extended memory, various kinds of disks and mass storage. A parallel processor might also include levels for the local memories of other processors at varying distances from it in the network, and/or distinct levels for local and shared memory. In such a hierarchical memory model, a read or write by a processor to a location is subjected to a delay (possibly zero) depending on the level of the hierarchy to be accessed. In a hierarchical memory model with *block transfer* (or *block-hierarchical memory* model for short), when a block of consecutive locations is accessed, the delay is only charged once, e.g., for the first location accessed.

It should be clear from the definition that the Block PRAM is a two-level block-hierarchical shared memory model. Because it does not model the memory hierarchy of any network topology, the Block PRAM is said to be an *architecture-independent* model of parallel computation. Algorithms can be designed for the Block PRAM model without regard to the interconnection structure of the target multiprocessor network, and our conclusions apply equally well to architectures based on hypercubes, butterflies and meshes.

4.2.6 Network congestion

The price of architecture independence is that the Block PRAM cannot model the pattern of traffic flow in the network. The flat charge of l time units for a given processor to perform a non-local memory access does not take into account the possibility that the data may have to travel through heavily congested wires in the network. This results in a significant discrepancy between the Block PRAM model and current-generation massively parallel machines. We believe that this gap has been primarily responsible for the fact that the Block PRAM model appears to have been abandoned

allow long messages (e.g. [24]). A message-passing variant of the Block PRAM is described in Section 7.1.

in the literature in the two years since its first appearance in [5]. In this thesis, we present two approaches to closing this discrepancy: minimizing the effect of network congestion, and modeling specific deterministic routing problems.

• First, we argue that congestion can be minimized by the use of randomized routing. As we discussed in Section 3.3.3, randomized algorithms can guarantee congestion-free routing of permutations with high probability. Recent algorithms [9, 126] even support the efficient routing of large message packets, corresponding to block transfers of shared memory. However, permutations represent highly uniform (bijective) patterns of memory access. To support general patterns of memory access with randomized routing, the shared memory space has to be randomly hashed so that the distribution of memory requests is nearly uniform. Hashing has the apparent effect of destroying spatial locality, thereby striking at the heart of the Block PRAM model. Earlier this year, Heywood and Ranka [89] commented: "One drawback to current PRAM simulation techniques is that they obliterate (through hashing) any natural communication latency that may be present in an algorithm."

We show in Chapter 6 that this is not necessarily the case, by exhibiting a universal family of hash functions with optimal locality. Subject to reasonable conditions, these functions can be performed and inverted efficiently with almost coarse granularity. If the memory space of a real machine is hashed using a random function in this family, a block access instruction can be supported using large-packet randomized routing; see Chapter 7. In other words, randomized routing can be used to minimize the effect of network congestion on the complexity of many Block PRAM algorithms.

• Second, we argue that deterministic network routing problems can be modeled reasonably well on the Block PRAM. The permutations most often studied in the literature [128, 147, 157, 167] as network routing problems are *rational permutations.*[3] A permutation Π on $\{0, 1, \ldots, 2^k - 1\} \cong \{0,1\}^k$ is *rational* if it can be expressed as a permutation π on the bit positions; i.e., if $\Pi(x_1, \ldots, x_k) = (x_{\pi(1)}, \ldots, x_{\pi(k)})$. Matrix transpositions and perfect shuffles are examples of rational permutations.

[3]The use of the word "rational" was introduced in [4]. The permutations are called "Frequently Used Bijections" in [128] and "admissible mappings" in [157].

In Chapter 5, generalizing work in [5], we give optimal bounds for the Block PRAM complexity of performing all rational permutations. Standard deterministic permutation algorithms on networks [27, 128, 147, 157, 167, 184, 185], and the optimal Block PRAM permutation algorithm [5] amount to factoring an arbitrary permutation into a product of rational permutations. We conclude that for practical purposes, our state of knowledge about deterministic routing in the Block PRAM model is about as good as it is for network models, and we are able to draw comparisons and contrasts between these complexity theories.

4.2.7 Logarithmic word size

In shared-memory models of parallel computation, *word size* is the number of bits stored in each shared memory cell. An access to a shared memory location is assumed to make all of these bits available for processing, and a basic unary or binary local instruction on words is assumed to take unit time. It is not surprising, then, that models with large word size can be asymptotically more powerful than models with small word size. For example, Bellantoni has recently exhibited a hierarchy of CRCW PRAM models based on word size [29]. This paper includes the result that computing the MOD_2 function on n inputs (see Section 9.2) requires $\Omega(\log n/\log\log n)$ time on Priority CRCW PRAMs with $O(1)$ word size, compared with the best known lower bound of $\Omega(\log^{1/2} n)$ on Priority CRCW PRAMs with unbounded word size [129], regardless of the number of processors and the overall size of the local and shared memories. Therefore, it seems important to fix the word size when defining a complexity model.

Although PRAMs with $O(1)$ word size can be used for general purpose parallel computing, e.g. by simulating Boolean circuits (see Section 4.2.8) [183], almost all of the PRAM algorithms in the literature assume a word size of $\Omega(\log p)$. (Usually the word size is not even specified and is assumed to be unbounded.) Graphs, linked lists and permutations, as well as many database and optimization problems, are normally specified as arrays of integers in the range $[0\ldots p^k]$ for some positive constant k, and PRAM algorithms using these data structures manipulate these integers as words. For these applications, a word size of $\Theta(\log p)$ is also sufficient.

Except in research such as [29, 154, 179] specifically studying the effect of word size on the complexity theory, it is not standard practice for a parallel complexity model to have a fixed word size. Even the original definition of the Block PRAM model [5] does not restrict word size. We believe that this omission belies the central purpose of the Block PRAM: modeling communication latency and pipelining. In practice, a word that is longer than the word size of a machine must be expressed as an array of smaller words [31, 91]; therefore, the effect of block pipelining in practice corresponds most closely to a complexity model with fixed word size. It is therefore essential to restrict the word size if we are to have a realistic complexity model for latency.

Fixing the word size is also necessary in this thesis if we are to discuss meaningfully any practical issues relating to contention, bandwidth or pipelining—or the rate of communication in general—since the relevant theoretical measure is words per time unit, and the relevant real measure is bits per second. It should also be considered that real machines have $O(1)$ word size, and it is far easier to implement algorithms on them if their word size is bounded [91]. Our choice of $\Theta(\log p)$ word size applies to most parallel algorithms in the theoretical literature and results in the simplest, most general, most rigorous Block PRAM complexity theory possible.[4]

4.2.8 Restricted arithmetic instruction set

The *instruction set* of a *unit-cost* RAM is a list of the operations it can perform on words in unit time. Most parallel complexity models in the literature fail to specify an instruction set for their constituent RAMs, taking the view of [7] that "the exact nature of the [instruction set] is not too important, as long as the instructions resemble those usually found in real computers." However, as [11, 154, 183] argue, the arithmetic operations allowed by instruction sets can vary in power, greatly affecting the complexity of algorithms. Parberry [154] describes five arithmetic instruction sets for RAMs, in increasing order of power:

- *Minimal:* Addition, subtraction, and shifts of a single bit (multiplication and division by 2).

[4]The theory can be extended to other word sizes. In Section 7.4 we consider the effect of *multigauging,* or changing the word size, on the Block PRAM model.

• *Restricted arithmetic:* The above, and also shifts of arbitrary length (multiplication and division by powers of 2).

• *Full arithmetic:* The above, and also multiplication and integer division.

• *Extended arithmetic:* The above, and also exponentiation.

• *General:* Any operation which can be performed by a sequential computer (e.g. a unit-cost RAM with any of the above instruction sets) in time polynomial in the size of the operands.

Which of these sets, if any, is appropriate for the RAMs which make up the Block PRAM model? If the word size were $O(1)$, there would be no basis for comparison: any of the above arithmetic operations could be performed sequentially in constant time. With a $\Theta(\log p)$ word size, however, differences in power appear: clearly, the general instruction set is asymptotically more powerful than the minimal instruction set.

The most natural way to quantify these distinctions is with the theory of Boolean function complexity. The realization of Boolean functions by logic and algebraic circuits has been a long-standing concern in electronic engineering. Recently, the study of Boolean function complexity has been motivated by its close relationship to the structural complexity theory. (For example, Turing machines can be efficiently simulated by Boolean circuits, and so lower bounds on circuit size may lead to a proof that $\mathcal{P} \neq \mathcal{NP}$.) Because of its generality and robustness, Boolean function complexity has emerged as a topic of fundamental importance in the theory of computation. For surveys of Boolean function complexity, see [59, 202].

Definitions.

Let B_n denote the set of n-ary *Boolean functions* $f : \{0,1\}^n \to \{0,1\}$. An *$\Omega$-circuit* on *inputs* $X = \{x_1, x_2, \ldots, x_n\}$ is a finite collection of *gates,* which may be represented as the nodes of a connected acyclic directed graph $G = (V, A)$, with $X \subseteq V$. In such a representation, each node $v_i \in V \backslash X$ is assigned a (Boolean) *basic function* $\omega_i \in \Omega \cap B_{n(i)}$ and some ordered $n(i)$-tuple $(u_1, u_2, \ldots, u_{n(i)})$ of *predecessors,* where $\{u_1, u_2, \ldots, u_{n(i)}\} = \{u \in V : (u, v) \in A\}$.

By $\mathrm{res}_{v_i} : \{0,1\}^n \to \{0,1\}$ we denote the Boolean function computed at node v_i, defined inductively for $y \in \{0,1\}^n$:

- $\mathrm{res}_{x_i}(y) = y_i$;
- $\mathrm{res}_{v_i}(y) = \omega_i(\mathrm{res}_{u_1}(y), \mathrm{res}_{u_2}(y), \ldots, \mathrm{res}_{u_{n(i)}}(y))$ for $v_i \in V \backslash X$.

The *output* of the circuit is some distinguished node v_0. The circuit *computes* the Boolean function res_{v_0}.

The *size* of the circuit is $|V|$, the number of gates or nodes. The *depth* of the circuit is the length of a longest directed path to the output node.

A *formula* is a circuit in which every gate has fanout (outdegree) one.

The key complexity measures of Boolean functions are circuit size, circuit depth and formula size. Research on the complexity of specific problems has included bounds for each of these measures over the bases B_2, $U_2 = \{\wedge^{(2)}, \vee^{(2)}$[5]$, \neg\}$, $U_* = \{\wedge^{(m)}, \vee^{(m)}, \neg : m \in \mathrm{N} \text{ and } m \geq 2\}$, $M_2 = U_2 \backslash \neg$, $M_* = U_* \backslash \neg$, and $\{\mathrm{NAND}\}$. Other important work has included finding lower bounds on the size of constant-depth and planar circuits. Unless otherwise specified in this thesis, we will will understand circuits to be over the basis U_2.

As observed in [154], the circuit size complexity of a Boolean function is (asymptotically) equal to the complexity of computing the function on a unit-cost RAM with a minimal instruction set and word size 1. In multiple precision arithmetic applications such as cryptography [203], the circuit size complexity of an operation is referred to simply as its "complexity," "bit complexity" or "bit serial complexity." Table 4.1 gives the best known bounds on circuit size, for n-bit inputs, for each of the operations in Parberry's taxonomy of instruction sets. It can be seen that the bounds correspond roughly to the increasing levels of power in the taxonomy, and in particular, that multiplication and integer division are indeed more advanced RAM operations than addition and subtraction.[6]

We use the circuit size, then, to describe an operation's power as a member of an instruction set. Having adopted this criterion, it makes sense for us initially to define the Block PRAM model with the restricted arithmetic instruction set. This will have the effect that all of the basic instructions

[5]U_2 is usually defined as $U_2 = B_2 \backslash \{\equiv, \oplus\}$. This definition and ours have been shown to be equivalent with respect to depth and asymptotically equivalent with respect to size in [134].

[6]Similar objections to multiplicat ion as a basic instruction have been put forward by Blum [32] and Parberry [153].

Operation	Circuit Size Bound
Addition, subtraction	$\Theta(n)$
Shifts of arbitrary length	$\Theta(n)$
Multiplication	$O(n \log n \log\log n)$ [174]
Integer division	$O(n \log n \log\log n)$ [16]
Exponentiation	$O(n \log n \log\log n)$ [16]
General (polynomial-time)	$n^{O(1)}$ [160]

Table 4.1: Circuit size of proposed basic instructions

can be performed sequentially with bit operations in time linear in the word size, or logarithmic in the maximum numerical value of a word.[7]

When it is desirable to do so, we will augment the instruction set, and so indicate. For example, in Chapter 5, describing algorithms for matrix multiplication or the Fast Fourier Transform, we will refer to "the Block PRAM model with multiplication" rather than explore the fine structure of integer multiplication. In Chapter 7, we will define *corner turning,* a natural but non standard instruction for the Block PRAM model, and its practicality will be defended in part by the fact that it has linear circuit size complexity.[8]

4.2.9 Synchronous, fault-free

The basic Block PRAM is synchronous, and free of network and processor failures. In Chapters 8 and 10 we show how Block PRAM algorithms can be implemented in the presence of asynchrony and network failures, respectively. In this way we are able to consider the effect of each of these practical issues, in turn, on the basic model.

[7]This means that to use the alternative "log-cost" charge for a RAM instruction [7] would also be correct in terms of bit complexity, rather than only approximately correct as in the case of multiplication. Our criterion is also consistent with the "semilogarithmic-cost" charge [97], for all polynomial-time computations.

[8]Another criterion, realization by polynomial size and constant depth circuits with unbounded fan-in, is proposed in [183] in order to relate PRAM time and circuit depth. This also has the effect of excluding multiplication and recommending the restricted arithmetic instruction set. We could adopt this criterion without affecting any results or conclusions in this thesis. For example, corner turning satisfies this constant-depth criterion as well.

4.2.10 SIMD

Like the PRAM, the Block PRAM is a *Single Instruction stream/Multiple Data stream* (SIMD) model of parallel computation. "Multiple data stream" means that the processors can work simultaneously on distinct sets of data. "Single instruction stream" means that during each time step, every processor is either executing the same instruction or waiting. Single instruction stream models are necessarily synchronous.

In *Multiple Instruction stream/Multiple Data stream* (MIMD) models, e.g. the Asynchronous PRAM (Chapter 8), each processor has its own program, which it executes autonomously. Multiple instruction stream models are generally asynchronous, but usually they allow algorithms to implement some kind of synchronization operation (at some cost). Since MIMD algorithms can exhibit more general behavior than SIMD algorithms, MIMD models are considered to be more general purpose than SIMD models. (This distinction need not manifest itself in the complexity theory: if we allow the processors in an SIMD model to interpret their local memories as instructions, it can simulate an MIMD model with only a constant factor increase in time [91].)[9]

SIMD algorithms are also more efficient than MIMD algorithms, writes Hillis [91], "for well-structured problems with regular patterns of control ... because more of the hardware is devoted to operations on the data." Because of the preponderance of research on the PRAM model, we should expect that most efficient parallel algorithms in the complexity theory literature are "well-structured" and, according to Hillis, best suited to SIMD-style programming.

We give strong theoretical support for Hillis's assertion. In Chapter 8, we will see that most of the benchmark problems for the (MIMD) Phase LPRAM model in Gibbons's thesis [76]—prefix sums, Fast Fourier Transform, bitonic sorting and matrix multiplication—can be solved within the same time bounds by generically simulating the corresponding Block PRAM algorithms (Chapter 5).[10] An important exception, list ranking, occurs be-

[9]The other possibilities in this taxonomy, due to Flynn [65], are SISD and MISD. SISD models are sequential. MISD models apply to a very limited range of problems; see [11].

[10]It is important to note that no such generic method exists for obtaining Block PRAM algorithms from Phase LPRAM algorithms on the Block PRAM, because the Phase

cause the Block PRAM models block pipelining and the Phase LPRAM does not. In other words, the proponents of MIMD complexity models have found it difficult to exhibit an advantage in return for the added difficulty of MIMD-style programming.

In conclusion, we should try to specify efficient parallel algorithms in SIMD models whenever possible. If we also seek to address practical issues in parallel computation, the Block PRAM, an SIMD model which can be adapted to capture many practical issues including asynchrony, is an ideal model for the task. This is the work of the next chapter.

LPRAM allows arbitrary pipelining.

Chapter 5

Problems and data structures

This chapter presents algorithms and lower bounds for solving several specific problems on the Block PRAM model. The results appear in sections according to the predominant data structure associated with each problem. Table 5.1 is a guide to the organization of this chapter.

5.1 Permutations of arrays

The natural data structure in the Block PRAM model is the user-defined array, consisting of a block of shared memory locations with consecutive logical addresses. Data in the Block PRAM shared memory is addressed according to its location in the relevant array. To exploit locality, it may be necessary to permute the data in an array.

In this section, we study the complexity of performing specific permutations on the Block PRAM model. As in [5], it is assumed that the permutation to be performed is known in advance, so that the only operation required is the movement of the array data in the shared and local memories. The resulting algorithms are said to be *conservative*. A computation is conservative if the only operation allowed is that of copying elements in memory.

For convenience throughout this section, we assume p and l are integral powers of 4, affecting our bounds by at most a constant factor.

Section	Data structure	Problems
5.1	Array	Permutations of arrays
5.2	Binary tree	Prefix sums computations
	Binary tree	Integer sorting
5.3	Linked list	List ranking
5.4	Expression tree	Tree contraction
5.5	Graph	Graph connectivity
5.6	String	String matching
5.7	Butterfly	Fast Fourier Transform
5.8	Matrix	Matrix multiplication
	Sorting network	General sorting

Table 5.1: Problems studied in this chapter

We study four classes of permutations, in increasing order of generality: rational, dihedral, linear and arbitrary permutations.

5.1.1 Rational permutations

Definition. [4]

A permutation Π on $\{0,1,\ldots,2^k-1\} \cong \{0,1\}^k$ is *rational* if it can be expressed as a permutation π on the bit positions; i.e. $\Pi(x_1,\ldots,x_k) = (x_{\pi(1)},\ldots,x_{\pi(k)})$. By convention, we write the most significant bit first. If a rational permutation is denoted by a capital Greek letter, we denote its associated bit permutation by the corresponding small Greek letter.

Definition.

Let Π be a rational permutation on $\{0,1,\ldots,2^k-1\}$ with bit permutation π. Denote $K = \{1,\ldots,k\}$, $C = \{1,\ldots,\log p\}$, and $F = \{k-\log l+1,\ldots,k\}$. Let $C(\Pi) = \{i \in C : \pi(i) \notin C\}$ and $F(\Pi) = \{i \in F : \pi(i) \notin F\}$. Define $c(\Pi)$, the *coarse crossing number* of Π, by $c(\Pi) = |C(\Pi)|$ and $f(\Pi)$, the *fine crossing number* of Π, by $f(\Pi) = |F(\Pi)|$. Define $\mu(\Pi)$, the *crossing number* of Π, by $\mu(\Pi) = \min(c(\Pi), f(\Pi))$.

Theorem 5.1.

Let Π be a rational permutation. The complexity of performing Π conservatively on $n = 2^k$ consecutive locations in shared memory is

- $\Theta(n/p + n\mu(\Pi)/(p\log(2n/pl)))$ if $pl \leq n$;
- $\Theta(l + l\mu(\Pi)/\log(2pl/n))$ if $pl > n$. $\square$

Lemma 5.2.

Let Π be a rational permutation. Π can be performed conservatively on $n = 2^k$ consecutive locations in shared memory in time

- $O(n/p + n\mu(\Pi)/(p\log(2n/pl)))$ if $pl \leq n$;
- $O(l + l\mu(\Pi)/\log(2pl/n))$ if $pl > n$.

Proof.

Case 1: $pl \leq n$. We may assume $pl \leq n/2$, affecting our bounds by at most a constant factor. We perform a sequence of basic rational permutations of one of the following forms:

- Rational permutations Σ where $\sigma(i) = i$ for $i \in C$. Each processor reads and permutes a block of size n/p and writes it back.
- Rational permutations Σ where $\sigma(i) = i$ for $i \in F$. Each processor reads n/pl blocks of size l and writes them into their new locations.

Each basic permutation can be performed in time $O(n/p)$.

Assume $c(\Pi) \leq f(\Pi)$; the case $f(\Pi) < c(\Pi)$ is analogous. Let $S = K\backslash(C \cup F) \neq \emptyset$. In one basic permutation, any $|S|$ (or fewer) bit positions can be moved from $C(\Pi)$ into S; in a second basic permutation, they can be moved to their images under π. When all of $C(\Pi)$ has been moved in this way, two more basic permutations suffice to map the remaining positions in C and $K\backslash C$ to their images under π. The number of basic permutations required is $2\lceil c(\Pi)/|S|\rceil + 2 = O(c(\Pi)/\log(2n/pl))$.

Case 2: $pl > n$. Similarly to Case 1, assume $pl \geq 2n$. We use the same basic permutations as above. Each basic permutation can be performed in time $O(l)$.

Assume $c(\Pi) \leq f(\Pi)$; the case $f(\Pi) < c(\Pi)$ is analogous. Let $T = C \cap F \neq \emptyset$. In one basic permutation, any $|T|$ (or fewer) bit positions can be moved from $\pi^{-1}(C(\Pi))$ into T; in a second basic permutation, they can be moved to their images under π. When all of $\pi^{-1}(C(\Pi))$ has been moved in this way, two more basic permutations suffice to map the remaining bit positions in C and $K\backslash C$ to their images under π. The number of basic permutations required is $2\lceil|\pi^{-1}(C(\Pi))|/|T|\rceil = 2\lceil c(\Pi)/|T|\rceil + 2 = O(c(\Pi)/\log(2pl/n))$. □

The proof of many lower bounds in this thesis is based on a potential function argument first used in [5] to prove a lower bound for transposing a square matrix on the Block PRAM. The argument can be stated in the following definition and lemma.

Definition.

Let $A = \{0, 1, \ldots, n-1\}$, let $m|n$ and let A be divided into n/m *segments* $A_0, \ldots, A_{n/m-1}$ each of length m: $A_i = \{im, \ldots, (i+1)m - 1\}$. Let Π be any (not necessarily rational) permutation on A. For $0 \le r, s \le n/m - 1$, let $x_{r,s}(\Pi, m) = |\Pi(A_r) \cap A_s|$. ($x_{r,s}(\Pi, m)$ denotes the number of elements that are mapped from A_r to A_s.) Define $\Phi(\Pi, m)$, the *m-wise potential* of Π, by $\Phi(\Pi, m) = \sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x^+_{r,s}(\Pi, m) \log x^+_{r,s}(\Pi, m)$ (the sum is taken over positive $x_{r,s}$ only).

Lemma 5.3. [5]

Let Π be any (not necessarily rational) permutation on $\{0, 1, \ldots, n-1\}$. Any conservative algorithm for performing Π on n consecutive locations in shared memory requires time

- $\Omega(n/p + (n \log l - \Phi(\Pi, l))/(pl \log(2n/pl)))$ for $pl \le n$ and $l \le p$;
- $\Omega(n/p + (n \log p - \Phi(\Pi, p))/(pl \log(2n/pl)))$ for $pl \le n$ and $p < l$;
- $\Omega(l + l(n \log(n/p) - \Phi(\Pi, n/p))/(n \log(2pl/n)))$ for $pl > n$ and $l \le p$;

and

- $\Omega(l + l(n \log(n/l) - \Phi(\Pi, n/l))/(n \log(2pl/n)))$ for $pl > n$ and $p < l$. □

Lemma 5.4.

Let Π be a rational permutation on $\{0, 1, \ldots, n\}$ with $n = 2^k$. Let $m = 2^j$ with $j < k$. Let $G(m) = \{k - j + 1, \ldots, k\}$, let $G(\Pi, m) = \{i \in G : \pi(i) \notin G(m)\}$ and let $g(\Pi, m) = |G(\Pi, m)|$. Then $\Phi(\Pi, m) = n \log m - g(\Pi, m)$.

Proof.

Let $r, s \in \{0, \ldots, 2^{k-j} - 1\} \cong \{0,1\}^{k-j}$. Then by a simple counting argument,

$x_{r,s}(\Pi, m) = |\Pi(A_r) \cap A_s|$
$= 0$ if there is an i with $1 \le i, \pi(i) \le k - j$ and $r(i) \ne s(\pi(i))$;
$= 2^{j-g(\Pi,m)}$ otherwise.

Moreover, for a given s, there are exactly $2^{g(\Pi,m)}$ choices of r for which $x_{r,s}(\Pi, m)$ is nonzero. Hence
$\Phi(\Pi, m) = \sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x^{+}_{r,s}(\Pi, m) \log x^{+}_{r,s}(\Pi, m)$
$= \sum_{s=0}^{n/m-1} 2^{g(\Pi,m)}(2^{j-g(\Pi,m)} \log 2^{j-g(\Pi,m)})$
$= (n/m)2^j(j - g(\Pi, m))$
$= n \log m - g(\Pi, m).$ □

Corollary 5.5.

Let Π be any rational permutation on $\{0, 1, \ldots, n-1\}$. Any conservative algorithm for performing Π on $n = 2^k$ consecutive locations in shared memory requires time

- $\Omega(n/p + n\mu(\Pi)/(p\log(2n/pl)))$ if $pl \le n$;
- $\Omega(l + l\mu(\Pi)/\log(2pl/n))$ if $pl > n$. □

This completes the proof of Theorem 5.1.

Corollary 5.6.

The complexity of transposing an $a \times b$ matrix conservatively on $n = 2^k$ consecutive locations in shared memory is

- $\Theta(n/p + n \log \min(p, l, a, b)/(p \log(2n/pl)))$ if $pl \le n$;
- $\Theta(l + l \log \min(p, l, a, b)/\log(2pl/n))$ if $pl > n$. □

5.1.2 Dihedral permutations

Definition.

A permutation Π on $\{0, 1, \ldots, 2^k - 1\} \cong \{0,1\}^k$ is *dihedral* if there is an *offset vector* $\alpha \in \{0,1\}^k$ and a permutation π on the bit positions such that $\Pi(x_1, \ldots, x_k) = (x_{\pi(1)} \oplus a_1, \ldots, x_{\pi(k)} \oplus a_k)$. ($\oplus$ is the bitwise exclusive-or operation.) The *rationalization* of a dihedral permutation Π with bit permutation π is the permutation Π_0 defined by $\Pi_0(x_1, \ldots, x_{\pi(k)})$.[1]

[1]The class of dihedral permutations is also called the class F [147] or the class BPC [167].

All of the notations and definitions related to rational permutations can be extended naturally to dihedral permutations Π through their rationalizations.

Lemma 5.7.

Let Π be a dihedral permutation on $\{0,1,\ldots,n-1\}$ and let Π_0 be its rationalization. Let $t(\Pi,n,p,l)$ (resp. $t(\Pi_0,n,p,l)$) denote the complexity of performing Π (resp. Π_0) on $n=2^k$ consecutive locations in shared memory. Then $t(\Pi,n,p,l)=\Theta(t(\Pi_0,n,p,l))$.

Proof.

Let α be the offset vector for Π_0. Any dihedral permutation can be transformed into its rationalization by applying two basic permutations, each taking time $O(n/p+l)$.

• The first permutation applies the offsets given by the last $\log l$ locations of α. Each processor reads and permutes n/pl blocks of size l and writes them back.

• The second permutation applies the offsets given by the first $k-\log l$ locations of α. Each processor reads n/pl blocks of size l and writes them into their new locations. □

Corollary 5.8.

Let Π be a dihedral permutation. The complexity of performing Π conservatively on $n=2^k$ consecutive locations in shared memory is

• $\Theta(n/p+n\mu(\Pi)/(p\log(2n/pl)))$ if $pl\leq n$;

• $\Theta(l+l\mu(\Pi)/\log(2pl/n))$ if $pl>n$. □

The crossing numbers of some frequently used examples of dihedral permutations are evaluated in Table 5.2. For convenience, we assume k is even.

5.1.3 Linear permutations

Definition.

A permutation Π on $\{0,1,\ldots,2^k-1\}\cong\{0,1\}^k$ is *linear* if there is a nonsingular $k\times k$ 0-1 valued matrix M and an offset vector $\alpha\in\{0,1\}^k$ such

Permutation Π	π	$\mu(\Pi)$
Identity	$[1,\ldots,k]$	0
Vector reversal	$[1,\ldots,k]$	0
Exclusive-or	$[1,\ldots,k]$	0
Perfect shuffle	$[k,1,\ldots,k-1]$	1
j-th power of shuffle	$[k-j+1,\ldots,k-j]$	$\log\min(p,l,2^{\lvert j\rvert},2^{k-\lvert j\rvert})$
$a\times b$ matrix transpose	$[k-\log a,\ldots,k-\log a-1]$	$\log\min(p,l,a,b)$
Square matrix transpose	$[k/2,\ldots,k/2-1]$	$\log\min(p,l)$
Bit reversal	$[k,\ldots,1]$	$\log\min(p,l)$
Bit shuffle	$[1,3,\ldots,k-1,2,4,\ldots,k]$	$\log\min(p,l)$
Shuffled row-major	$[1,k/2+1,2,\ldots,k]$	$\log\min(p,l)$
j-way shuffled row-major	$[1,k/j+1,2k/j+1,\ldots,k]$	$\log\min(p,l)$

Table 5.2: Crossing numbers of some dihedral permutations

that $\Pi(x) = Mx + \alpha$. (All matrix arithmetic is modulo 2.) If α is the zero vector the permutation Π is *homogeneous linear.*

Theorem 5.9.

Let Π be a homogeneous linear permutation. Π can be performed conservatively on $n = 2^k$ consecutive locations in shared memory in time

- $O(n/p + n\log\min(p,l)/(p\log(2n/pl)))$ if $pl \le n$;
- $O(l + l\log\min(p,l)/\log(2pl/n))$ if $pl > n$.

Proof.

We give the proof for the case $pl \le n$, $l \le p$; the other cases are analogous. Let M be the matrix for Π. We recall that any nonsingular square matrix M can be factored into the form LUP, where L is a lower triangular matrix, U is an upper triangular matrix, and P is a permutation matrix (e.g. see [66]). We can perform the permutation Mx by successively applying P, U and L to x. We show that each of these permutations can be performed within the stated time bounds.

An application of P is just a rational permutation; the bounds follow from Theorem 5.1.

Let I denote the $k \times k$ identity matrix. We show how to decompose U into a product of nine matrices, each of which represents one of the following

Using an argument similar to the proof of Lemma 5.7, the result can be extended to non-homogeneous linear permutations.

Corollary 5.10.

Let Π be a linear permutation. Π can be performed conservatively on $n = 2^k$ consecutive locations in shared memory in time

- $O(n/p + n \log \min(p, l)/(p \log(2n/pl)))$ if $pl \leq n$;
- $O(l + l \log \min(p, l)/ \log(2pl/n))$ if $pl > n$. $\square$

5.1.4 Arbitrary permutations

Theorem 5.11. [5]

Any permutation Π can be performed conservatively on n consecutive locations in shared memory in time

- $O(\min(nl/p, n \log p/(p \log(2n/pl))))$ if $pl \leq n$;
- $O(\min(nl/p, l \log(n/l) \log \min(p, l)/(\log l \log(2pl/n)))$ if $pl > n$.

Proof sketch.

Each processor can move $\lceil n/p \rceil$ elements one at a time within time $O(nl/p)$.

For the nontrivial bound, we recall that any permutation of the entries of a matrix can be performed in three phases by executing permutations on the rows, the columns, and the rows again. For the case $pl \leq n$, we take the n items to be a $p \times n/p$ matrix stored in row-major order. For the case $pl > n$, we use a $n/l \times l$ matrix. Each permutation on the rows can be performed in time $O(n/p + l)$. To perform the permutation on the columns, we transpose the matrix between phases and apply the algorithm recursively in parallel. $\square$

The following lower bound shows that for almost all permutations, the above algorithm is optimal in the case $pl \leq n$.

Theorem 5.12. [5]

There is a permutation Π on n elements such that any conservative algorithm for performing Π on n consecutive locations of shared memory requires time $\Omega(\min(nl/p, n \log p/(p \log(2n/pl))))$ for $pl \leq n$. $\square$

For our purposes we will require the following slightly stronger version of the theorem.

Theorem 5.13.
Let S_n be the set of all permutations on $\{0, 1, \ldots, n-1\}$. Then there is a subset $E \subseteq S_n$ with $\log |E| = o(n \log n)$ such that for all $\Pi \in S_n \backslash E$, any conservative algorithm for performing Π on any n locations in shared memory requires time $\Omega(\min(nl/p, n \log p/(p \log(2n/pl))))$ for $pl \leq n$. □

The proof of this stronger version is easily obtained from the original one given in [5], if we note that:

• The lower bound can be applied to permutations of any n locations in shared memory, not necessarily consecutive. This can be seen by generalizing the notion of a "segment" to be a contiguous block of memory containing l of the elements to be permuted.

• The number of "easy" permutations can be bounded. This is because the proof is a counting argument bounding the number of different permutations that can be attained by a Block PRAM within a given time bound.

5.2 Balanced trees

The *balanced binary tree* is a fundamental structure in parallel computation:

• Consider the problem of adding n numbers. The numbers can be placed at the leaves of a balanced binary tree, and the additions can be performed by parallel computation as follows. At each level of the tree, numbers are added together in pairs by different processors in parallel and communicated to the next level. If there is no communication latency, the computation can be performed by n processors in the same time as $O(\log n)$ sequential binary additions.

• Consider the problem of broadcasting a datum to p processors. The processors can be placed at the leaves of a balanced binary tree, and the broadcasting can be performed by parallel computation as follows. At each level of the tree, the datum is copied to both nodes on the next level. If there is no communication latency, the computation can be performed by p processors in the same time as $O(\log p)$ single copying steps.

When there is substantial communication latency, the balanced binary tree is no longer the natural structure for these parallel computations. Because each level of the tree represents a communication step, it is desirable to use a tree with fewer levels. For many computations with communication latency l, the balanced l-ary tree is the best choice. For n leaves, a balanced l-ary tree gives computation time $O(l \log n / \log l)$, improving the time of $O(l \log n)$ for the binary tree. We illustrate this technique in this section with important algorithms and lower bounds for the Block PRAM.

5.2.1 A general lower bound

A function $f(x_1, \ldots, x_n)$ is *sensitive on all its variables* if there is a data instance $x_1 = a_1, \ldots, x_n = a_n$ such that for each i, $1 \leq i \leq n$, there is a b_i with $f(a_1 \ldots, a_i, \ldots, a_n) \neq f(a_1, \ldots, b_i, \ldots, a_n)$. The following result is stated without proof in [5]. We supply a proof in the spirit of [179].

Theorem 5.14. [5]
Let $f(x_1, \ldots, x_n)$ be sensitive on all its variables. Then any Block PRAM algorithm computing f requires time $\Omega(n/p + l \log n / \log l)$.

Proof.
Let $\mathcal{W}$ be a Block PRAM algorithm that computes $f(A)$ for the array A consisting of the first n locations of shared memory. We can make the following simplifications while increasing the running time of $\mathcal{W}$ by at most a constant factor: the algorithm executes in alternating *read rounds* and *write rounds*. During a read round each processor may read at most l locations of shared memory; during a write round each processor may write into at most l locations of shared memory. Each round takes time at least l.

We use the following data dependency argument.[2] After each round, to each shared memory location and processor we assign a subset of $\{0, \ldots, n-1\}$ corresponding to the entries of A it has "sensed." At termination, the shared memory location containing $f(A)$ must have "sensed" all n entries of A, and the corresponding set must have cardinality n.

More formally, let the shared memory have $N \geq n$ (initialized) locations. We define $R(i, r)$, $P(j, r)$ inductively for $1 \leq i \leq n$, $1 \leq j \leq p$, $r \geq 0$

[2]Also called a "fan-in" or "fan-out" argument.

as follows. As the algorithm begins, let $R(i,0) \leftarrow \{i\}$ for $0 \leq i \leq n-1$, $R(i,0) \leftarrow \emptyset$ for $n \leq i \leq N-1$ and $P(j,0) \leftarrow \emptyset$ for $1 \leq j \leq p$. During the r-th read round, for $r \geq 1$ and $1 \leq j \leq p$, let $P(j,r) \leftarrow P(j,r-1) \cup \bigcup\{R(i,r-1)$: Processor j reads memory location i during the r-th read round$\}$. During the r-th write round, for $r \geq 1$ and $0 \leq i \leq N-1$ let $R(i,r) \leftarrow P(j,r-1)$ if some processor j writes into memory location i during the write round; otherwise let $R(i,r) \leftarrow R(i,r-1)$. It is easy to verify that after the r-th write round, memory location i is a well-defined function on the variables $\{A(j) : j \in R(i,r)\}$.

Let $M(r) = \max_i\{|R(i,r)|\}$. Since at most l locations of global memory can be accessed during a round, $M(1) \leq l+1$ and $M(r) \leq (l+1) \cdot M(r-1)$. Since the output value must be sensitive on n variables, the algorithm $\mathcal{W}$ can terminate only when $M(r) \geq n$; hence at least $\log n / \log(l+1)$ write rounds are required.

Finally, we observe that computing any nondegenerate function on n variables requires time $\Omega(n/p)$. The theorem follows. □

5.2.2 Prefix sums computations

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

This problem has a trivial linear-time sequential algorithm and a well known $O(n/p + \log n)$ time EREW PRAM algorithm [63, 112, 119]. A prefix addition algorithm for the CRCW PRAM model with $O(\log p)$ word size runs in time $O(n/p + \log n / \log\log n)$ [49, 170].

The prefix addition computation is sensitive on all its variables since the sum of all the elements in the array A must be computed, so that the lower bound of Theorem 5.14 applies. We now give an optimal Block PRAM algorithm for performing the prefix sums computation.

Algorithm PREFIX $(A, n, \oplus, S)$

Input: Array A of length n.

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

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

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

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

time. The overall time is $O(n/p + l \log p/ \log l) = O(n/p + l \log n/ \log l)$.

Theorem 5.15.

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

In his book on vector models of parallel computation, Blelloch [31] introduces the *scan vector model* of parallel computation, a variant of the EREW PRAM in which prefix sums computations on up to p inputs are assumed to be primitive operations which can be implemented in unit time. He argues the practicality of the scan vector model by noting the low Boolean function complexity [119] and the relatively low empirical running time [91] of prefix sums computations.

Blelloch catalogs a number of problems in graph theory, static networks and computational geometry for which there are parallel algorithms in the scan vector model which run asymptotically faster than the corresponding benchmark EREW PRAM algorithms. In particular, scan vector model algorithms for connected components, biconnected components, minimum spanning tree, maximum flow, maximum independent set, logic simulation, neural networks, convex hull, building a k-d tree, closest pair in the plane and line of sight all run $O(\log n)$ times faster than on the EREW PRAM model. Since by Theorem 5.15 the cost of adapting a scan vector algorithm to the Block PRAM model using the procedure PREFIX is $O(l \log n/ \log l)$, it follows that the scan vector model algorithms run at most $O(l/ \log l)$ times slower on a Block PRAM than on an EREW PRAM. This leads us to suggest the Block PRAM model as a basis in PRAM theory for the specific study of prefix sums-based parallel algorithms, beyond the empirical justifications given by Blelloch in [31].

5.2.3 Integer sorting

We conclude our discussion of prefix sums with a simple example of a prefix sums-based parallel algorithm, a Block PRAM algorithm for integer sorting.

Let A be an array of n numbers, or *keys,* in the range $\{1, 2, \ldots, n\}$. The *integer sorting* problem is to produce an array A' whose elements are the keys arranged in nondecreasing order. The *stable integer sorting problem* asks us to produce also a permutation π such that $A'(i) = A(\pi(i))$; and $\pi(i) < \pi(j)$ if $A'(i) = A'(j)$ and $i < j$.

A deterministic sequential algorithm for stable integer sorting taking $O(n)$ time is well known [7], as is a deterministic EREW PRAM algorithm taking time $O(n \log n/p + \log n)$ [93]. Many CRCW PRAM algorithms have been developed recently, including:

- a deterministic Common CRCW PRAM algorithm taking worst-case time $O(n \log\log n/p + \log n/\log\log n)$ [30];
- a randomized Arbitrary CRCW PRAM algorithm taking expected time $O(n/p + \log n/\log\log n)$ [82];
- a randomized Arbitrary CRCW PRAM algorithm which outputs the sorted integers as a linked list (see Section 5.3) in expected time $O(n/p + \log\log n \log^* n/\log\log\log n)$ [82]; and
- randomized algorithms for sorting n integers in the range $\{1, 2, \ldots, m\}$ taking expected time $O(n(\log\log m)^2/p + (\log\log m)^2)$ for the Arbitrary CRCW PRAM and $O(n \log m \log\log m/p + \log n + \log\log m)$ for the Common CRCW PRAM [139].

Specific algorithms for the integer sorting problem work by moving each key into a group of cells *(bucket)* according to its place values *(radices)* when expressed in some base system. Hence integer sorting is sometimes referred to as *bucket sorting* or *radix sorting.*

In the general sorting problem, the keys are not assumed to have any numerical value, and must be ordered according to comparisons among the keys. This makes the problem more difficult: the sequential complexity of general sorting is $\Theta(n \log n)$.

We give results for both integer sorting and general sorting in this chapter. In this section, we give a deterministic Block PRAM algorithm for integer sorting. For our Block PRAM model with $O(\log p)$ word size, we could express characters and floating-point numbers as integers and use the integer sorting algorithm for all purposes. Clearly, this can be done in practice. However, in this thesis we will use the integer sorting only for its stated purpose. An upper bound on the Block PRAM complexity of general sorting is given in Section 5.8.

Algorithm INTEGER SORT (A, A', n, π)

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

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

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

We assume $x \leq p \leq n^2$; with fewer processors we can appeal to Brent's Theorem (Theorem 2.1), which allows us to trade time for processors. We cannot use more than n^2 processors efficiently. Let $m = \lfloor p/x \rfloor$, the number of processors available for each bucket. Let $R = \lceil \log n / \log x \rceil$, the number of rounds in the algorithm.

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

```
0     for all $i$, $0 \leq i < p$ do in parallel
          for $k \leftarrow 0$ until $\lceil n/p \rceil - 1$ do
              $\pi(i \cdot \lceil n/p \rceil + k) \leftarrow i \cdot \lceil n/p \rceil + k$
              $A'(i \cdot \lceil n/p \rceil + k) \leftarrow A(i \cdot \lceil n/p \rceil + k)$
      for $r \leftarrow 1$ until $R$ repeat steps 1–8
1     for all $i, j$, $0 \leq i < x$, $0 \leq j < m$ do in parallel
          for $k \leftarrow 0$ until $\lceil n/m \rceil - 1$ do
              $B(i, j \cdot \lceil n/m \rceil + k) \leftarrow 0$
2     for all $i$, $0 \leq i < x$ do in parallel
          for $k \leftarrow 0$ until $\lceil n/p \rceil - 1$ do
              $radix \leftarrow A'(i \cdot \lceil n/p \rceil + 1) - x^{R-r} \cdot \lfloor A'(i \cdot \lceil n/p \rceil + k)/x^r \rfloor$
              $radix \leftarrow \lfloor radix/x^{r-1} \rfloor$
              $B(radix, i \cdot \lceil n/p \rceil + k) \leftarrow 1$
```

```
3       for all $i$, $0 \le i < x$ do in parallel
            PREFIX $(B_i, n, +, S_i)$
4       for all $i$, $0 \le i < x$ do in parallel
            $T(i) \leftarrow S_i(i-1)$
5       PREFIX $(T, x, +, G)$
6       for all $i$, $0 \le i < x$ do in parallel
            $C(i,0) \leftarrow G(i-1)$
            PREFIX $(C_i, m, \mathbf{copy}, C'_i)$
7       for all $i, j$, $0 \le i < x$, $0 \le j < m$ do in parallel
            for $k \leftarrow 0$ until $\lceil n/m \rceil - 1$ do
                if $B(i, j \cdot \lceil n/m \rceil + k) = 1$ then
                    $rank \leftarrow S_i(j \cdot \lceil n/m \rceil + k) + C'_i(j)$
                    $\rho(rank) \leftarrow j \cdot \lceil n/m \rceil + k$
8       for all $i$, $0 \le i < p$ do in parallel
            for $k \leftarrow 0$ until $\lceil n/p \rceil - 1$ do
                $\pi(i \cdot \lceil n/p \rceil + k) \leftarrow \pi(\rho(i \cdot \lceil n/p \rceil + k))$
                $A'(i \cdot \lceil n/p \rceil + k) \leftarrow A(\pi(i \cdot \lceil n/p \rceil + k))$
```

Analysis: The initialization step 0 takes $O(n/p + l)$ time. During each round, steps 1 and 7 take time $O(n/m + l)$, steps 2 and 8 take time $O(nl/p)$, step 3 takes time $O(n/m + l \log n / \log l)$, step 4 takes time $O(l)$, step 5 takes time $O(l \log x / \log l)$, and step 6 takes time $O(l \log m / \log l)$. The overall time complexity is $O((\log n / \log x) \cdot (n(x + l + \log x)/p + l \log n / \log l))$. By setting $x = 2^{\lceil \log(l + (pl \log n)/(n \log l)) \rceil}$ we have

Theorem 5.16.

n integers in the range $\{1, 2, \ldots, n\}$ can be sorted stably in time

- $O(ln \log n / (p \log l) + l \log^2 n / \log^2 l)$ for $p \le n$;
- $O(l \log^2 n / (\log l \log 2(pl/n)))$ for $n \le p \le n^2$. □

Corollary 5.17.

(1) n integers in the range $\{1, 2, \ldots, n\}$ can be sorted stably in time $O(l \log^2 n / \log^2 l)$ on $n \log l / \log n$ processors.

(2) Let $\epsilon > 0$ be a constant. Then n integers in the range $\{1, 2, \ldots, n\}$ can be sorted stably in time $O(l \log n / \log l)$ on $n^{1+\epsilon}$ processors. □

5.3 Linked lists

The *linked list* is an alternative to the array in storing sequences of elements in memory. Instead of forming an orderly block of consecutive memory locations, the elements in a linked list can appear anywhere in memory. Stored with each element is a *pointer* giving the address of the next element of the sequence. Arrays are preferable to linked lists when it is necessary to find the k-th element of a sequence. However, linked lists are preferable to arrays when it is necessary to perform many insertion and deletion operations.

Definition.

On the Block PRAM model, a *linked list* of n elements consists of two arrays $A[1 \ldots N]$ and $S[1 \ldots N]$ of size $N \geq n$. A is called the *data array* and S is called the *pointer array.* The first element in the list, called the *head*, is stored in $A(1)$. For $1 \leq i \leq n-1$, if the i-th element of the list is stored in $A(j)$, then the $(i+1)$-st element of the list is stored in $A(S(j))$. That is, the $(i+1)$-st element of the list is stored in $A(S^{(i)}(1))$. The values of $S^{(i)}(1)$, $1 \leq i \leq n-1$, are distinct integers in the range $\{2, 3, \ldots, N\}$. By convention, $S^{(n)}(1) = 0$.

Note that only n of the N locations in each array are defined; the others do not need to be initialized. However, it is frequently assumed in definitions of linked lists that $N = n$ (e.g. see [48, 84]). In such a case we say that the linked list is *compact.*

Definition.

Given a linked list (A, S), the *list ranking* problem is to compute the equivalent array B. That is, compute $B[1 \ldots n]$ such that $B(i) = A(S^{(i-1)}(1))$. Equivalently, compute elements in an array of *distances* $D[1 \ldots N]$ such that $A(i) = A(S^{(n-D(i)-1)}(1))$ for each element $A(i)$ of the list.

The problem restricted to the case $N = n$ is called the *compact list ranking problem.*

The list ranking problem has a trivial linear time serial algorithm. We do not know of an efficient parallel algorithm for general list ranking. However, for compact list ranking, there is a standard $O(n \log n/p + \log n)$ time

EREW PRAM algorithm [206]. Optimal $O(n/p+\log n)$ time EREW PRAM algorithms are also known [18, 51]. The latter algorithms are "rather elaborate" [104], and [18] concedes that the standard algorithm "is still probably the best deterministic algorithm" in practice. Trading determinism for simplicity and lower constants, randomized EREW PRAM algorithms taking time $O(n/p + \log n)$ have recently appeared [20, 104].

The basic step in all list ranking algorithms known to the author is *shortcutting* individual elements from the linked list. To shortcut an element $A(i)$, $A(i)$ is removed from the list and the pointer $S(S^{-1}(i))$ of its predecessor $A(S^{-1}(i))$ is updated, or "jumped," to its successor $S(i)$.[3] The distance computation follows each pointer jumping step: if $S(i) \neq 0$ then $D(i)$ is updated to $D(i) + D(S(i))$.[4]

Algorithm Standard Parallel List Ranking

Comment: $L(i) = 1$ until $A(i)$ has been removed from the list, when $L(i) \leftarrow 0$.

```
1    for all $i$, $1 \le i \le n$ do in parallel
         $L(i) \leftarrow i$
         if $S(i) \neq 0$ then $D(i) \leftarrow 1$ else $D(i) \leftarrow 0$
2    for $r \leftarrow 1$ until $\lceil \log n \rceil$ do
         for all $i$, $1 \le i \le n$ do in parallel
             if $S(i) = 0$ then $L(i) \leftarrow 0$
             if $L(i) = 1$ then
                 $D(i) \leftarrow D(i) + D(S(i))$
                 $S(i) \leftarrow S(S(i))$
```

Elements can be shortcut in parallel, provided that no two adjacent elements are shortcut simultaneously. Finding a large set of nonadjacent elements to shortcut—*symmetry breaking*—is a nontrivial problem, and the improved deterministic algorithms in the literature have used the parallel symmetry-breaking technique of Cole and Vishkin, called *deterministic coin tossing* [48, 77].

[3]In this section, we frequently view arrays as invertible functions: $S^{-1}(i)$ denotes the value x, $1 \le x \le N$, such that $S(x) = i$.

[4]Pointer jumping is also frequently called "pointer chasing" and "shunting."

The sequential list ranking algorithm can be viewed as shortcutting the elements one at a time from the head of the list. It is observed in [18] that the standard parallel algorithm is inefficient because it shortcuts the same element from a number of different lists instead of leaving it alone once it has been removed: this technique is called *recursive doubling* (see e.g. [75, 127]). The optimal parallel algorithms address this inefficiency by shortcutting each element exactly once.

For $lp \leq n$, we prove an $\Omega(\min(ln/p, (n \log p)/(p \log(2n/lp))))$ lower bound on the time for a Block PRAM to perform list ranking by shortcutting. The proof uses a reduction to arbitrary permutations (Theorem 5.13) as follows. Assume there is a fast Block PRAM algorithm for list ranking. Consider a "hard" permutation. We apply the algorithm to a linked list whose pointers are the actions of the permutation. By observing how the shortcutting takes place, we derive an impossibly fast conservative algorithm for performing the permutation on any segment of memory.

Definition.

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

- For each element $A(i)$, some processor executes a *shortcut* step:

```
        $S(S^{-1}(i)) \leftarrow S(i)$
        REMOVE $A(i)$
```

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

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

Theorem 5.18.

Any shortcut-based Block PRAM algorithm for compact list ranking requires time $\Omega(\min(nl/p, n \log p/(p \log(2n/pl))))$ for $pl \leq n$. □

Proof.

Let $\mathcal{W}$ be a Block PRAM algorithm which performs list ranking by shortcutting within time $T = T(n, l, p)$. Consider an n-cycle Π on $\{1, \ldots, n\}$ such that $\Pi \in S_n \backslash E$, where E is the set of "easy" permutations in Theorem 5.13. (There are $(n-1)! = 2^{\Omega(n \log n)}$ n-cycles, so that almost all of them are "hard.") Given an array M, we give a conservative algorithm $\mathcal{V}$ which performs Π on the elements of M in time bounded by $3T$.

Let $L = (A, S)$ be a compact linked list whose pointers are given by the actions of Π; i.e., $S(i) = \Pi(i)$ for $1 \leq i \leq n$ unless $\Pi(i) = 1$. By convention, we define $S(\Pi^{-1}(i)) = 0$.

Execute $\mathcal{W}$ on the problem instance L. Since $\mathcal{W}$ is shortcut-based, each of the list elements $A(2), A(3), \ldots, A(n)$ must be shortcut from L. To shortcut $A(i)$, some processor must have exclusive access to each of $S(S^{-1}(i))$ and $S(i)$.

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

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

These procedures comprise the entire algorithm $\mathcal{V}$. We check that $\mathcal{V}$ is well-defined. Clearly each exchange procedure requires at most three times as long as the corresponding shortcutting procedure, regardless of how global memory accesses are pipelined, so that the exchanges can be performed at the scheduled times and in the correct sequence.

The action of $\mathcal{V}$ on M may be written as a product of 2-cycles. Exchanges which are performed in parallel correspond to shortcutting nonadjacent elements of L and, therefore, to disjoint 2-cycles, which commute. Hence no conflicts in access to global memory are created, and the action of $\mathcal{V}$ is well-defined. It should be noted also that $\mathcal{V}$ is obviously a conservative algorithm.

Finally, we establish that $\mathcal{V}$ performs the permutation Π on the elements of M. Let M_0, A_0 be the original input arrays to algorithms $\mathcal{V}$ and $\mathcal{W}$, respectively. Then it is easy to check that the invariant "For each i such that $A(i)$ is in the list, $M(i) = M_0(A_0^{-1}(A(i)))$" remains true throughout the reduction. Now, when any element $A(i)$ is shortcut in $\mathcal{W}$, the corresponding memory exchange in $\mathcal{V}$ places the value of $M(i)$ into the $A(i)$-th location of M; this value is never moved again. But by the invariant, this is the same value that the permutation Π would have placed into the location. The algorithm $\mathcal{V}$ terminates within time $3T$, as required. □

It is interesting to observe the 1-1 correspondence between factorizations of Π into 2-cycles and possible sequences of shortcutting steps in reducing L.

Corollary 5.19.

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

Proof.

The general list ranking problem reduces to the problem of performing arbitrary n-cycles conservatively on some (not necessarily contiguous) set of n locations $M(i_1), \ldots, M(i_n)$ in an array $M[1 \ldots N]$. □

We observe that compact list ranking is finely granular for $l = O(\log p)$, $p = n/\log n$.

Corollary 5.20.

Assuming shortcut-based algorithms, the complexity of the problem of ranking a compact list of length n on $p = n/\log n$ processors is $\Theta(l \log n)$ for $l = O(\log p)$.

Proof.

The upper bound follows immediately from any of the optimal $O(n/p + \log n)$ EREW PRAM algorithms for compact list ranking [18, 51], since all of them are shortcut-based. □

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

• In a discussion paper titled "Are pointer-based parallel algorithms realistic?," Miller [142] recognizes a growing gap between the body of efficient PRAM algorithms, and the real machines being designed to implement them. On the Connection Machine, a single shortcutting step takes about 1000 times as long as a single local instruction step. Miller concludes: "I hope that designers of future general purpose parallel machines will consider the list ranking problem when they design their machines."

• Leiserson and Maggs [127] point out that recursive doubling can lead to congestion because all of the active pointers propagate toward the end of the list. This quickly results in too many processors attempting to access pairs of earlier and later pointers. However, they also show that this problem can be eliminated by symmetry breaking (e.g. deterministic coin tossing) and shortcutting each pointer only once.

• Gazit, Miller and Teng [71] offer a critical comparison between prefix sums and list ranking. They observe that on a hypercube machine with $p = n/\log n$ processors, prefix sums can be computed in time $6 \log n$, but the list ranking algorithms in [18, 20, 51] seem to have running time $O(\log^2 n)$. They conclude that "if the ultimate purpose of a parallel algorithm is to run it on a fixed connection machine, then we should minimize the number of list rankings we perform; and, whenever, possible, replace the list ranking procedure with the prefix sums procedure." However, they do not justify this conclusion by proving a $\omega(n/p + \log n)$ lower bound for list ranking on the hypercube.

Taken together, our results for prefix sums computations and list ranking give the first rigorous theoretical justification for these emerging practical concerns about the difficulty of pointer jumping. As a shortcut-based procedure, list ranking represents an asymptotic bottleneck in many Block PRAM computations.[5]

One way to overcome the list ranking bottleneck is to exploit the similarity between list ranking and prefix sums computations (observed e.g. in [104, 138]). It is possible, although not immediately obvious, that we can replace list ranking with prefix sums to obtain asymptotic improvements in complexity.

• Gazit, Miller and Teng [71] give an algorithm for tree contraction (see Section 5.4) which replaces a constant proportion of the list ranking operations with prefix sums. This has no overall effect on the algorithm's asymptotic complexity in any model.

• Prefix and list ranking computations have been used interchangeably

[5]Hagerup's algorithm for integer sorting which outputs a linked list (see Section 4.2.3 and [82]) suggests that list ranking currently also represents an asymptotic bottleneck in PRAM computations.

for very simple procedures such as finding the minimum of n values [75, 138]. When implementing such procedures on the Block PRAM model, prefix sums computations should be used.

• More substantively, we demonstrate in Section 5.5 that we can choose prefix sums over list ranking as the basic step for a Block PRAM algorithm to find the connected components of dense graphs, with an asymptotic savings in complexity.

A more general solution to the list ranking bottleneck is to allow arbitrary pipelining. A Block PRAM model allowing arbitrary pipelining is introduced by Gibbons [76], and Gibbons's EREW Phase LPRAM list ranking algorithm can be easily adapted to run on such a model in $O(n/p + l \log n / \log l)$ time. (See Chapter 8 for this result, and a detailed comparison between these models.) That is, in hoping for hardware support for the list ranking procedure, Miller [142] is essentially asking for the capability of arbitrary pipelining. The absence of such support strengthens the case for studying a block-hierarchical model of parallel computation.

5.4 Expression trees

Perhaps the most obvious application of computing machinery is to long chains of arithmetic calculations. Today the youngest schoolchild is familiar with the value of electronic calculators in adding long columns of numbers. The evaluation of arithmetic expressions is a popular benchmark problem for describing computing power.

In parallel expression evaluation, it is helpful to use a data structure which identifies the dependencies between the various subexpressions during the evaluation. An *expression tree* is such a data structure.

Definition.

A *rooted tree* $T = (V, E, r)$ is a connected directed acyclic graph in which the root r has outdegree 0 and the other vertices have outdegree 1. The leaves of a tree are those vertices having indegree 0; the other vertices are called *internal vertices.* If the leaves of a tree are labeled with constants (words) and the internal vertices are labeled with basic arithmetic instructions, the resulting data structure is an *expression tree.*

We define the *value* of an expression tree inductively as follows. The value of a leaf is the value of its label. The value of an internal vertex is the value of its arithmetic instruction applied to its inputs. The value of an expression tree is the value of its root. If the operation at an internal vertex is not commutative, the inputs to that vertex must be ordered in the specification of the expression tree.

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

Theorem 5.21. [26]

Given an array representing an arithmetic expression of length n with operations $+, -, \times, /$ and brackets, the corresponding expression tree can be computed on an EREW PRAM in $O(n/p + \log n)$ time. $\square$

Theorem 5.22. [1, 50, 71, 74, 110]

An expression tree with n vertices can be evaluated on an EREW PRAM in $O(n/p + \log n)$ time. $\square$

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

Proposition 5.23.

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

Proof.

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

Corollary 5.24.

Assuming shortcut-based algorithms, for $l = O(\log p)$, the complexity of the problem of contracting an expression tree with n vertices on $p = n/\log n$ processors is $\Theta(l \log n)$ for $l = O(\log p)$. □

5.5 Graphs

By *graphs* in this section we will mean simple undirected graphs $G = (V, E)$, where V is a nonempty set of n vertices and E is a set of m edges. G is *connected* if there exists a path between every pair of distinct vertices in V. A *connected component* of G is a maximal connected subgraph of G. G is *dense* if $m = \Theta(n^2)$ and *sparse* otherwise.

Connectivity is a fundamental property of graphs. The problem of finding connected components is a key subproblem to the problems of testing a graph for planarity, biconnectivity and triconnectivity, and of finding a minimum spanning forest and an ear decomposition of a graph. It is also an important problem in its own right.

Connected components can be found sequentially by depth-first search in time $O(m + n)$ (i.e., $O(n^2)$ for dense graphs). The best known PRAM algorithms for finding connected components are as follows:

- an ARBITRARY CRCW PRAM algorithm taking $O((m+n)\alpha(m, n)/p + \log n)$ time;[6]
- a CREW PRAM algorithm taking $O(n^2/p + \log^2 n)$ time [47, 194]; and
- an EREW PRAM algorithm taking $O(((m + n) \log n)/p + \log^2 n)$ time [111].

Note that the CREW PRAM algorithm is optimal for dense graphs and the CRCW PRAM algorithm is almost optimal for sparse graphs. Whether there exists an optimal $O(\log n)$ time algorithm for sparse graphs remains an important open problem in the parallel complexity theory [98].

We consider two algorithms for finding connected components: the CREW PRAM algorithm of Chin, Lam and Chen [47] and the $O((m + n) \log n/p +$

[6] $\alpha(m, n)$ is the inverse Ackermann function, a function that grows so slowly that it is constant for all practical purposes.

$\log n$) time ARBITRARY CRCW PRAM algorithm of Shiloach and Vishkin [176].

The main idea of the CREW PRAM algorithm [47] is the *path compression* technique of [92]. Trees in an undirected graph are constructed, or "hooked" together, by having each vertex point to its lowest-numbered neighbor. These trees are compressed, and vertices along these paths are merged into "supervertices." The process is then repeated on the graph induced by the supervertices, then on the super-supervertices, and so on. The number of supervertices is reduced by half after each iteration, so that the algorithm is finished after $\log n$ stages.

Definition.

Let $C[1 \ldots n]$ be an array of integers with $1 \leq C(i) \leq n$ such that C induces a pointer graph containing no cycles of length ≥ 2. That is, for $k \geq 2$, there are no distinct $i_1, \ldots, i_k$ such that $C(i_1) = i_2, \ldots, C(i_{k-1}) = i_k, C(i_k) = i_1$. The *path compression* problem is to compute C^*, where $C^*(i) = C^{(n-1)}(i)$.

In the algorithms of [47, 92] the path compression problem is solved on a CREW PRAM in time $O(n \log n/p + \log n)$: for each of $\log n$ iterations, each processor performs the shortcutting operation $C(i) \leftarrow C(C(i))$. This procedure is essentially the standard parallel algorithm for list ranking where we do not bother to compute ranks. Here, concurrent reads are necessary because $C(i) = C(j)$ can occur for distinct i, j. The path compression problem can also be solved using just the pointer-jumping steps from any shortcut-based tree contraction algorithm, and there are $O(n/p + \log n)$ time EREW PRAM algorithms for this (Theorem 5.22). However, as we have seen, pointer-jumping on the Block PRAM model is finely granular.

The CRCW PRAM algorithm of [176] improves on [47] by performing the shortcutting operation only a constant number of times on each processor during each iteration. All of the vertices, not just the supervertices, share the burden of "hooking" trees together. This change creates possible write conflicts in global memory. However, in the Block PRAM model, resolving these conflicts using the PREFIX procedure takes time $O(l \log n/ \log l)$, asymptotically faster than performing $\log n$ shortcutting operations in time

$O(l \log n)$. We establish this now.

The algorithm in [176] uses $2m+n$ processors by assigning one processor to each vertex and two processors to each edge, one at each end. The vertex processors are never active at the same time as the edge processors. Among the edge processors, concurrent accesses occur only between processors corresponding to adjacent edges, and then only in accessing data associated with the common vertex. Among the vertex processors, concurrent accesses occur only at pointer jumping steps $C(i) \leftarrow C(C(i))$.

Let $\delta(v)$ denote the number of edges incident to vertex v. To provide for $\delta(v)$ concurrent reads of a datum in global memory associated with v (a *v-datum*), we maintain a block of $\delta(v)+1$ copies of it. (One copy is kept as the original.) Copies of the initial data can be produced using a prefix **copy** computation.

To provide for $\delta(v)$ concurrent writes, each processor writes to its own copy of the v-datum. After each write round, we perform three prefix sums computations: the first two to find a copy which disagrees with the original (if any), the third to make $\delta(v)+1$ copies of it. (For example, the operations **min** and **max** could be used.) By choosing different operations, we can simulate different rules for resolving concurrent writes.)

Now suppose we are performing the algorithm on a Block PRAM with p processors, where $n \leq p < m$. Then we can schedule the algorithm so that each Block PRAM processor always simulates several CRCW PRAM processors accessing the same v-datum. Then since the copies of each v-datum are stored contiguously, the accesses from each processor can be pipelined.

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

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

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

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

Analysis: Steps 1, 3, 4, 6 and 7 take $O(n/p + l)$ time. Steps 5 and 8 take $O(n/p + l \log n/\log l)$ time. Step 9 takes $O(nl/p + l)$ time. Step 2 takes $S(n, l, p)$ time, where $S(n, l, p)$ is the Block PRAM complexity of integer sorting n keys. Since the outputs of an integer sorting algorithm are sensitive on all variables, the lower bound of Theorem 5.14 applies to $S(n, l, p)$ and the overall complexity of the procedure is $O(S(n, l, p) + nl/p)$.

We have the following result:

Theorem 5.25.

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

Note that by setting $l = O(1)$ our result matches the performance of the EREW PRAM algorithm of [111] for sufficiently dense graphs with $m = \Omega(n \log n)$. The following corollary shows that our algorithm is efficient for sufficiently dense graphs with $m = ln^{1+\Omega(1)}$.

Corollary 5.26.

The connected components of a graph with n vertices and $m = ln^{1+\Omega(1)}$ edges can be found in time $O(m \log n/p + l \log^2 n/\log l)$. □

5.6 Strings

Let $P[1 \ldots m]$ and $T[1 \ldots n]$ be arrays with $m \leq n$. P is the *pattern string* and T is the *text string*. The *string matching* problem is to produce an array $M[1 \ldots n - m + 1]$ such that $M(i) = 1$ if $P[1 \ldots m] = T[i \ldots i + m - 1]$ and

$M(i) = 0$ otherwise.

String matching is an important problem in text and database processing, with applications to "pattern recognition and computer vision, molecular sequence analysis, data compression and encoding, speech synthesis, [and] text retrieval and editing" [95].

$O(n)$ time sequential algorithms for string matching have been known as early as 1977 [69, 108]. Optimal $O(n/p + \log n)$ deterministic [195] and randomized [103] EREW PRAM algorithms are also known.

In this section, we show that the randomized algorithm of [103] matches the deterministic lower bound for the Block PRAM model. This work formalizes the fairly obvious intuition that there is some spatial locality inherent in text processing problems, and illustrates probably the first optimal randomized Block PRAM algorithm in the literature.

Following standard PRAM algorithms, and for clarity, we will assume the strings are over a binary alphabet: $P \in (01)^m, T \in (01)^n$. A fan-out argument similar to Theorem 5.14 gives the following lower bound.

Theorem 5.27.

Any deterministic Block PRAM algorithm for string matching requires time $\Omega(n/p + l \log n / \log l)$.

Proof sketch.

Case 1: $\log m = \Theta(\log n)$. Fix an array T. Each element of the output array M is sensitive on all of the elements in P, so $\Omega(l \log m / \log l) = \Omega(l \log n / \log l)$ time is required.

Case 2: $\log m = o(\log n)$. Fix an array T. Each element of the output array M must see at least one of the elements in P, so some element of P must be copied at least $\lceil n/m \rceil$ times, and $\Omega(l \log(n/m)/\log l) = \Omega(l \log n / \log l)$ time is required.

There are instances when it is obvious that all of the text has to be read (e.g., when the pattern is all 0's), so $\Omega(n/p)$ time is required in either case. □

The randomized string matching algorithm of Karp and Rabin [103] is particularly amenable to implementation on the Block PRAM model. The idea is to associate to each string a short *fingerprint*. These fingerprints are easier to compare than the strings. The fingerprints are computed using a randomly chosen *fingerprint function* with the property that the fingerprints of any two distinct strings are very unlikely to be equal.

We consider the following family of fingerprint functions. Let $k \geq 1$ be a constant and let q be a random prime number less than n^{k+1}. The fingerprint function K_q is a homomorphism from $(01)^*$ to the group of 2×2 unimodular (determinant 1) matrices over the ring Z_q defined as follows:

$$K_q(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix};$$

$$K_q(0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix};$$

$$K_q(1) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \text{ and}$$

$K_q(X : Y) = K_q(X)K_q(Y)$;
where λ denotes the empty string and : denotes string concatenation.

Algorithm String Match (P, T, n, m, k)

```
1    choose a random prime $q \leq n^{k+1}$
2    $J \leftarrow K_q(P)$
3    for all $i$, $1 \leq i \leq n - m + 1$
         $M(i) = 0$
         $N(i) \leftarrow K_q(T[i \ldots i + m - 1])$
         if $N(i) = J$ then $M(i) = 1$
```

Theorem 5.28. [103]
The probability that Algorithm String Match gives the correct output is greater than $1 - n^{-k}$. □

In order to parallelize Algorithm String Match, Karp and Rabin suggest the following procedure for computing $N(i)$ in step 3.

Procedure COMPUTE N

Comments: $*$ is 2×2 matrix multiplication modulo q. I is the 2×2 identity matrix.

```
1    for all $i$, $1 \le i \le n - m + 1$ do in parallel
         $K(i) \leftarrow K_q(T(i))$
2    PREFIX $(K, *, n - m + 1, R)$
3    $R(0) \leftarrow I$
4    for all $i$, $1 \le i \le n - m + 1$ do in parallel
         $S(i) \leftarrow R(i)$
5    for all $i$, $1 \le i \le n - m + 1$ do in parallel
         $N(i) \leftarrow R(i-1)^{-1} * S(i + m - 1)$
```

The arithmetic operations used in the algorithm are scalar addition and multiplication, and 2×2 matrix multiplication modulo q. (The matrices in R have determinant 1 and can be inverted without divisions.) Clearly each of these operations can be performed in a constant number of instructions on a Block PRAM with multiplication and $\Omega(\log q)$ word size. Note that the word size is sufficient provided $p = n^{\Omega(k)}$. Algorithm STRING MATCH can now be implemented directly on the Block PRAM model provided that we broadcast J and q after step 2. We have the following

Theorem 5.29.

String matching between a pattern of length m and a text of length n can be performed probabilistically on a Block PRAM with multiplication in time $O(n/p + l \log n / \log l)$ with probability of error less than n^{-k}, provided $p = n^{\Omega(k)}$. □

5.7 Butterflies

Definition.

For $n = 2^k$, the *n-input butterfly network* consists of $1 + \log n$ *levels,* each containing n vertices. The vertices in the j-th level of the butterfly, $1 \le j \le k$, are denoted by (i, j), where $i \in [0 \ldots n-1] \cong (01)^k$. The edges of the butterfly are between consecutive levels: from (i, j) there are edges to $(i, j+1)$ and $(i \oplus e_j, j+1)$ for $1 \le j < k$.

Like the balanced binary tree, the butterfly is a fundamental structure in parallel computation. Besides being one of the most thoroughly studied network topologies proposed for general purpose parallel computer architectures (e.g. see [169]), the butterfly is the natural communications architecture for important special purpose applications.

• Consider the problem of computing the *discrete Fourier transform.* Given an array $A[0 \ldots n-1]$ of real numbers, its discrete Fourier transform is the array $B[0 \ldots n-1]$ of complex numbers, where $B(j) = \sum_{k=0}^{n-1} A(k)e^{2\pi ijk/n}$ and $i = (-1)^{1/2}$. The butterfly is the data-dependency graph for the fastest known sequential algorithm for this problem, the *Fast Fourier Transform* of Cooley and Tukey [54], which takes $O(n \log n)$ time. The computation can be performed in parallel as follows. Each of n processors simulates a line of nodes, one node in each level of the butterfly. At each level, each processor evaluates a constant time function on its two inputs and communicates the result to the two outputs at the next level. If there is no communication latency, the computation can be performed in the same time as $O(\log n)$ sequential arithmetic operations.

• Consider the problem of sorting a *bitonic sequence* on a network of *comparators.* Array $A[1 \ldots n]$ is bitonic if there is a k, $1 \leq k \leq n$, such that $A(1), A(2), \ldots, A(k)$ is nondecreasing and $A(k), A(k+1), \ldots, A(n)$ is nonincreasing. A comparator is a node which has two input wires and two distinguished output wires H, L. Given two inputs x, y, a comparator outputs $\max(x, y)$ on wire H and $\min(x, y)$ on wire L. The butterfly is the graph representing the optimal comparator network for bitonic sorting due to Batcher [27]. The sorting network can be computed in parallel as follows. At each level of the graph, each of $n/2$ processors compares its two inputs and outputs them to the next level in sorted order.

Where there is substantial communication latency, the $\log n$-level butterfly is no longer the natural structure for these parallel computations. Because each level of the butterfly represents a communication step, it is desirable to view the butterfly as having fewer, wider levels. In this section we show that for many computations with p processors and communication latency l, a version of the butterfly with $\Theta(\log n / \log(n/p + l))$ levels is the best choice. For n processors, this butterfly gives computation time

$O(l \log n / \log l)$, improving the time of $O(l \log n)$ for the $\log n$-level butterfly. It is instructive to compare the work in this section with our results for the binary tree in Section 5.2.

5.7.1 Properties of the butterfly

The butterfly has many well-known properties which make it a suitable architecture for recursive parallel algorithms.

- The n-input butterfly may be derived from n $(2n-1)$-vertex balanced binary trees by identifying twice as many vertices at each level recursively from the roots to the leaves.
- The n-input butterfly may be derived from the n-vertex hypercube by subdividing each hypercube vertex into one vertex at each level of the butterfly and partitioning the edges into levels according to their dimensions.
- For $1 \leq m \leq k$, any m consecutive levels of the n-input butterfly comprise 2^{k-m} independent copies of the butterfly on 2^{m-1} input vertices.

Note that the n-input butterfly has $\Omega(n \log n)$ vertices while its binary tree and hypercube relatives have only n vertices. Given n processors, it may be inefficient to perform algorithms designed for the $\Omega(n \log n)$-vertex butterfly when binary tree and/or hypercube algorithms are available. However, for an important class of computations on the butterfly, the number of vertices required can be reduced from $\Omega(n \log n)$ to n.

Definition. [124]

A *normal* computation on the n-input butterfly is a synchronous parallel computation in which:

- each of n processors simulates a line of vertices, one vertex in each level of the graph;
- each vertex represents a computation consisting of a constant number of basic local operations;
- the processors simulate only one level of vertices at a time (the computation can begin and end at any level);
- the levels simulated in consecutive steps, if different, are consecutive (either forward or backward);

- communication at any given time occurs only between pairs of processors connected by edges in the butterfly between the consecutive levels being simulated.

Clearly, the Fast Fourier Transform and bitonic sorting algorithms described in the beginning of this section are examples of normal computations.

Theorem 5.30.
Any normal computation on an n-input butterfly can be implemented on a butterfly with n vertices with only a constant factor slowdown. $\square$

Note that the number of vertices cannot be reduced *ad absurdum* because the resulting algorithm on n vertices is not normal in general. Also, some important butterfly algorithms are not normal, e.g. the recent randomized sorting algorithm of Leighton and Plaxton [162].

5.7.2 The simulation

A Block PRAM algorithm for computing the Fast Fourier Transform is given in [5].

Theorem 5.31. [5]
The Fast Fourier Transform on n values can be computed on the Block PRAM with multiplication as a basic operation in time

- $O(n \log n/p)$ for $lp \leq n$;
- $O(n \log n/p + l \log n \log \min(l,p)/(\log l \log(2lp/n))$ for $lp > n$. $\square$

This result can be generalized to any normal computation on the butterfly, as we now show.

Theorem 5.32.
For $t = \Omega(\log(n/p + l))$, t steps of a normal computation on an n-input butterfly can be performed on the Block PRAM model in time

- $O(nt/p)$ for $lp \leq n$;
- $O(nt/p + lt \log \min(l,p)/(\log l \log(2lp/n))$ for $lp > n$.

Proof.

Overview. The processors communicate by accessing shared memory arrays $A[0 \ldots n-1]$ and $B[0 \ldots n-1]$. Array A (B) is unshuffled (shuffled) once for each level, allowing forward (backward) communication along edges in the butterfly.

Case 1: $lp \leq n$. The processors perform the computation one level at a time, accessing the data arrays in blocks of size l. By Theorem [4.?], A and B can be permuted between levels in time $O(n/p)$.

Case 2: $lp > n$. For $1 \leq m \leq k$, the k levels of the butterfly can be partitioned into $\lceil k/m \rceil$ *stages* each containing at most $m+1$ consecutive levels. Each stage consists of 2^{k-m} independent 2^m-input butterflies. A processor can perform m steps of a normal computation on a 2^m-input butterfly by reading the corresponding inputs from the data array, performing the computation sequentially, and writing the outputs back into the data array. For $m \geq \log(n/p)$, it follows that a stage can be computed in $O(l + m2^m)$ time.

The outputs of one stage can be permuted to provide the inputs to the next stage by using a rational permutation Π with $\mu(\Pi) = \min(m, k - m, \log l, \log p)$ and taking $O(l \min(m, k-m, \log l, \log p)/\log(2lp/n))$ time.

The computation can be divided into at most $\lceil 2t/m \rceil$ phases, such that each phase consists of at most m steps of a normal computation on exactly one of the stages of the butterfly. The theorem now follows by choosing $m = \log \max(l/\log l, n/p)$. □

5.8 Other results

In addition to the Block PRAM algorithms and lower bounds mentioned (and extended) in this chapter, Aggarwal, Chandra and Snir prove results for matrix multiplication and general sorting. We state them here for completeness.

Theorem 5.33. [5]

Two $n \times n$ matrices can be multiplied on the Block PRAM model using only addition and multiplication as basic arithmetic operations in time $O(n^3/p + l \log p/\log(lp^{2/3}/n^2))$. □

The algorithm recursively decomposes the matrices into smaller submatrices, using rational permutations to move the entries of each submatrix into consecutive locations.

Theorem 5.34. [5]
Multiplying two $n \times n$ matrices on the Block PRAM model using only addition and multiplication as arithmetic operations requires time $\Omega(n^3/p + l \log n / \log l)$. □

The lower bound uses the fan-in argument of Theorem 5.14.

Theorem 5.35. [5]
The Block PRAM can sort n words in time $O(n \log n/p + l \log n)$. □

This result is based on the Ajtai-Komlós-Szemerédi sorting network (AKS network) [10]. Although this is a strong result, it does not settle the problem of sorting for practical purposes. Even after improvements by Paterson [155], the AKS network involves such large constants that results based on its existence still have only theoretical value. Also, there is a gap of $\log l$ from the lower bound of $\Omega(l \log n / \log l)$ time given by Theorem 5.14. The stable integer sorting algorithm of Theorem 5.16 attained $O(l \log n / \log l)$ time, but at the price of an asymptotic increase in the number of processors.

Chapter 6

Hashing and skewing

The practical value of Block PRAM algorithms depends critically on the mapping of memory addresses to their physical locations. In real machines, block pipelining occurs for physically contiguous blocks of memory. When the memory is hashed or skewed, these physical blocks may not correspond to the block accesses to shared memory specified in Block PRAM algorithms. Hashing and skewing therefore affect the entire Block PRAM complexity theory. In this chapter we describe this impact by examining the Block PRAM complexity of hashing and skewing.

6.1 Universal hashing

By hashing in this chapter we shall mean *universal hashing.* In universal hashing, a hash function is chosen randomly from a *universal family* consisting of hash functions that are easy to specify and compute.

Definition.

The performance of a universal family of hash functions is expressed in terms of its *independence.* A family F of hash functions with domain D and range R is said to be $(h)_\mu$-wise independent (for $h \geq 2, \mu \geq 1$) if for all $y_1, \ldots, y_h \in R$ and all distinct $x_1, \ldots, x_h \in D$, $|\{f \in F : f(x_i) = y_i, 1, \ldots, h\}| \leq \mu|F|/|R|^h$. (When μ is omitted we understand $\mu = 1$.)

Examples.

In the following, let $N = \{0, \ldots, n-1\}$ and denote by $x \bmod m$ the residue of x modulo m.

H_1: Let r be a prime with $n \leq r \leq 2n$. Let $f_{a,b}(x) = ax+b \bmod r \bmod n$. Define $H_1 = \{f_{a,b} : a, b \in Z_r, a \neq 0\}$.

H_2: Let $n = 2^k$ and identify N with the set of 0-1 vectors of length k (most significant bit first). Let M be the set of $k \times k$ 0-1 matrices. Let $f_A(x) = Ax \bmod 2$. Define $H_2 = \{f_A : A \in M\}$.

H_3: Let n be a prime and choose an integer k. Let M be the set of arrays of length k with elements from N. Let $f_m(x) = \sum_{0 \leq i < k} m(i)x^i \bmod n$. Define $H_3 = \{f_m : m \in M\}$.

H_4: Same as H_2, but M is restricted to nonsingular matrices.

It is easy to verify that families H_1 and H_2 are 2-wise independent, while an H_3 hash family is k-wise independent. Families H_1 and H_2 were the subject of Carter and Wegman's first paper on universal hashing [37] and have been well studied in other contexts. H_3 hash families have been studied even more extensively. Because polynomials are easy to visualize and compute, they have been convenient benchmarks for specified degrees of independence.

Since $H_4 \subset H_2$ and $|H_4| \geq |H_2|/\mu$ for $\mu = e^{7/5} \doteq 4.0552$ [141], we observe that H_4 is 2_μ-wise independent. Studying the problem of distributing shared memory among a number of physical modules, Mehlhorn and Vishkin proved a further useful independence property of the family H_4.

Proposition 6.1. [141]

Let $y \in R$ and $x_1, \ldots, x_h \in D$, and let $d = \dim(x_1 - x_2, \ldots, x_1 - x_h)$. Then $|\{f \in H_4 : f(x_i) = y, i = 1, \ldots, h\}| \leq \mu |H_4|/|R|^d$, where $\mu = e^{7/5} \doteq 4.0552$. $\square$

Recently, Siegel [177] presented n^ϵ-wise independent families of hash functions which can be defined in sublinear space and computed in constant time. These families, based on expander graphs, are widely applicable in theoretical PRAM simulations, including those for the Block PRAM and

Faulty Block PRAM models (see Theorem 7.7); however, they involve constants too large for practical purposes.

Theorem 6.2. [177]

Let $0 < \delta < 1$ be fixed. Then there is an $M^{\delta^3/2}$-wise independent universal family H_5 of hash functions from $[0 \ldots M-1]$ to $[0 \ldots M-1]$ which can be defined in $O(M^{\delta})$ space and evaluated in $O(1)$ time. □

6.2 Hashing shared memory

In this section, we study the complexity of hashing on the Block PRAM model. We shall confine our attention to conservative hashing algorithms. We will not consider the complexity of evaluating addresses, choosing to focus instead on the effect of hashing on locality. This issue is of interest in several contexts.

• *Simulations between theoretical models:* In order for a Block PRAM to simulate a uniform (i.e. non-hierarchical) shared memory model such as the EREW PRAM, the memory addresses are hashed. (This simulation result is presented in Section 7.2.) The Block PRAM complexity of hashing corresponds to the minimum complexity of performing this particular simulation, and to the complexity of moving between uniform and block-hierarchical views of shared memory.

• *PRAM simulations on real hardware:* Permutations of large (length m) message packets can be performed *bit serially* by randomized routing on the hypercube [9] and butterfly and shuffle-exchange networks [126] in time $O(m + \log p)$. PRAM simulations using these routing algorithms can take advantage of block pipelining. The Block PRAM complexity of hashing corresponds to the minimum complexity of performing these simulations. See Chapter 7 for a formal discussion of this relationship.

• *Programmer-controlled hashing:* The prevailing vision of general purpose parallel computers [189] is that the network topology should be hidden, but that the programmer should retain control of memory management, including the decision whether or not to hash the shared memory. This decision should take into account the complexity of hashing, as well as the relative complexities of Block PRAM and simulated PRAM algorithms.

- *Automatic hashing:* If the shared memory is always hashed, then we will probably want to change the hash function from time to time [145, 169]. This may be necessary if a particular hash function proves ineffective in preventing contention during a given computation.
- *Input/output:* Files will read into (and out from) the shared memory in unhashed form and will need to be hashed (and unhashed).
- *Partially hashed shared memory:* In certain PRAM system designs, some of the memory address space will be hashed (uniform) and some left unhashed (block-hierarchical) [76]. The Block PRAM complexity of hashing is important in determining the possible benefits to this approach, and the cost of changing the partition.

6.2.1 Noninjective hashing

The hash functions in H_1, H_2 and H_3 are not necessarily permutations; we must formalize what it means to perform hash functions in general on the Block PRAM model. Suppose we are hashing shared memory array $A[0 \ldots n-1]$ using function $f : [0 \ldots n-1] \rightarrow [0 \ldots n-1]$. The array will be hashed into an $n \times n$ matrix B, stored in row-major order, such that for $j = 0, \ldots, n-1$, the first $|f^{-1}(j)|+1$ entries in the column vector B_j are the elements of $f^{-1}(j)$ (possibly none) in some order followed by an end marker. This generalization preserves locality considerations, and in particular, the complexity of performing permutations (within a factor of 2). The rows of the matrix correspond to successive probes in the hash table, which must take place in distinct accesses to shared memory.

6.2.2 A lower bound on universal hashing

Theorem 6.3.

Let $\mu > 0$ be a constant, let F be a universal family of 2_μ-wise independent hash functions and let f be chosen at random from F. Then performing f conservatively requires expected time

- $\Omega((n \log \min(n/l, n/p))/(p \log 2(n/lp)))$ for $lp \leq n$; and
- $\Omega(l + l \log \min(l, p)/\log 2(lp/n))$ for $lp > n$.

Proof.

For each of the cases $l = O(1)$, $l = \Omega(n)$, $p = O(1)$, $p = \Omega(n)$, the above bounds are $\Theta(n/p + l)$, so there is nothing to prove.

In the nontrivial case, we apply the potential function argument of [5] (Lemma 5.3). This time, let $A = X \cup Y$ and define segment $A_i = X[im, \ldots, (i+1)m - 1] \cup Y_{im} \cup \ldots \cup Y_{(i+1)m-1}$. The definition of m-wise potential and the proof of Lemma 5.3 extend naturally to an arbitrary hash function f.

Let $m \in \{l, p, n/l, n/p\}$. Then $m = \omega(1)$ and $m = o(n)$. We estimate $E(\Phi(f,m))$, the expected m-wise potential of f, as follows:

$$\begin{aligned}
&E(\textstyle\sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x^+_{r,s}(f,m) \cdot (x^+_{r,s}(f,m) - 1)) \\
&\quad = E(\textstyle\sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} |\{(x_1, x_2, y_1, y_2) : x_1 \neq x_2; x_1, x_2 \in A_r; \\
&\qquad y_1, y_2 \in A_s; f(x_1) = y_1; f(x_2) = y_2\}|) \\
&\quad \leq \mu m^2 \text{ (by } 2_\mu\text{-wise independence); and}
\end{aligned}$$

$$\begin{aligned}
&E(\textstyle\sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x^2_{r,s}(f,m)) \\
&\quad = E(\textstyle\sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x^+_{r,s}(f,m) \cdot (x^+_{r,s}(f,m) - 1)) \\
&\qquad + E(\textstyle\sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x^+_{r,s}(f,m)) \\
&\quad \leq \mu m^2 + n \\
&\quad = o(nm) \text{ (in the nontrivial case).}
\end{aligned}$$

Now assume for contradiction that

$$E(\Phi(f,m)) = E(\textstyle\sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x^+_{r,s}(f,m) \log x^+_{r,s}(f,m)) = \Omega(n \log m).$$

Denote $\Phi_r(f,m) = \sum_{s=0}^{n/m-1} x^+_{r,s}(f,m) \log x^+_{r,s}(f,m) \leq m \log m$. Then there are constants $c_1, c_2 > 0$ such that for at least $c_1 n/m$ of the r's, $E(\Phi_r(f,m)) \geq c_2 m \log m$. By convexity, for these r's we have $E(\sum_{s=0}^{n/m-1} x^2_{r,s}(f,m)) = \Omega(m^2)$, and therefore $E(\sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x^2_{r,s}(f,m)) = \Omega(nm)$, a contradiction.

Therefore, $E(\Phi(f,m)) = o(n \log m)$ for $m \in \{l, p, n/l, n/p\}$, and the theorem follows from Lemma 5.3. □

6.2.3 The class H_4

The class H_4 is exactly the class of homogeneous linear permutations. ¿From Theorem 5.9 we have the following result.

Theorem 6.4.

Any member of H_4 can be performed in time

- $O((n \log \min(n/l, n/p))/(p \log 2(n/lp)))$ for $lp \leq n$; and
- $O(l + l \log \min(l, p)/ \log 2(lp/n))$ for $lp \geq n$. □

6.3 Skewing

Definition.

For $u|n$, suppose that the elements of an array $A[0 \ldots n-1]$ are to be stored in n/u *modules* $M_0, \ldots, M_{n/u-1}$, or contiguous arrays of length u. Let $\mathcal{C}$ be a collection of subsets of $\{0, \ldots, n-1\}$ of size not greater than n/u. A *skewing scheme* for $(\mathcal{C}, n, u)$ is a function $f : [0 \ldots n-1] \rightarrow [0 \ldots n/u - 1]$ which assigns the elements of A to the modules such that

- each array element $A(i) \in \bigcup_{0 \leq j \leq n/u-1} M_j$; and
- for each $T \in \mathcal{C}$, each module M_i, $0 \leq i \leq n/u - 1$, contains at most one element of $\{A(t) : t \in T\}$.

The purpose of skewing is to prevent contention when the pattern of memory access is known in advance. Skewing schemes can be very difficult to design, limiting their practical value to frequently used, special-purpose applications. One such application is in linear algebra, when various vectors of a matrix must be accessed. Standard skewing schemes [115, 121] provide for the case when A is an $n^{1/2} \times n^{1/2}$ matrix (stored in row-major order) and $\mathcal{C}$ contains all rows and columns of the matrix. In this section, we prove a lower bound for this core skewing problem and give an optimal skewing scheme for $2^{2k} \times 2^{2k}$ matrices where $\mathcal{C}$ contains all rows and columns, and also diagonals and $2^k \times 2^k$ subsquares.

6.3.1 A lower bound on skewing

In this section, we show that skewing is as hard as universal hashing.

Theorem 6.5.

Let f be a skewing scheme for $(\mathcal{C}, n, u)$, where $n = u^2$ and $\{iu + j : 0 \leq j \leq u-1\}, \{ju + i : 0 \leq j \leq u-1\} \in \mathcal{C}$ for all i, $0 \leq i \leq u-1$. (That is, A is a $u \times u$ matrix and C contains all rows and columns of A.) Then performing f conservatively requires time

- $\Omega((n \log \min(n/l, n/p))/(p \log 2(n/lp)))$ for $lp \leq n$; and
- $\Omega(l + l \log \min(l, p)/\log 2(lp/n))$ for $lp > n$.

Proof.

In the nontrivial case, we apply the potential function argument of [5] (Lemma 5.3). We estimate $\Phi(f, m)$, the m-wise potential of f, for $m \in \{l, p, n/l, n/p\}$. Recall that m is the size of a segment in the computation of Φ and u is the size of a module. Without loss of generality, we assume in each case that either $m|u$ or $u|m$; this affects our bounds by at most a constant factor. Segment A_i, $0 \leq i \leq n/m - 1$, is the union of $A[im \ldots (i+1)m - 1]$ with some locations from the modules M_j. If $m \leq u$, we divide each module among u/m segments (it does not matter how as long as each segment gets m elements). If $m > u$, each segment contains m/u consecutive modules. The proof of Lemma 5.3 extends to the segments thus defined.

Case 1: $m \leq u$. For $0 \leq r, s \leq n/m - 1$, we have $x_{r,s}(f, m) \leq 1$ and $\Phi(f, m) = \sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x_{r,s}^{+}(f, m) \log x_{r,s}^{+}(f, m) = 0$.

Case 2: $m > u$. For $0 \leq r, s \leq n/m - 1$, we have $x_{r,s}(f, m) \leq (m/u)^2$ and

$$\begin{aligned}\Phi(f, m) &= \sum_{s=0}^{n/m-1} \sum_{r=0}^{n/m-1} x_{r,s}^{+}(f, m) \log x_{r,s}^{+}(f, m)\\ &\leq (n/m)^2 (m^2/n) \log(m^2/n)\\ &= n \log(m^2/n).\end{aligned}$$

These estimates together give the result. □

6.3.2 An optimal skewing scheme

In this section we show that the skewing scheme of [107] is optimal for the Block PRAM model. This skewing scheme is based on a combinatorial design called a *perfect Latin square*. Latin squares have been constructed for centuries; the standard reference is [58].

Definition.

A *Latin square* of order n is an $n \times n$ square composed with symbols from $N = [0 \ldots n-1]$ such that no symbol appears more than once in any row or column.

Let $n = u^2$ and $U = [0 \ldots u-1]$. Consider the locations in the square to be labeled with coordinates from $N \times N$. A *diagonal* is a set of locations of the form $\{(t, k+t) : t \in N\}$ or $\{(t, k-t) : t \in N\}$ for some fixed $k \in N$ (using modulo n arithmetic). A $u \times u$ *subsquare* is a set of locations of the form $\{(iu + t_1, ju + t_2) : t_1, t_2 \in U\}$ for some fixed $i, j \in U$.

For $n = u^2$, a *perfect Latin square* of order n is an $n \times n$ square composed with symbols from N such that no symbol appears more than once in any row, column, diagonal or $u \times u$ subsquare.

Taking the symbols to represent module indices, a Latin square gives a skewing scheme for (C, n^2, n), where C is the collection consisting of all rows and columns of a matrix. Similarly, a perfect Latin square gives a skewing scheme which allows C also to contain diagonals and subsquares.

Construction. [107]

We describe the construction of a perfect Latin square of order $n = 2^{2k}$. We adopt the following notations to simplify the presentation considerably. Let $N = [0 \ldots 2^{2k} - 1] \cong (01)^{2k}$. For $x, y \in N$, let $s(x,y) \in N$ denote the symbol to be placed in location (x, y). For $1 \le i \le 2k$, let $x_i, y_i, s_i(x,y)$ denote the i-th bit of $x, y, s(x,y)$ respectively, most significant bit first. Let $\oplus$ denote the bitwise exclusive-or operation.

The construction is as follows:

$s_i(x,y) = x_{i-1} \oplus y_{k+i}$ for all even i, $1 \le i \le k$;

$s_i(x,y) = x_{k+i-1} \oplus y_i$ for all even i, $k+1 \le i \le 2k$;

$s_i(x,y) = x_i \oplus x_{i+1} \oplus y_{k+i}$ for all odd i, $1 \le i \le k$; and

$s_i(x,y) = x_{k+i} \oplus x_{k+i+1} \oplus y_i$ for all odd i, $k+1 \le i \le 2k$.

For odd k only, the following boundary conditions are given as exceptions to the formulae above:

$s_{k-2}(x,y) = x_{k-1} \oplus y_{2k-2}$

$s_{2k-2}(x,y) = x_{2k-1} \oplus y_{k-2}$

$s_{k-1}(x,y) = x_k \oplus y_{2k-1} \oplus 1$

$s_{2k-1}(x,y) = x_{2k} \oplus y_{k-1} \oplus 1$

$s_k(x,y) = x_{k-2} \oplus y_{2k-2} \oplus y_{2k}$

$s_{2k}(x,y) = x_{2k-2} \oplus y_{k-2} \oplus y_k.$

Let $A[0\ldots 2^{4k}-1]$ be an array representing a $2^{2k} \times 2^{2k}$ matrix in row-major order. The above construction can be applied to give a skewing scheme for $(\mathcal{C}, 2^{2k}, 2^k)$, where $\mathcal{C}$ represents all rows, columns, diagonals and $2^k \times 2^k$ subsquares of the matrix. Let $N = [0\ldots 2^{2k}-1]$, $J = [0\ldots 2^{4k}-1]$ and define a function $\Pi : J \to J$ bitwise by

$\Pi_i(x \cdot 2^{2k} + y) = s_i(x,y)$ for $1 \le i \le 2k$, $x, y \in N$;

$\Pi_i(x \cdot 2^{2k} + y) = x_{i-2k}$ for $2k+1 \le i \le 4k$, $x, y \in N$.

Given that the above construction correctly gives a Latin square, it follows that Π is a (nonhomogeneous) linear permutation which performs s. The modules are the rows of the permuted $2^{2k} \times 2^{2k}$ matrix. From Corollary 5.10 we have the following result.

Theorem 6.6.

There is a skewing scheme s for $(\mathcal{C}, n, u)$, where $n = 2^{4k}, u = 2^{2k}$, such that:

- when s is applied to a $u \times u$ matrix A, $\mathcal{C}$ contains all rows, columns, diagonals and $2^k \times 2^k$ subarrays of A; and
- s can be performed conservatively in time

$O((n \log \min(n/l, n/p))/(p \log 2(n/lp)))$ for $lp \le n$; and

$O(l + l \log \min(l,p)/\log 2(lp/n))$ for $lp > n$. □

Chapter 7

Exploiting locality

A major open question in the study of realistic PRAM models is whether communication locality can be exploited in general purpose parallel computation [76, 89, 189]. The challenge is to design a real machine which will not only efficiently support general purpose parallel computations, but will allow those algorithms which exploit locality (in particular, fast and efficient Block PRAM algorithms) to gain the enhancements to performance of doing so.

In this chapter, we express our belief that such machines could be built by using the Block PRAM as a bridging model for parallel computation. First, we discuss the extent to which the Block PRAM meets the criteria for a bridging model. Recall from Section 3.1 that a bridging model should (a) be realizable in hardware and (b) allow efficient simulations of PRAM algorithms. These issues are addressed in Sections 7.1 and 7.2, respectively. Second, we argue that the natural locality in Block PRAM algorithms can be preserved even during PRAM simulations which use hashing. The notion of *locality-preserving hashing* is introduced in Section 7.3. Finally, we discuss multitasking and multigauging, two features of real multiprocessing systems that do not represent obstacles to PRAM simulation but will nevertheless affect performance. These issues are discussed in Sections 7.4 and 7.5, respectively.

7.1 Distributed-memory support for the Block PRAM

Recall from Section 3.1 that a distributed-memory machine can simulate shared memory by message passing. In this section, we call attention to recently discovered routing algorithms which provide the block pipelining necessary to support the Block PRAM model on distributed-memory machines.

7.1.1 Routing support for block pipelining

Recent *bit-serial* randomized routing algorithms [9, 126] provide strong theoretical support for block pipelining in message-passing architectures. These algorithms allow any permutation of n message packets of size m to be routed in time $O(m + \log n)$ with high probability on hypercube and butterfly networks. Previous randomized routing algorithms [87, 125, 133, 161, 168, 169, 186, 187, 191] are designed for fixed size packets and run in time $O(m \log n)$ with high probability when implemented on real machines [9]. The following theorems demonstrate that routing algorithms are available to support block pipelining on hypercube and butterfly networks, respectively.

Theorem 7.1. [9]
Any permutation of n packets of size m can be routed in $O(m + \log p)$ time with probability $1 - n^{-\Omega(1)}$ on a hypercube network with $p = n$. □

Theorem 7.2. [126]
Any permutation of n packets of size m can be routed in $O(m + \log p)$ time with probability $1 - 2^{-2^{\Omega(\log^{1/2} n)}}$ on a butterfly network with $p = n \log n$. □

7.1.2 Shared memory vs. message passing

The bit-serial routing algorithms in the previous section support block pipelining in the passing of long messages between processors. This amounts to support for the following variant of the Block PRAM, which was also introduced by Aggarwal, Chandra and Snir.

Definition. [5]

A *message passing Block PRAM* with p processors and latency l is a set of p numbered processors, each having a memory of unbounded size. Each processor is a unit-cost RAM with a restricted arithmetic set and $\Theta(\log p)$ word size. All processors synchronously execute the same program; some processors may wait during any given instruction.

The instruction set includes *message passing.* During a message passing step, each processor may send or receive at most one message packet. Packets cannot be combined; i.e., each processor can communicate with at most one other processor during a message passing step.

A message passing step takes time $l + m$, where m is the number of words in the longest packet sent or received. All other instructions may be performed in unit time.

Since message passing can be simulated by reads and writes to shared memory, the message passing Block PRAM model is weaker than the Block PRAM model. However, as the following results of Aggarwal, Chandra and Snir [5] show, the difference in power is not great. In other particular, we can consider bit-serial routing for the message passing Block PRAM model as also substantially supporting the Block PRAM model.

Definitions.

A *Block PRAM with fixed block size* is a Block PRAM with the following restriction: processors can access the global memory only in blocks of size l.

A (parallel) computation is *oblivious* if, when executed on all inputs of the same size, the operations executed and memory locations accessed on each processor during each time step are the same. (Note that all conservative algorithms are oblivious.)

Theorem 7.3. [5]

• For oblivious computations, the message passing Block PRAM is equivalent to the Block PRAM with fixed block size.

• Oblivious computations taking time T on a Block PRAM can be performed in time $T \log l$ on a Block PRAM with fixed block size.

• Nonoblivious computations taking time T on a Block PRAM can be performed in time $T \log p$ on a Block PRAM with fixed block size. □

7.1.3 Hashing in distributed memory

As noted in Section 3.2.2, computations on distributed-memory machines may cause contention and congestion, and the memory space may need to be hashed. In this section, we show that universal hashing can be performed optimally on the message passing Block PRAM model.

Theorem 7.4.

For $lp \leq n$, any member of H_4 can be performed conservatively on a message passing Block PRAM in time $O((n \log \min(n/l, n/p))/(p \log 2(n/lp)))$. This bound is tight.

Proof.

Conservative algorithms are oblivious, so it is equivalent to prove the result for a Block PRAM with fixed block size. The lower bound of Theorem 6.3 for the Block PRAM holds also for the weaker model.

For $lp \leq n$, the algorithm in Theorem 5.9 accesses the shared memory only in blocks of size l and $n/p \geq l$. Blocks of size n/p can be accessed l locations at a time within time $2n/p + 2l = O(n/p + l)$. □

7.2 PRAM simulation on the Block PRAM

In the paper introducing the Block PRAM model, Aggarwal, Chandra and Snir [5] gave a sketch proof that the EREW PRAM can be simulated efficiently on the Block PRAM. This result suggests that the Block PRAM, a model which allows the exploitation of communication locality in particular algorithms, is also a suitable model for general purpose parallel computation.

Theorem 7.5. [5]

Let ϵ and k be positive constants. Then T steps of a PRAM computation with q processors and q^k memory can be simulated (probabilistically) in $O(Tq/p)$ steps on a Block PRAM with p processors, provided that $lp^{1+\epsilon} \leq q$.

Proof sketch.

The PRAM shared memory is distributed across the local memories of the Block PRAM processors, using a random universal hash function. The Block PRAM shared memory is used to route memory requests and replies among processors. With high probability each processor reads and writes to $O(q/p)$ locations at each simulation step, and the routing can be done in time $O(q/p)$. □

We supply details of the simulation which allow us to bound the probability that the simulation will not be efficient. We will use the following well-known tail estimation.

Lemma 7.6. [166]

If $Y_1, \ldots, Y_N$ are independent Bernouilli trials such that the expected value E for their sum is $E(\sum_i Y_i) = n$, and if $\delta > 0$, then $Prob\{\sum_i Y_i \geq (1+\delta)n\} \leq (e^\delta/(1+\delta)^{1+\delta})^n$. □

Theorem 7.7.

Let ϵ and k be positive constants. Then one step of an EREW PRAM computation with q processors and q^k memory can be simulated with probability $1-\exp(-\Omega(q/p))$ in $O(q/p)$ steps on a Block PRAM with p processors, provided that $lp^{1+\epsilon} \leq q$.

Proof.

Overview. The virtual EREW PRAM shared memory is distributed among the local memories of the Block PRAM processors, using a random hash function $f \in H_5$ (see Theorem 6.2). Without loss of generality, we assume $k > 27/4$ (the Block PRAM memories are unbounded). Also, we assume that l, p and q are powers of two and that the word size of both machines is at least $\log q$ (affecting our bounds by at most a constant factor). We take $\delta = 2/3$; it may be verified that this gives a q-wise independent class of hash functions from $[0 \ldots q^k - 1]$ to $[0 \ldots q^k - 1]$ which can be defined in $o(q^{k-1})$ space.

Assume the Block PRAM processors are numbered $[0 \ldots p-1]$ and the EREW PRAM processors are numbered $[0 \ldots q-1]$. Then virtual address x, $0 \leq x \leq q^k - 1$, is hashed to the local memory of Block PRAM processor $f(x) \bmod p$, and EREW PRAM processor y, $0 \leq y \leq q-1$, is simulated by Block PRAM processor $y \bmod p$.

We will use a segment of shared memory corresponding to an array $A[0 \ldots 4q-1]$ to route messages among the processors. The shared memory segment $A[4iq/p \ldots 4(i+1)q/p - 1]$ is called the *home block* for processor i.

A local computation instruction on the EREW PRAM is simulated by having each Block PRAM processor perform (at most) q/p local computations in time $O(q/p)$.

We describe how each EREW PRAM global read and write instruction is simulated in time $O(q/p)$.

- *Coding:* Each processor encodes its (at most) q/p global read requests or write instructions as messages (some possibly null) in a *string* of $2q/p$ words. A read message is two words giving the requesting processor's ID and the virtual address. A write message is two words giving the virtual address and the data to be written.

- *Message passing:* Each processor copies its message string into its home block. Each message has a virtual address to be accessed, which has been hashed to the local memory of a *destination processor.* To route each message, it suffices to copy it into the home block of its destination processor. With high probability, no processor is the destination processor for more than $2q/p$ messages, and the messages can be routed as a permutation on array A in time $O(q/p)$ by Theorem 5.11.

- *Finishing up:* Each processor copies its home block into shared memory. For write instructions, each processor updates the accessed memory locations. For read instructions, each processor encodes the contents of the accessed memory locations as messages and these replies are sent back to the requesting processors as above.

We will call a processor *overloaded* if it is the destination processor for more than $2q/p$ messages during a simulation step. It remains to show that in each simulation step, the probability that some processor is overloaded is $\exp(-\Omega(q/p))$.

Fix a congruence class $\Pi = a \bmod p$ and a set of memory addresses $S \subseteq [0 \ldots q^k - 1]$, $|S| \le q$. From the q-independence of F we have $Prob\{|f(S) \cap \Pi| > 2q/p\} \le (e/4)^{q/p}$. Thus the probability that some processor receives more than $2q/p$ messages is bounded above by $p \cdot \exp(-\Omega(q/p)) = \exp(-\Omega(q/p))$. □

7.3 Locality-preserving hashing

Thus far in this chapter, we have presented evidence that the Block PRAM can be used as a bridging model of parallel computation. In this section, we present an important advantage to choosing the Block PRAM as a bridging model: namely, the possibility of exploiting locality even during PRAM simulations which use hashing.

Definition.

Let $\mathcal{A}$ be a Block PRAM algorithm with time bound $T(n,l,p)$ which uses at most $S(n,l,p)$ shared memory locations. Let F be a universal family of hash functions from $[1\ldots S(n,l,p)]$ to $[1\ldots S(n,l,p)]$. Then F is *locality-preserving* for $\mathcal{A}$ if, for any $f \in F$, f can be performed and inverted by a Block PRAM on $S(n,l,p)$ consecutive locations in shared memory in time $O(T(n,l,p))$.

Our results in Chapter 6 show that H_4 has optimal locality-preserving properties possible for a universal family of hash functions.

Assuming that a locality-preserving hash function has been used, a Block PRAM can exploit locality in hashed shared memory as follows. First, the memory space is unhashed; next, the Block PRAM algorithm is executed; and finally, the memory space is rehashed. The algorithm uses at most $S(n,l,p)$ shared memory locations; without loss of generality these can be consecutive. The definition of locality preservation implies that the hashing and rehashing do not dominate the complexity of the algorithm.

The Block PRAM algorithms of Chapter 5 are immediate applications of locality-preserving hashing. Assume that the shared memory of a parallel computer based on the Block PRAM model has been hashed in order to provide for the PRAM simulation in Theorem 7.7. The complexity of performing many algorithms on such a system will depend on whether a locality-preserving hash function has been used. Table 7.1 illustrates some improvements attainable for the case $lp \leq n$, $l \leq p$.

We should note that PRAM simulations (including Theorem 7.7) typically use at least $\Omega(\log p)$-wise independent universal hash families. H_4, a 2_μ-wise independent family, is not appropriate for these theoretical results. However, there is no evidence to suggest that H_4 and other 2_μ-wise inde-

Algorithm	Generally	Using H_4
Linear permutations	$O(n/p + ln^\epsilon)$	$O(n/p + l^{1+\epsilon})$
Prefix sums	$O(n/p + ln^\epsilon)$	$O(n/p + l \log n / \log l)$
String matching	$O(n/p + ln^\epsilon)$	$O(n/p + l \log n / \log l)$

Table 7.1: Block PRAM complexity in hashed shared memory

pendent universal hash families fail in practical use. Valiant [189] suggests that H_1 can be used in practice for PRAM simulations, with the advantage that H_1 hash functions can be specified in constant space and evaluated in constant time. Because of the advantages of locality preservation, we suggest that H_4 should also be seriously considered for practical applications in general purpose parallel computation.

7.4 Multitasking: the Hierarchical PRAM

Typically, high-performance computers will be *multitasking,* or performing many different algorithms and procedures at once. Note that our scheme for locality-preserving hashing requires that each algorithm running on a multitasking multiprocessor has its own separately hashed memory space.

The benefits of this memory partitioning have recently been recognized by Heywood and Ranka [89], who introduce the *Hierarchical PRAM.* The Hierarchical PRAM consists of a number of *sub-PRAMs* running asynchronously with respect to one another. Each sub-PRAM is a synchronous shared-memory model (e.g., a Block PRAM). Heywood and Ranka write: "One drawback to current PRAM simulation techniques is that they obliterate (through hashing) any natural communication locality that may be present in an algorithm ... The scope of this problem [can be reduced] via simulation of multiple smaller sub-PRAMs. This constrains the extent of obliteration to be within the sub-network simulating the sub-PRAM."

The discovery of locality-preserving hash functions now confirms the value of memory partitioning for general multitasking. This approach may also be helpful when running a single algorithm. For the purpose of demonstrating the value of partitioned shared memory, it would be helpful to find a natural algorithm which can be partitioned into sub-PRAM procedures, some of which exploit locality for the benefit of the whole algorithm.

7.5 Multigauging and corner turning

Many high-performance multiprocessors have been designed to support *multigauging,* or the capability of changing the word size during a computation. By reducing the word size, multigauging multiprocessors are able to provide a proportionate increase in effective parallelism.

Multigauging is not a new idea. Twenty years ago, the Illiac IV's 64 64-bit processing elements could simulate 128 32-bit elements or 512 8-bit elements [34] (although this capability was never actually utilized). Today, multigauging is now a standard feature of *hardware-reconfigurable architectures* including the Blue CHiP [180], the Connection Machine [146, 182] and the Content Addressable Array Parallel Processor [150, 201].

Given multigauging capability, it becomes advantageous to match the word size of the architecture to the word size required by the algorithm. In our analysis thus far, we have assumed a word size of $\Theta(\log p)$ in order to allow the indirect addressing steps present in many PRAM algorithms. For many of the applications in Chapter 5, however, including permutations, prefix sums computations and string matching, this assumption is not necessary.

The algorithmic advantages of multigauging for these and other applications are explored in [150, 172, 181]. However, all of the studies to date relate to specific network architectures, including the mesh, binary tree, butterfly and similar topologies. This is because much of the interest of this work is in preserving the communication structures of network topologies under multigauging simulations.

As an architecture-independent model of parallel computation which accounts for some aspects of communication locality, the Block PRAM model is a natural setting for studying the implications of multigauging. In this section, we illustrate the power of multigauging for performing permutations on the Block PRAM. Similar adaptations can be made for prefix sums computations and string matching.

7.5.1 Corner turning

Definition.

Consider a processor with word size κ. Let W denote an array of κ consecutive locations in the processor's local memory, and let $w_{ij} \in \{0,1\}$ denote the j-th bit of word $W(i)$ for $1 \leq i,j \leq \kappa$. The *corner turning* operation replaces W with the array V, where $V(i) = w_{1i}w_{2i}\cdots w_{\kappa i}$ for $1 \leq i \leq \kappa$.

The corner turning operation is at the heart of all multigauging architectures known to the author. Under certain simplifying assumptions, corner turning is actually necessary to provide multigauging capability [150]. The value of the operation is in providing parallel access to the individual bits in words, thus enabling *bit parallel data transfer.* Fast corner-turning capability is usually provided by a combination of hardware and software support. For example, the Connection Machine's SPRINT chip can input 32 32-bit words from local memory in 32 cycles, and output them, corner-turned, to the floating-point processor in another 32 cycles [146].

We can describe corner-turning capability in complexity-theoretic terms as follows.

Definition.

A RAM-based complexity model (e.g. RAM, PRAM, Block PRAM) with word size κ has *corner turning capability* if the corner turning operation can be performed by a single processor in time $O(\kappa)$.

We believe that corner-turning capability is a reasonable and realistic extension to a RAM-based complexity model. The corner turning operation meets all of the Boolean circuit complexity-based criteria of Section 4.2.8, because it can be performed trivially by circuits in $O(\kappa^2)$ size and $O(1)$ depth.

7.5.2 Bitwise permutations

We can show that on the Block PRAM model, corner turning allows *bitwise permutations* to be performed as quickly (up to a constant factor) as arbitrary word-level permutations (Theorem 5.11).

Theorem 7.8.

Let $\mathcal{B}$ be a Block PRAM with corner-turning capability and word size κ. Then $\mathcal{B}$ can perform any permutation Π on the κn bits in n consecutive locations in shared memory in time

- $O(\min(nl/p, n\log p/(p\log(2n/pl))))$ if $pl \leq n$;
- $O(\min(nl/p, l\log(n/l)\log\min(p,l)/\log l\log(2pl/n)))$ if $pl > n$.

Proof.

Without loss of generality, we assume l, p, κ, n are powers of 2. The permutation algorithm of Theorem 5.11 consists of basic permutation operations of the following two forms:

- Each processor reads n/pl blocks of size l and writes them into their new locations.
- Each processor reads and permutes a block of size n/p and writes it back.

The first operation amounts to reading n/pl blocks of $l\kappa$ bits each and writing them into their new locations, in time $O(n/p + l)$.

The second operation can be adapted for bitwise permutations as follows.

Case 1: $\kappa \leq n/p$. For a given processor, consider the $n\kappa/p$ bits in the block as a $n/p \times \kappa$ matrix of bits, stored in row-major order. Any permutation on these bits can be performed in three phases by executing permutations on the rows, the columns, and the rows again. Each permutation on the rows is a permutation of n/p words in local memory and can be performed in time $O(n/p)$. To perform the permutation on the columns, we transpose the matrix between phases and perform a permutation on the rows instead. The matrix transposition can be performed in time $O(n/p)$ using corner-turning and permutations of words. The overall time required (including the access to shared memory) is $O(n/p + l)$.

Case 2: $\kappa > n/p$. For a given processor, consider the $n\kappa/p$ bits in the block as a $n/p\kappa \times \kappa^2$ matrix of bits, stored in row-major order. Any permutation on these bits can be performed in three phases by executing permutations on the rows, the columns and the rows again. Each permutation on the rows is a permutation of n/p words in local memory and can be performed in time $O(n/p)$. To perform the permutation on the columns, we repeat Case 1 for $n/p\kappa$ instances of $\kappa \times \kappa$ matrices of bits, each taking time $O(\kappa)$. The overall time required is $O(n/p + l)$.

The Block PRAM algorithm can now be adapted, step by step, to perform arbitrary bitwise permutations in the same time bounds as Theorem 5.11. □

Chapter 8

Bulk synchrony

The analytical results thus far relate to the Block PRAM, a synchronous parallel complexity model. The synchronization issue sets our work apart from most of the recent studies on realistic parallel complexity models, which are more concerned with the issue of asynchrony.

Two of the most comprehensive of these studies relate to the Bulk Synchronous Parallel (BSP) model of Valiant [189] and the Asynchronous PRAM model of Gibbons [76]. The BSP, the Asynchronous PRAM and the Block PRAM share many of the same theoretical parameters, and, not surprisingly, the three models have very similar complexity theories.

8.1 The models

The Bulk Synchronous Parallel model and the Asynchronous PRAM are models which allow for subset synchronization. In this chapter, we focus mostly on special instances of these models—the XPRAM and EREW Phase LPRAM, respectively—which allow only for barrier synchronization. In this section, we define each of these models. We make some remarks about subset synchronization in the Asynchronous PRAM model in Section 8.3.

8.1.1 The Bulk Synchronous Parallel model

Definition. [189]

The Bulk Synchronous Parallel (BSP) model consists of p asynchronous processor-memory elements with message-passing and synchronization capability. There are four parameters: p, the number of processors; s, the communication latency or "startup cost"; g, the pipelining rate or "throughput"; and L, the synchronization rate or "periodicity." (Note that lower values of g and L correspond to *faster* pipelining and synchronization, respectively.) The parameters L and s may be taken to be functions of p. It is assumed throughout [189, 190] and in this section that $2 \leq s \leq L \leq p$ and $g \geq 1$.

Each processor is a RAM, capable of performing each local instruction in unit time (measured by its own clock).[1]

Message-passing is performed by a routing algorithm such that each processor can send and receive up to h messages in time $s + gh$. (Each message can be thought of as the contents of one memory location.) It is frequently assumed in [189, 190] and this chapter that $g = O(1)$. Note that this is equivalent to assuming support for arbitrary pipelining.

Synchronizations may be among any subset S (possibly all) of the processors, and take place periodically, as follows. At regular intervals of L time units, the processors in S are tested to determine whether all of them have reached the synchronization point in their local computations. If not, then the processors in S which have already reached the synchronization point must wait for the others to catch up.

8.1.2 The XPRAM

Definition. [190]

The XPRAM is a BSP with the following restriction: the only synchronization steps allowed are *synchronization barriers* for all p processors. The interval between two successive synchronization barriers is called a *superstep*.

During a superstep, processor i performs a_i local instructions, sends b_i messages to other processors, and receives c_i messages from other processors. Let $t = \max\{a_i + gb_i + gc_i + s : 1 \leq i \leq p\}$. The time required for the superstep is $\lceil t/L \rceil L$. Note that $\max(t, L) \leq \lceil t/L \rceil L \leq t + L$ and therefore $\lceil t/L \rceil L = \Theta(t + L)$.

[1]Although not specified in [189], for consistency we assume a restricted arithmetic instruction set and $O(\log p)$ word size.

8.1.3 The Asynchronous PRAM

Definition. [76]

The Asynchronous PRAM model consists of a set of p sequential processors. Each processor is a RAM with a local memory of unbounded size.[2] There is also a shared memory of unbounded size. Each processor runs its own local program independently of the timing of the other processors. Any desired timing dependencies between processors must be specified in the programs for those processors.

Each processor has a local clock. The time required for a processor to execute an instruction is measured in time units on its local clock. There are four types of instructions.

- *Local instruction:* Perform any basic RAM instruction on elements in local memory.
- *Synchronization step:* Processors in a set S synchronize at a logical point in a computation. Each processor in S waits for all of the processors in S to arrive at this point before continuing its local program.
- *Global write:* Write the contents of a private memory location into a shared memory location. A synchronization step must be placed between writes by any two different processors to the same shared memory location.
- *Global read:* Read the contents of a shared memory location into a private memory location. A synchronization step must be placed between a write by a processor to a shared memory location and a read by a different processor to the same location.

8.1.4 The Phase LPRAM

Definition.

A *Phase LPRAM* is an Asynchronous PRAM with two additional parameters B and l, and the following time costs for instructions[3]:

- A local instruction can be performed in unit time.
- The only synchronization steps allowed are *synchronization barriers* for all p processors. A synchronization barrier can be implemented in time

[2]Although not specified in [76], for consistency we assume a restricted arithmetic instruction set and $O(\log p)$ word size.

[3]Gibbons [76] uses the letter d rather than l to represent the latency. For consistency with the Block PRAM, we use the letter l.

B. The interval between two successive synchronization barriers is called a *phase*.

• A processor can write to any k locations in shared memory in time $l + k$ and can read any k locations in shared memory in time $2l + k$, subject to synchronization requirements. The locations must be addressed directly; i.e. by indices in a global array where the indices are present in the program or the local memory.

The parameters B and l may be taken to be functions of p. It is assumed throughout [76] and in this section that $2 \leq l \leq B \leq p$.

CRCW, CREW and EREW variants of the Phase LPRAM model may be considered, according to whether memory locations can be accessed by different processors during the same phase. All of the Phase LPRAM algorithms for specific problems presented in [76] are written for the EREW variant. The EREW variant is also most closely related to the Block PRAM model, and so we focus on this variant.

8.1.5 The arbitrary-Block PRAM

As we have already discussed, block pipelining is a unique feature of the Block PRAM among shared-memory models of parallel computation in the literature. The BSP model and the Asynchronous PRAM both allow arbitrary pipelining. To separate the issues of block pipelining and bulk synchrony, it is helpful to introduce a variant of the Block PRAM model that allows for arbitrary pipelining.

Definition.

An *arbitrary-Block PRAM* is a Block PRAM in which accesses to arbitrary sequences of memory locations can be pipelined. That is, a processor can write to any k locations in shared memory in time $l + k$ and can read any k locations in shared memory in time $l + k$. Concurrent accesses to intersecting sets of memory elements are serviced in some arbitrary order. The locations must be addressed directly; i.e., by indices in a global array where the indices are present in the program or the local memory.

	Simulating model			
Simulated model	**XPRAM**	**LPRAM**	**BPRAM**	**PRAM**
XPRAM		$1^{(1)}$	$1^{(1)}$	$1^{(1)}$
EREW Phase LPRAM	$\log p^{(2,3)}$		$1^{(1)}$	$1^{(1)}$
arbitrary-Block PRAM	$\log p^{(2,4)}$	B/l		$1^{(1)}$
EREW PRAM	$\log p^{(2,5)}$	B	$l^{(1)}$	

(1) results proved in this thesis
(2) for $g = O(1)$, $s = L = \log p$
(3) for general computations; 1 for oblivious computations
(4) for general computations; L/s (i.e. 1) for oblivious computations
(5) for general computations; L (i.e. $\log p$) for oblivious computations

Table 8.1: Summary of simulation results: parallel slackness used

8.2 Simulation results

In this section we describe the best known simulation results among the four models: XPRAM, EREW Phase LPRAM, arbitrary-Block PRAM and EREW PRAM. In particular, we demonstrate that the order in which we have listed them corresponds to their relative power from weakest to strongest.

Table 8.1 summarizes the simulation results in this section by giving the parallel slackness required for each simulation. For example, the arbitrary-Block PRAM can simulate the EREW Phase LPRAM without parallel slackness, and the EREW Phase LPRAM can simulate the arbitrary-Block PRAM given B/l parallel slackness.

The closeness among the models in this section is important from the standpoint of software portability. Valiant [189] envisions that portable software written for the BSP model will include parameters for problem size and the machine-specific values of p, g and L. Software for the Block PRAM model can be written without regard for the problem of asynchrony, using only two machine-specific parameters, p and l. The close simulations among these models indicate that the costs for this relative simplicity of programming are modest.

8.2.1 Computations on the XPRAM

Theorem 8.1. [190]

Let k be a positive constant. Then T steps of an EREW PRAM computation with q processors and q^k memory can be simulated (probabilistically) in $O(Tq/p)$ steps on an XPRAM with p processors, provided that $g = O(1)$, $s = L = \log p$, and $q \geq p \log p$. □

This simulation result is the best known, even for simulating the weaker Block PRAM and EREW Phase LPRAM models on the XPRAM.

For oblivious computations, however, we can show that the EREW Phase LPRAM and the XPRAM are actually equivalent (for $g = O(1)$). This is an application of a simulation technique of Vishkin and Wigderson [196], by which any oblivious EREW PRAM algorithm for p processors can be reconstructed to run in asymptotically the same time, using only p shared memory locations.

Description of simulation. The virtual shared memory is distributed across the local memories of the processors. Each simulation step consists of three stages. In the first stage, each processor copies (at most) one of its virtual shared memory locations into the shared memory. In the second stage, the time step is simulated, with memory locations readdressed as necessary. In the third stage, each processor copies (at most) one shared memory location into its local memory, where it becomes the new virtual shared memory location.

It is proved in [196] that the virtual shared memory can be kept distributed so that of the (at most) p virtual shared memory locations required for each simulation step, at most one is stored in each local memory. The initial distribution of the virtual shared memory and subsequent redistributions (in the third stage of each simulation step) are precomputed according to an edge coloring of a bipartite multigraph associated with the data dependency graph.

The simulation can be adapted as follows to give a simulation of the EREW Phase LPRAM on the XPRAM.

Theorem 8.2.

An oblivious EREW Phase LPRAM algorithm running in time $t(n, p, l, B)$ can be simulated by an XPRAM in time $O(t(n, p, s, L))$ provided $g = O(1)$.

Proof.

Without loss of generality, consider an EREW Phase LPRAM algorithm such that no phase takes time more than B between synchronization barriers. We show how to simulate a given phase of the algorithm in one XPRAM superstep taking time $O(B)$.

During each phase, there are no conflicts to shared memory. We can therefore apply the simulation of [196] with pB processors to maintain an even distribution of the virtual shared memory among p XPRAM processors. We define the corresponding XPRAM superstep in three stages. In the first stage, each processor sends messages giving the contents of the virtual shared memory locations to the processors that need to access them during the superstep. In the second stage, the phase is simulated, with memory locations readdressed as necessary. In the third stage, each processor sends messages to redistribute the virtual shared memory in preparation for the next superstep.

Using the result of [196], each XPRAM processor sends and receives at most B messages in each of the first and third stages. Each XPRAM processor therefore has at most B local computations, $2B$ sent messages and $2B$ received messages during the superstep. The running time of the superstep on the XPRAM is therefore at most $B + 4gB + s = O(B)$. □

8.2.2 Computations on the EREW Phase LPRAM

Theorem 8.3.

An XPRAM algorithm running in time $t = t(n, p, s, g, L)$ can be simulated by an EREW Phase LPRAM in time $O(t(n, p, l, 1, B))$.

Proof.

Consider a superstep running in time t on an XPRAM with the given parameters. Suppose that processor i performs a_i local instructions, sends b_i messages and receives c_i messages during this superstep. Then $t = \Omega(a_i + b_i + c_i + s + L)$ for all i (since $g \geq 1$). This superstep can be simulated in two phases on an EREW Phase LPRAM with the given parameters by allocating one shared memory location to each message.

• *Phase 1:* All processors perform their local instructions, write their sent messages to the shared memory, and then synchronize. This takes time $\max_{1\leq i\leq p}\{a_i + b_i + l\} + B$.

• *Phase 2:* All processors read their received messages from the shared memory, and then synchronize. This takes time $\max_{1\leq i\leq p}\{c_i + l\} + B$.

The overall time required for simulating the superstep on the EREW Phase LPRAM is $\max\{a_i+b_i\}+\max\{c_i\}+2B \leq 2\max\{a_i+b_i+c_i+l+B\} = O(t)$. □

Theorems 8.2 and 8.3 together yield the following

Corollary 8.4.

For oblivious computations, the EREW Phase LPRAM (p, l, B) and the XPRAM (p, s, g, L) models are equivalent for $s = l$, $L = B$ and $g = O(1)$. □

Theorem 8.5. [76]

An arbitrary-Block PRAM algorithm running in time t using p processors can be simulated by an EREW Phase LPRAM running in time $O(tB/l)$ with pl/B processors. □

For most of the efficient EREW Phase LPRAM algorithms in the literature, the simulation result is very close to optimal. Gibbons's EREW Phase LPRAM algorithms for prefix sums computations, Fast Fourier Transform and bitonic sorting [76] can be matched closely by generically simulating the corresponding Block PRAM algorithms. Of the $\Theta(B/l)$ parallel slackness used in the general simulation, at least a factor of $\Omega(B\log l/l\log B)$ parallel slackness is necessary. It is important to note that the algorithms in Chapter 5 cannot be obtained by generically simulating Phase LPRAM algorithms on the (original) Block PRAM model, because the Phase LPRAM allows arbitrary pipelining. The similarities in performance between EREW Phase LPRAM algorithms and simulated Block PRAM algorithms is indicated in Table 8.2.

Algorithm	**Complexity**	
	EREW Phase LPRAM	**Simulated Block PRAM**
Optimal		
Prefix sums	$O(n/p + B\log n/\log B)$	$O(n/p + B\log n/\log l)$
FFT	$O(n\log n/p + B\log n/\log B)$	$O(n\log n/p + B\log n/\log l)$
Bitonic sort	$O(n\log n/p + B\log n/\log B)$	$O(n\log n/p + B\log n/\log l)$
Not optimal		
List ranking	$O(n/p + B\log n/\log B)$	
Integer sorting	$O(B\log n/\log B)$, $p = n$	$O(B\log n/\log l)$, $p = n^{1+\epsilon}$

Table 8.2: Representative EREW Phase LPRAM algorithms

Proposition 8.6. [76]

An EREW PRAM algorithm running in time t using p processors can be simulated by an EREW Phase LPRAM running in time $O(tB)$ with p/B processors.

Proof.

Each Phase LPRAM processor simulates B PRAM processors, synchronizing after each simulated PRAM step. □

8.2.3 Computations on the arbitrary-Block PRAM

Proposition 8.7.

An EREW Phase LPRAM algorithm running in time $t = t(n, p, B, l)$ can be simulated by an arbitrary-Block PRAM in time $O(t(n, p, l, l))$.

Proof.

Set $B = l$ and execute the resulting algorithm, having processors wait at synchronization barriers as necessary. □

Corollary 8.8.

An XPRAM algorithm running in time $t = t(n, p, s, g, L)$ can be simulated by an arbitrary-Block PRAM in time $O(t(n, p, l, 1, l))$. □

We observe that the additional power of the arbitrary-Block PRAM allows for an efficient deterministic PRAM simulation with only $O(l)$ parallel slackness. The following result may be compared with the randomized simulation of Theorem 7.7, which uses $O(lp^{\epsilon})$ parallel slackness.

Proposition 8.9.

An EREW PRAM algorithm running in time t using p processors can be simulated by an arbitrary-Block PRAM running in time $O(tl)$ with p/l processors.

Proof.

Each Block PRAM processor simulates l PRAM processors, pipelining shared memory accesses during each simulated PRAM step. □

The arbitrary-Block PRAM is asymptotically more powerful than the Block PRAM for finely granular problems such as general permutations and list ranking, as the next two propositions show.

Proposition 8.10.

Any permutation on n elements in shared memory can performed conservatively on an arbitrary-Block PRAM in time $O(n/p + l)$.

Proof.

Each processor reads a set of l distinct elements and writes them into their new locations. □

Proposition 8.11.

The complexity of ranking a compact linked list of size n on an arbitrary-Block PRAM is $\Theta(n/p + l \log n / \log l)$.

Proof.

The upper bound is a consequence of Proposition 8.7 and Gibbons's $O(n/p + B \log n / \log B)$ time EREW Phase LPRAM algorithm for list ranking [76].

The lower bound can be proved by a fan-in argument similar to Theorem 5.14. □

Taken together, Theorem 5.18 and Proposition 8.11 show that the observed difficulty of pointer jumping may be taken as evidence that arbitrary pipelining has not yet been supported in practice.

8.2.4 Computations on the EREW PRAM

Propositions 8.12.

An XPRAM algorithm running in time $t = t(n,p,s,g,L)$ can be simulated by an EREW PRAM in time $O(t(n,p,1,1,1)$.

An EREW Phase LPRAM algorithm running in time $t = t(n,p,l,B)$ can be simulated by an EREW PRAM in time $O(t(n,p,1,1))$.

An arbitrary-Block PRAM algorithm running in time $t = t(n,p,l)$ can be simulated by an EREW PRAM in time $O(t(n,p,1))$.

Proof.

EREW Phase LPRAM and arbitrary-Block PRAM algorithms can be run directly on an EREW PRAM with the parameters l and B set to 1.

XPRAM algorithms can be executed on an EREW PRAM by first going through the EREW Phase LPRAM simulation with the parameters s, g and L set to 1 (Theorem 8.3). □

8.3 Subset synchronization

In this section, we summarize the current state of research relating to the power of subset synchronization in the parallel complexity theory. Our discussion relates to a second instance of the Asynchronous PRAM, the *Subset LPRAM* model.

8.3.1 The Subset LPRAM

Definition.

A *Subset LPRAM* is an Asynchronous PRAM with two additional parameters, the constant l and the function $B(x) : [2 \ldots p] \rightarrow [1 \ldots p]$, and the following time costs for instructions:

- A local instruction can be performed in unit time.
- A synchronization step among any x processors can be implemented in time $B(x)$.
- Reads and writes to shared memory are charged as for the Phase LPRAM.

	Complexity	
	Subset LPRAM	**Simulated BPRAM**
Optimal		
Prefix sums	$O(n/p + l\log n/\log l)$	$O(n/p + l\log^2 n/\log^2 l)$
Not optimal		
FFT	$O(l\log n/\log l)$, $p = n$	$O(n\log n/p + l\log^2 n/\log^2 l)$
Bitonic sort	$O(l\log n/\log l)$, $p = n$	$O(n\log n/p + l\log^2 n/\log^2 l)$

Table 8.3: EREW Subset LPRAM algorithms vs. simulated Block PRAM algorithms

It is assumed throughout [76] and in this section that $2 \leq l \leq B(x) \leq p$, and that $B(l) = \Theta(l)$. The value of $B(l)$ is significant because [76] shows that for many fundamental problems it is desirable to synchronize processors in subsets of size l.

CRCW, CREW and EREW variants of the Subset LPRAM may be considered, according to whether memory locations can be accessed by different processors between synchronization steps. All of the algorithms for specific problems suggested in [76] are for the EREW Subset LPRAM. The EREW variant is also most closely related to the Block PRAM model, and so we focus on this variant.

8.3.2 Complexity results

Proposition 8.13. [76]

A Block PRAM algorithm running in time t using p processors can be simulated by an EREW Subset LPRAM running in time $O(t\log p/\log l)$ with $p\log l/\log p$ processors. □

Gibbons [76] suggests Subset LPRAM complexity bounds for various problems, which can be compared with simulation results for the corresponding Block PRAM algorithms in Table 8.3.

¿From these known problem examples, it remains unclear what benefits are to be gained from subset synchronization, particularly if optimal efficiency is to be achieved. The general simulations of Block PRAM algorithms on the Phase LPRAM and Subset LPRAM models (Propositions 8.5 and 8.13) can also be compared. Note that for $B(p) = O(l \log p / \log l)$, the Phase LPRAM is at least as powerful for these simulations as the Subset LPRAM.

More study is needed to relate the issue of subset synchronization to the other models discussed in this chapter. The Hierarchical PRAM of Heywood and Ranka [89], which introduces asynchrony and subset synchronization in a more tightly structured fashion (see Section 7.4) may be helpful in this study.

Chapter 9

Asynchrony

Our treatment of asynchrony thus far has been confined to demonstrating the close relationships between the Block PRAM model and two bulk-synchronous models of parallel computation. That is, we have related our results for the Block PRAM fairly closely to semi-synchronous algorithms but not to asynchronous algorithms.

We concur with Gibbons [76] and Valiant [188] that synchronous and semi-synchronous algorithms will continue to be designed in preference to asynchronous algorithms because of ease of programming and debugging; see Section 3.2.3. Algorithm designers have found it difficult to "think asynchronously," and accordingly, relatively few fundamental procedures have been developed to provide the building blocks for asynchronous parallel algorithms.

In this chapter, we illustrate techniques for the design and analysis of efficient asynchronous parallel algorithms. We first describe a relatively simple but fundamentally important asynchronous algorithm due to Kung [118], the two-processor zero-finding algorithm. We then show how the specification of this algorithm can be adapted to provide fast Boolean circuits for counting modulo 3, improving a long-standing upper bound [134]. Both of these algorithms are based on a Fibonacci divide-and-conquer scheme and are the result of "asynchronous thinking."

These results cover only a small area in a rapidly growing field of research. For a survey of efficient asynchronous algorithms, see [138, 148].

Throughout this chapter, let $\theta = (\sqrt{5}-1)/2 \doteq 0.618$ be the reciprocal of the "golden ratio." Note that $\theta^2 + \theta = 1$. Algorithms will be defined by their transitions between various states, and their complexity will be defined as the maximum number of transitions required for inputs of a given size.

9.1 Zero-finding

The *zero-finding* problem is described as follows. Given a continuous function f having opposite signs at the endpoints of an interval of length L, locate a zero of f within an interval of length 1. In this section we describe Kung's asynchronous two-processor algorithm for the zero-finding problem [118].

To find the zero, processors determine the sign f at various points in the interval, reducing the interval of uncertainty while maintaining the invariant that f has opposite signs at the endpoints. A transition occurs every time a processor evaluates f at a point. There are two types of states:

- $S_1(l)$: The interval of uncertainty has length l, and a processor is performing an evaluation on the point in the interval at distance $\theta^2 l$ from the left endpoint.
- $S_2(l)$: The interval of uncertainty has length l, and two processors are performing evaluations on the points in the interval at distances $\theta^2 l$ and θl from the left endpoint.

Depending on which processor first completes its evaluation, and the result of that evaluation, the following transitions are possible:

- $S_1(l) \rightarrow S_1(\theta^2 l) \vee S_1(\theta l) \vee S_2(l)$;
- $S_2(l) \rightarrow S_1(\theta^2 l) \vee S_2(\theta l)$.

The algorithm can now be described formally as follows:

```
**Process** $P_i$ $(i = 1, 2)$
**While** the length of the interval of uncertainty $> 1$ **do**
    **Compute** the position of the next evaluation point $\Delta$
    **Evaluate** $f(\Delta)$
    **Update** the endpoints and the current state
```

Theorem 9.1. [118]

The above algorithm solves the zero-finding problem in an interval of length L in at most $\lceil \log_{1/\theta} L \rceil \doteq 1.44 \log L$ transitions. This bound is optimal. $\square$

9.2 Shallow circuits for MOD$_3$

For $x \in \{0,1\}^n$, define the *weight of* x as $w(x) = \sum_{i=1}^{n} x_i$. The Boolean *counting functions* $\mathrm{MOD}_{k,r}^{(n)} : \{0,1\}^n \to \{0,1\}$ are defined by $\mathrm{MOD}_{k,r}^{(n)}(x) = 1$ if $w(x) \equiv r \bmod k$ and 0 otherwise. (Where r is omitted we understand $r = 0$.) The circuit size and depth of the counting functions has been of fundamental interest to the structural complexity theory [67, 178]. In this section, we show how to construct U_2-circuits for $\mathrm{MOD}_{3,r}^{(n)}$ having small depth for each n.

A "naive" upper bound for the U_2-depth complexity $D_{U_2}(\mathrm{MOD}_{k,r}^{(n)})$ of the counting functions is described by the following result.

Proposition 9.2.
$D_{U_2}(\mathrm{MOD}_{k,r}^{(n)}) \leq \lceil 1 + \log k \rceil \cdot \log n.$

Proof.
The circuits can be designed recursively by using the identity

$$\mathrm{MOD}_{k,r}^{(n)}(x) = \bigvee_{i=0}^{k-1} \left(\mathrm{MOD}_{k,r+i}^{(\lceil n/2 \rceil)}(x^L) \wedge \mathrm{MOD}_{k,r-i}^{(\lfloor n/2 \rfloor)}(x^R) \right),$$

where x^L and x^R are the left and right halves of x, respectively. $\square$

Recent work by Paterson and Zwick has produced the following global upper bound.

Theorem 9.3. [156]
$D_{U_2}(\mathrm{MOD}_{k,r}^{(n)}) \leq c \log n$, where $c \leq 4.95 \log n$. $\square$

9.2.1 A circuit design tool

With every Boolean function $f : \{0,1\}^n \to \{0,1\}$, let us associate a *mismatch bit problem* $MB(f)$ involving two players $P1$ and $P0$: $P1$ receives a string $x_1 \in f^{-1}(1)$; $P0$ receives a string $x_2 \in f^{-1}(0)$. Their task is to find a coordinate i such that $x_{1,i} \neq x_{0,i}$. Let $C(MB(f))$ denote the minimum number of bits they have to communicate in order for both to agree on such a coordinate. (Unlike standard problems in communication complexity, the task of the players here is to solve a search problem rather than a decision problem.) Karchmer and Wigderson proved the following result.

Theorem 9.4. [101]
For every function $f : \{0,1\}^n \to \{0,1\}$ we have $D_{U_2}(f) = C(MB(f))$. □

The elegant proof of this result describes very natural constructions, so that explicit algorithms for $MB(f)$ yield U_2-circuit designs for f, and vice versa. From a protocol for $MB(f)$, we may build a circuit upward from the output gate, where each internal gate represents one bit of communication (and each path through the circuit represents a communication sequence). The details may be found in [101].

9.2.2 A communication algorithm for MOD$_3$

The mismatch bit problem may be described in the following terms. Given an interval $[1, n]$ which is known to contain the mismatch, locate one of these mismatches within an interval containing one element (i.e. locate the exact position of one mismatch).

In our algorithm for $MB(\text{MOD}_3)$, players communicate the weights (mod 3) of various substrings of their own strings, reducing the interval of uncertainty while maintaining the invariant that the weights of the associated substrings are different. A transition occurs after each two bits of communication. There are three types of states:

• $S_1(l)$: The interval of uncertainty has length l. (For convenience we will refer to the interval as $[1, l]$.) A player has determined that the weights of the substrings associated with $[\theta l + 1, l]$ are congruent (mod 3) and is communicating this fact, along with the weight (mod 3) of its own substring associated with $[\theta^2 l + 1, \theta l]$.

• $S_2(l)$: The interval of uncertainty has length l. A player has determined that the weights of the substrings associated with $[\theta l + 1, l]$ are incongruent (mod 3) and is communicating this fact.

• $S_3(l)$: The interval of uncertainty has length l. A player has just

Previous state	*Congruence?*	*Weight* (mod 3)	*Message*	*New state*
S_1 or S_3	Yes	0	00	S_1
S_1 or S_3	Yes	1	01	S_1
S_1 or S_3	Yes	2	10	S_1
S_1 or S_3	No		11	S_2
S_2		0	00	S_3
S_2		1	01	S_3
S_2		2	10	S_3

Table 9.1: Encoding scheme for $MB(\mathrm{MOD}_3^{(n)})$

communicated an incongruence and is now communicating the weight (mod 3) of its own substring associated with $[\theta l + 1, l]$.

Depending on the input strings, the following transitions are possible:

- $S_1(l) \rightarrow S_1(\theta l) \vee S_2(\theta l)$;
- $S_2(l) \rightarrow S_3(\theta^2 l)$;
- $S_3(l) \rightarrow S_1(l) \vee S_2(l)$.

The algorithm can now be described formally as follows:

```
Players P1, P0 alternately do
While the length of the interval of uncertainty > 1
    Compute the new interval of uncertainty
    Communicate the required message
    Update the endpoints and the current state
```

We can now analyze the performance of the algorithm. First we bound the communication used for each transition. The following encoding scheme shows that each transition can be effected using only two bits of communication.

Next we bound the number of transitions needed. Consider the changes possible from each state after $k \geq 3$ transitions.

- $S_1(l) \rightarrow S_1(\theta^k l) \vee S_2(\theta^k l) \vee S_3(\theta^{k+1} l)$
- $S_2(l) \rightarrow S_1(\theta^k l) \vee S_2(\theta^k l) \vee S_3(\theta^{k+1} l)$
- $S_3(l) \rightarrow S_1(\theta^{k-1} l) \vee S_2(\theta^{k-1} l) \vee S_3(\theta^k l)$

Since the first state used is either $S_1(n)$ or $S_2(n)$, it is now straightforward to see that after k transitions the interval of uncertainty has length at most $\max(1, \theta^k n)$. Hence at most $\log_{1/\theta} n \doteq 1.44 \log n$ transitions are required. We have the following result.

Theorem 9.5.

The problem $MB(\mathrm{MOD}_3^{(n)}(f))$ can be solved using at most $2\log_{1/\theta} n + O(1) \doteq 2.88 \log n$ bits of communication. □

Corollary 9.6.

$D_{U_2}(\mathrm{MOD}_3^{(n)}) \leq c \log n + O(1)$, where $c = 2/\log(1/\theta) \doteq 2.88.$ □

Similar communication algorithms can be designed which lead to depth $3.47 \log n$ and $4.88 \log n$ U_2-circuits for $\mathrm{MOD}_5^{(n)}$ and $MOD_{11}^{(n)}$, respectively; see [40] for details.

9.2.3 An explicit construction

Recently Zwick [209] has produced the following explicit description of the Boolean circuits for $\mathrm{MOD}_3^{(n)}$ defined by the previous section.

Let F_m denote the m-th Fibonacci number ($F_0 = F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, \ldots$). For $x = (x_1, \ldots, x_{F_m})$ let $x^L = (x_1, \ldots, x_{F_{m-1}})$ and let $x^R = (x_{F_{m-1}} + 1, \ldots, x_n)$. If $w(x) \equiv a \pmod 3$ we abbreviate this as $x \equiv a$; otherwise we write $x \not\equiv a$.

The circuit is defined recursively by the following identities for $a, b \in \{0, 1, 2\}$.

$(x \equiv a) = (x \equiv a)(x^L \equiv 0) \vee (x \equiv a)(x^L \equiv 1) \vee (x \equiv a)(x^L \equiv 2)$; where

$(x \not\equiv a) \vee (x^L \not\equiv b) = ((((x^L \not\equiv b) \wedge (x^{LL} \equiv 0)) \vee ((x^L \not\equiv b) \wedge (x^{LL} \equiv 1))) \vee (((x^L \not\equiv b) \wedge (x^{LL} \equiv 2)) \vee ((((x^R \not\equiv a - b) \wedge (x^{RL} \equiv 0)) \vee ((x^R \not\equiv a - b) \wedge (x^{RL} \equiv 1))) \vee ((X^R \not\equiv a - b) \wedge (x^{RL} \equiv 2)))))$;

$(x \equiv a) \vee (x^L \not\equiv b) = ((((x^L \not\equiv b) \wedge (x^{LL} \equiv 0)) \vee ((x^L \not\equiv b) \wedge (x^{LL} \equiv 1))) \vee (((x^L \not\equiv b) \wedge (x^{LL} \equiv 2)) \vee ((((x^R \equiv a - b) \wedge (x^{RL} \equiv 0)) \vee ((x^R \equiv a - b) \wedge (x^{RL} \equiv 1))) \vee ((X^R \equiv a - b) \wedge (x^{RL} \equiv 2)))))$;

$(x \equiv a) \wedge (x^L \equiv b) = \neg((x \not\equiv a) \vee (x^L \not\equiv b))$; and

$(x \not\equiv a) \wedge (x^L \equiv b) = \neg((x \equiv a) \vee (x^L \not\equiv b))$.

Chapter 10

Robust communication

The aim of this chapter is to show how general purpose parallel computation can continue to proceed correctly even in the presence of randomly occurring communication failures. We do this by introducing a variant of the Block PRAM model which is subject to errors in reading and writing to the shared memory.

10.1 The Faulty Block PRAM

Definition.
A *Faulty Block PRAM* is a Block PRAM with the additional parameter ϕ, such that accesses to locations in shared memory fail independently with probability ϕ. That is, a processor attempting to read a location may instead receive an error symbol; and a processor attempting to write into a location may instead write an error symbol into the location. The parameter ϕ may be taken to be a function of p.

Such errors may result from either interruptions or corruptions in the interprocessor network. In practice, the "error symbol" may be implemented e.g. by using fingerprints (see Section 5.6) or error-checking codes [164].

10.2 Simultaneous bounds

Using the Faulty Block PRAM model, we can prove asymptotically tight simultaneous bounds on complexity and fault tolerance for finely granular problems. These computations can tolerate a certain probability of communication failure while running at optimal speed. In order to make the computations any more robust, however, they must be slowed down. We believe the results in this section to be the first bounds of this kind to have appeared in the literature.

Lemma 10.1.

Assume that a computation requires that $\Omega(n)$ specified locations in shared memory must be read correctly. At time t, the probability that a Faulty Block PRAM has not succeeded in performing the computation is $\phi^{O(tp/n)}$.

Proof.

By the Pigeonhole Principle, at time t, there is some location which has been read at most $O(tp/n)$ times. □

Clearly this bound applies to conservative permutation computations, and to shortcut-based list ranking and tree contraction algorithms.

Lemma 10.2.

An EREW PRAM algorithm taking time $l(n,p)$ can be simulated by a Faulty Block PRAM in time $l \cdot t(n,p)$ with probability of failure $O(t(n,p)\phi^{l})$.

Proof.

For each global variable in the EREW PRAM algorithm we maintain a block of l copies on the Faulty Block PRAM, restoring lost copies during each write step. □

Combining these lemmas with Corollaries 5.20 and 5.24, we have

Theorem 10.3.

Assume conservative and shortcut-based algorithms, and let $\mathcal{B}$ be a Faulty Block PRAM with $n/\log n$ processors, latency $\Theta(\log p)$ and failure parameter $\phi(p) = \Theta(1/\log p)$. Then:

- $\mathcal{B}$ can perform permutation, list ranking or tree contraction in $O(\log^2 n)$ time with probability of failure $\exp(-\Omega(\log n \log\log n))$.
- $\mathcal{B}$ requires $\Omega(\log^2 n)$ time to perform permutation, list ranking or tree contraction with positive probability of correctness.
- Any $O(\log^2 n)$ time algorithm for $\mathcal{B}$ to perform permutation, list ranking or tree contraction must fail with probability $\exp(-O(\log n \log\log n))$.

Proof.

The propositions are applications of Corollaries 5.20 and 5.24. The first proposition uses Lemma 10.2 and the third proposition uses Lemma 10.1.□

10.3 EREW PRAM simulation

In this section, we show that a Faulty Block PRAM with $\phi(p) = 1/\log p$ can be used efficiently for general PRAM computations, provided that the Faulty Block PRAM instruction set is extended to include the *information dispersal operation* [165].

10.3.1 Information dispersal

Let α, β be positive integers with $\alpha > \beta$. The (α, β) *information dispersal problem* is to divide a given data item D into α pieces such that D can be reconstructed from any β of the pieces. We formalize in the following definition.

Definitions.

Let $W = [0 \ldots N-1]$ for some $N \in \mathbb{N}$. Let $< W^\alpha >$ (resp. $< W^\beta >$) represent the set of unordered α-tuples (β-tuples) of elements from W. A (α, γ) *dispersal function* is a mapping $d : W^\gamma \rightarrow < W^\alpha >$ with an associated (β, γ) *reconstruction function* $r :< W^\beta > \rightarrow W^\gamma$ such that for any array $A \in W$ and any $S \subseteq d(A)$ with $|S| = \beta$, $r(S) = A$. (In computing d and r, the sets $d(A)$ and S may be specified as arrays in local memory with their elements given in any order.)

For integers $\alpha > \beta \geq \gamma > 0$, an (α, β, γ) *information dispersal protocol* consists of algorithms for computing a (α, γ) *dispersal function* and an associated (β, γ) *reconstruction function* on words in local memory.

Note that an (α, β, γ) information dispersal protocol can be used on files of any size by segmenting the data into arrays of length γ.

Shamir's "secret sharing" scheme [175], which uses the idea of polynomial interpolation over a finite field, is a $(\alpha, \beta, 1)$ protocol. Recently Rabin [165] reported an improved (α, β, β) protocol based on sets of linearly independent vectors over a finite field.

Theorem 10.4. [165]
For all primes N and all integers $\alpha > \beta > 0$, there is a (α, β, β) information dispersal protocol using words from $[0 \ldots N-1]$. □

When α is not much greater than β, Rabin's protocol is "space-efficient": the dispersed fragments taken together are only α/β times as long as the original data. This observation has led Rabin and Lyuu to design PRAM simulations on the hypercube based on Rabin's protocol [133, 165].

In order to use information dispersal in efficient PRAM simulations, the dispersal and reconstruction functions must also be time-efficient. We can define information dispersal capability in complexity-theoretic terms as follows.

Definition.
A RAM-based complexity model (e.g. RAM, PRAM, Block PRAM) has (α, β, γ) *information dispersal capability* if the dispersal function can be computed in time $O(\alpha + \gamma)$ and the reconstruction function can be computed in time $O(\beta + \gamma)$.

Compared with corner-turning capability (Subsection 7.5.1), information dispersal capability is more difficult to justify in a RAM-based complexity model. In Rabin's (α, β, β) information dispersal protocol, the dispersal algorithm takes time $O(\alpha\beta)$ and the reconstruction algorithm takes time $O(\beta^2)$. This complexity may be reduced somewhat by special-purpose hardware or multigauging techniques, or by future algorithmic improvements. Rabin remarks encouragingly on the feasibility of information dispersal in real architectures in [165].

(It should be noted in this context that PRAM simulations using hashing (e.g. [189, 190]) have proceeded without justifying the assumption that the hash functions used can be evaluated in constant time. Our simulation result in the next section proceeds under an analogous assumption.)

10.3.2 The simulation

In order to fully simulate an PRAM computation while making use of information dispersal in shared memory, some pre- and post-computation is necessary so that the input and output data are specified in undispersed form. Our simulation result refers to *intermediate* steps of PRAM computations.

Theorem 10.5.

Let ϵ and k be positive constants. Then one intermediate step of an EREW PRAM computation with q processors and q^k memory can be simulated in $O(q/p)$ steps with probability $1 - \exp(-\Omega\{\min(q/p, -l\log\phi - \epsilon\log p)\})$ by a Faulty Block PRAM with p processors, latency l, failure parameter ϕ and $(2l, l, l)$ information dispersal capability, provided that $lp^{1+\epsilon} \leq q$.

Proof sketch.

We adapt the simulation of Theorem 7.7 as follows. Without loss of generality, assume that l, p and q are powers of two. The array $A[0\ldots 4q-1]$ used to route messages among the processors is simulated by $A'[0\ldots 8q-1]$. A and A' are subdivided into $4q/l$ *pieces* each of length l and $2l$, respectively. The idea is to let each piece of A' contain the fragments of the corresponding piece of A, using a $(2l, l, l)$ information dispersal protocol. Each fragment is represented by one word.

In the simulation of Theorem 7.7, each access by a processor to the shared memory is to a set of whole pieces:

• Each home block $A[4iq/p\ldots 4(i+1)q/p - 1]$ is a set of $4q/pl$ whole pieces.

• The permutation algorithm of Theorem 5.11 consists of basic permutation operations of the following two forms:

– Each processor reads $4q/pl$ blocks of size l (i.e. whole pieces) and writes them into their new locations.

– Each processor reads and permutes a block of size $4q/p$ (i.e. $4q/pl$ consecutive whole pieces) and writes it back.

We can therefore replace each access to shared memory as follows:

• For each read from a piece of A, instead read from a piece of A' and reconstruct the piece of A.

• For each write into a piece of A, instead disperse the data and write the $2l$ fragments into a piece of A'.

This simulation fails only if the original simulation of Theorem 7.7 fails, or if at some read step, some processor does not have enough (i.e. at least l) fragments to reconstruct a piece of A.

Each fragment is lost with independent probability ϕ, so we can use Lemma 7.6 directly to estimate the probability of failure in reconstructing a given piece. If $Y_i = \{\text{Fragment } i \text{ lost}\}$, this is $Prob\{\sum_i Y_i \geq l\} \leq (e^{1-1/(2\phi)}/(2\phi)^{-1/(2\phi)})^{2l\phi} = \exp(\Omega(l \log \phi))$.

During the simulation, there are at most q/l piece-reconstructions. The overall result follows. □

For illustration, we may consider the parameters for the model $\mathcal{B}$ in Theorem 10.3. The following corollary shows that our simulation (asymptotically) matches the fault-tolerance properties of Rabin's PRAM simulation for the hypercube (see [165] and Section 3.3.5). In particular, the simulation has failure probability $o(1)$ for all polynomial-time computations ($T = q^{O(1)}$).

Corollary 10.6.

A Faulty Block PRAM with p processors, latency $l = \log p$, failure parameter $\phi = 1/\log p$ and $(2\log p, \log p, \log p)$ information dispersal capability can simulate one step of an EREW PRAM with $p^{1+\epsilon}$ processors and p^k memory in time $O(p^\epsilon)$ with probability of failure $\exp(-\Omega(\log p \log\log p))$.□

Many of the Block PRAM algorithms of Chapter 5 have the property that each access by a processor to the shared memory is to a set of whole pieces. We will say that such algorithms are *piecewise coarse.* In these cases information dispersal can be used without going through the simulation of Theorem 10.5, giving the following corollary.

Corollary 10.7.

For $lp \leq n$, $l = \Theta(\log n)$ and $\phi = 1/l^{1+\Omega(1)}$, the following Block PRAM algorithms can run in (asymptotically) the same time on a Faulty Block PRAM (l, p, ϕ) as on a Block PRAM (l, p) with probability $1-\exp(\Omega(l \log \phi))$:

- Fast Fourier Transform (Theorem 5.31);
- Matrix multiplication (Theorem 5.33); and
- Sorting (Theorem 5.35).

Proof sketch.

Each access to the shared memory is supported by $(2l, l, l)$ information dispersal. The initial input and final output steps must be handled specially. Let the input/output size be m. The input is read l times; the probability that any location in the input has not been successfully read is $O(m \cdot \phi^l) = \exp(\Omega(l \log \phi))$. The output is written by alternately writing the (reconstructed) output in blocks of length l and then reading each block to make sure that all locations have been written successfully. The probability that any block fails to be written after l iterations is $O(m/l \cdot (l \cdot \phi)^l) = \exp(\Omega(l \log \phi))$. □

Chapter 11

Conclusions

This thesis has contributed results in two complementary areas of study. First, we have determined the granularity of several specific algorithms and problems, as defined in Section 4.2.1. Second, we have shown that granularity is of fundamental importance among the many practical concerns relating to the state of the art in general purpose parallel computation.

11.1 A taxonomy of granularity

In Section 4.2.1, we introduced a simple and meaningful quantitative definition of granularity based on Block PRAM complexity. Our notion of communication overhead allows the granularity of an algorithm to vary (as it does in practice) with problem size, number of processors and communication latency. This refines previous definitions of granularity [96, 114, 173].

Although there is a three-dimensional space of parameters to explore for each algorithm, we can use the taxonomy given in Section 4.2.1 to determine the granularity of an algorithm from its EREW PRAM and Block PRAM complexities. Using the algorithms and lower bounds in Chapter 5 and 6, we are able to give a detailed classification of several problems and algorithms according to their granularities. The granularity of the problems studied in this thesis depends on the relationship between the latency and the work/processors ratio. Table 11.1 illustrates this dependency for $l = \Theta(\log p)$, $p = O(w)$.

		Work/Processors			
Algorithm[1]	Work	Coarse	A/Coarse	A/Fine	Fine
Matrix transpose	$\Theta(n)$	$l^{1+\Omega(1)}$	$\Omega(l)$	$O(\log l)$	$O(1)$
H_4 hashing	$\Theta(n)$	$l^{1+\Omega(1)}$	$\Omega(l)$	$O(\log l)$	$O(1)$
Skewing	$\Theta(n)$	$l^{1+\Omega(1)}$	$\Omega(l)$	$O(\log l)$	$O(1)$
Prefix sums	$\Theta(n)$	$\Omega(l^2/\log l)$	$\Omega(l^2/\log^2 l)$	$O(l)$	never
String matching[2]	$\Theta(n)$	$\Omega(l^2/\log l)$	$\Omega(l^2/\log^2 l)$	$O(l)$	never
Matrix product[3]	$\Theta(n^3)$	$\Omega(l^2/\log l)$	$\Omega(l^2/\log^2 l)$	$O(l\log l)$	never
Connectivity[4]	$\Theta(m\log n)$	$\Omega(l^2/\log l)$	$\Omega(l^2/\log^2 l)$	$O(l)$	never
FFT	$\Theta(n\log n)$	$\Omega(l^2)$	$\Omega(l^2)$	$o(l^2)$	never
Sorting	$\Theta(n\log n)$	$\Omega(l^2)$	$\Omega(l^2/\log l)$	$O(l\log l)$	$O(l)$
Permuting	$\Theta(n)$	$\exp(\Omega(l))$	$\exp(\Omega(l/\log l))$	$l^{1+O(1)}$	$O(l)$

Table 11.1: Granularity of algorithms for $l = \Theta(\log p)$, $p = O(w)$

Examples. An implementation of the graph connectivity algorithm of Theorem 5.26 will be almost fine for $m \log n/p = O(l)$; i.e., when $p = \Omega(m)$, since the Block PRAM algorithm will take at least $\Omega(l \log n/\log l)/O(l + \log n)) = \Omega(l/\log l)$ times longer than the Shiloach-Vishkin algorithm.

Since the skewing algorithm of Theorem 6.6 is optimal, the parameters in Table 11.1 relate to the granularity of the *problem* of skewing. Skewing can be performed with coarse granularity provided that $n/p = \Omega(l^{1+\Omega(1)})$; i.e., when $p = n/\log^{1+\Omega(1)} n$, and this bound is best possible.

11.2 Granularity in the implementation of algorithms

Years of experience in the mapping of algorithms onto specific parallel architectures have demonstrated the fundamental importance of the granularity issue. In this thesis, we have presented relationships between granularity and major practical issues in general purpose parallel computation. We

[1]Optimal algorithms in italics
[2]Randomized
[3]Using $+, \times$
[4]For $m = n^{1+\Omega(1)}$

have introduced the prospect of locality-preserving simulations, which would allow coarse-grained implementations of Block PRAM algorithms to run efficiently on hashed shared memory. We have also shown how fine and coarse granularity can each be exploited to provide for tolerance of communication failures. We can summarize the recommendations of this thesis for fine- and coarse-grained implementations of algorithms as follows.

Fine-grained implementations of algorithms:

- Should use general simulation techniques
- Can naively achieve optimal fault tolerance
- Suggest the need for hardware support for arbitrary pipelining and/or pointer jumping
- General simulation techniques can achieve sufficient fault tolerance for practical purposes

Coarse-grained implementations of algorithms:

- Should be performed directly or with locality-preserving simulation techniques
- Support the choice of the Block PRAM as a bridging model for general purpose parallel computation
- Should use partitioned shared memory when multitasking
- Piecewise coarse implementations can be modified easily to achieve optimal fault tolerance

11.3 Topics for future research

In this section we suggest a number of topics for further research in areas related to this thesis.

Improving universal hash families: Siegel's universal hash families H_5 [177] can be computed in constant time and sublinear space and are therefore suitable for proving asymptotic simulation results. However, the functions are constructed from expander graphs and large constant factors are involved. Until these constants are improved, it is likely that the families H_1–H_4 will continue to be preferred in practice.

In view of the importance of granularity in parallel computation, we suggest that it is now appropriate to consider locality preservation as a fundamental property of universal hash families. We have seen that H_4, a 2_μ-wise independent hash family, has optimal locality-preserving properties. The precise relationship between the independence and complexity properties of hash families will be a difficult but rewarding study. In particular, the hash function $f(x) = x^2 \bmod n$ seems as hard to perform on the Block PRAM model as any permutation. Because of the growing popularity of the H_3 families, it would be an interesting candidate for improved lower bounds.

Improving information dispersal algorithms: As noted in Section 10.3.1, information dispersal capability is not yet available to support our simulation result for the Faulty Block PRAM (Theorem 10.5). If information dispersal is to be used in practice, faster dispersal and reconstruction algorithms will be of considerable interest.

Assessing the costs and benefits of multigauging architectures: As discussed in Section 7.5, multigauging can be used to increase effective parallelism by allowing the effective word size of a multiprocessor to be matched to the word size required by an algorithm. Some possible applications of multigauging for specific network architectures have been suggested in [150, 172, 181], but a more systematic study is needed in the context of general purpose computation. We have suggested the Block PRAM model as a suitable network-independent model for assessing the costs and benefits of multigauging architectures.

Proving Block PRAM lower bounds: Fine granularity (in the sense of our definition) has important consequences in parallel computation, so that further Block PRAM lower bounds will be of interest. Our lower bound reductions suggest that performing general permutations may become the canonical hard problem for the Block PRAM model of computation.

Designing efficient Block PRAM algorithms: As we have seen in this thesis, efficient Block PRAM algorithms are a suitable foundation for portable and scalable parallel software. Block PRAM algorithms also identify locality-preserving patterns of communication which can be used in the construction of more advanced algorithms. In this thesis, we have demonstrated the usefulness of prefix sums computations and rational permutations as basic operations. It is likely that further efficient Block PRAM algorithms will yield other useful and interesting primitives for general purpose parallel computation. The list ranking bottleneck described in Section 5.3 is one significant challenge to this study.

Designing efficient PRAM algorithms: There is much scope for improvement even in designing algorithms for the ideal PRAM models. The transitive closure bottleneck and the question $P =?NC$, as described in Section 2.3, are two of the many major challenges to theoretical research on parallel complexity.

Designing efficient asynchronous algorithms: Although we have demonstrated close relationships between bulk-synchronous models of computation and the Block PRAM model, little is known about the relationship between general asynchrony and communication locality. The design of asynchronous algorithms seems to require a different style of problem solving from the design of algorithms for the PRAM, Block PRAM and bulk-synchronous models. Chapter 9 illustrates this pattern of "asynchronous thinking" with a fundamental case study.

11.4 Publication history

Unless otherwise indicated, all results in this thesis are my own independent work.

• The results in Chapter 9 are based on research completed in June 1989 and published in [40]. The work improves results presented in my qualifying dissertation [39] and at the Fifth British Colloquium on Theoretical Computer Science (London, April 1989).

• The results in Sections 5.2–5.5, completed in November 1989, were presented at the Sixth British Colloquium on Theoretical Computer Science (Manchester, April 1990) and now appear as a joint paper with William F. McColl in [46].

• The results in Chapter 10, which were completed in November 1990, were presented at the International Conference on Sets, Graphs and Numbers (Budapest, January 1991) and appear as [43].

• The results in Sections 6.1–6.2 and Chapter 7, completed in December 1990, were presented at the Seventh British Colloquium on Theoretical Computer Science (Liverpool, April 1991) and appear as [42].

- The results in Chapter 8 were completed in February 1991 and will appear as [44].
- The results in Sections 5.1 and 6.3 were completed in March 1991, and the results in Section 5.1.1 appear as [41].

These results have also been presented at workshops in Catania and Warwick, and departmental seminars in Oxford, Southampton, Warwick and Austin. A review paper based on the results in this thesis will appear as [45].

Bibliography

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

[2] A. Aggarwal, B. Alpern, A. K. Chandra and M. Snir, "A model of hierarchical memory," *Proc. 19th Annual ACM Symp. on Theory of Computing,* 1987, 305-314.

[3] A. Aggarwal and A. K. Chandra, "Virtual memory algorithms," *Proc. 20th Annual ACM Symp. on Theory of Computing,* 1988, 173–185.

[4] A. Aggarwal, A. K. Chandra and M. Snir, "Hierarchical memory with block transfer," *Proc. 28th Annual IEEE Symp. on Foundations of Computer Science,* 1987, 204–216.

[5] A. Aggarwal, A. K. Chandra and M. Snir, "On communication latency in PRAM computations," *Proc. First Annual ACM Symp. on Parallel Algorithms and Architectures,* 1989, 11–21.

[6] A. Aggarwal, A. K. Chandra and M. Snir, "Communication complexity of PRAMs," *Theor. Comput. Sci.* **71** (1990), 3–28.

[7] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Reading, Mass.: Addison-Wesley Publishing Company, 1974.

[8] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms,* Reading, Mass.: Addison-Wesley Publishing Company, 1983.

[9] W. Aiello, T. Leighton, B. Maggs and M. Newman, "Fast algorithms for bit-serial routing on a hypercube," *Proc. 2nd Annual ACM Symp. on Parallel Algorithms and Architectures,* 1990, 55–64.

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

[11] S. G. Akl, *The Design and Analysis of Parallel Algorithms,* Englewood Cliffs, N.J.: Prentice-Hall International, Inc., 1989.

[12] S. G. Akl and F. Dehne, "The complexity of parallel search on coarse-grained networks," in: *Parallel and Distributed Algorithms* (M. Cosnard *et al.,* ed.), Amsterdam: North-Holland, 1989, 127–136.

[13] F. E. Allen, "Compiling for parallelism," *Proc. IBM Europe Institute Course on Parallel Processing* (G. S. Almasi and G. Paul, ed.), Amsterdam: North-Holland, 1988.

[14] J. Allwright, "The WP6 BSP-Occam library," ESPRIT Project 2701 Working Paper 20, University of Southampton, 1991.

[15] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing,* Redwood City, Calif.: Benjamin/Cummings Publishing Company, Inc., 1989.

[16] H. Alt, "Comparing the combinatorial complexity of arithmetic functions," *J. ACM* **35** (1988), 447–460.

[17] Anderson, Earle, Goldschmidt and Powers, "The IBM System/360 model 91: floating-point execution unit," *IBM J. Res. Dev.* **11** (1967), 34–57.

[18] R. J. Anderson and G. L. Miller, "Deterministic parallel list ranking," *Proc. Aegean Workshop on Computing,* 1988, 81–90.

[19] R. J. Anderson and G. L. Miller, "Optical communication for pointer-based algorithms," Technical Report CRI-81-14, University of Southern California, 1988.

[20] R. J. Anderson and G. L. Miller, "A simple randomized parallel algorithm for list ranking," *Inf. Process. Lett.* **33** (1990), 269–273.

[21] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent processing," *Computing Surveys* **15** (1983), 3–43.

[22] Arvind, D. E. Culler and G. K. Maa, "Assessing the benefits of fine-grained parallelism in dataflow programs," *Int. J. Supercomputer Applications* **2** (1988), 10–36.

[23] C. A. Asmuth and G. R. Blakley, "Pooling, splitting and restituting information to overcome total failure of some channels of communication," *Proc. IEEE Symp. on Security and Privacy,* 1982, 156–169.

[24] W. C. Athas and C. L. Seitz, "Multicomputers: message-passing concurrent computers," *Computer* **21** (1988), 9–25.

[25] J. Backus, "Can programming be liberated from the von Neumann style?" *Comm. ACM* **8** (1978), 613–641.

[26] I. Bar-On and U. Vishkin, "Optimal parallel generation of a computation tree form," *ACM Trans. Program. Lang. Syst.* **7** (1985), 348–357.

[27] K. Batcher, "Sorting networks and their applications," *Proc. AFIPS Spring Joint Computing Conference* **32** (1968), 307–314.

[28] B. Becker and H. U. Simon, "How robust is the n-cube?" *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science,* 1986, 283–291.

[29] S. J. Bellantoni, "Parallel RAMs with bounded memory wordsize," *Proc. First Annual ACM Symp. on Parallel Algorithms and Architectures,* 1989, 83–91.

[30] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik and S. Saxena, "Improved deterministic parallel integer sorting," *Inf. Comput.,* to appear.

[31] G. E. Blelloch, *Vector Models for Data-Parallel Computing,* Cambridge, Mass.: MIT Press, 1990.

[32] N. Blum, "A note on the 'parallel computation thesis,' " *Inf. Process. Lett.* **17** (1983), 203–205.

[33] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications,* Amsterdam: North-Holland, 1976.

[34] W. J. Bouknight *et al.,* "The Illiac IV system," *Proc. IEEE* **60** (1972), 369–379.

[35] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. ACM* **21** (1974), 203–205.

[36] D. A. Calahan, "Performance evaluation of static and dynamic memory systems on the CRAY-2," *Proc. Int. Conf. on Supercomputing,* 1988, 519–524.

[37] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *J. Comput. Syst. Sci.* **18** (1979), 143–154.

[38] D. R. Cheriton, A. Gupta, P. Boyle and H. Goosen, "The VMP multiprocessor: initial experience, refinements and performance evaluation," *Proc. 15th Int. Symp. on Computer Architecture,* June 1988.

[39] A. Chin, "Advances in communication complexity," D. Phil. qualifying thesis, University of Oxford, 1989.

[40] A. Chin, "On the depth complexity of the counting functions," *Inf. Process. Lett.* **35** (1990), 325-328. Extended version available as Technical Report PRG-8-90, University of Oxford.

[41] A. Chin, "Permutations on the Block PRAM," *Inf. Process. Lett.*, submitted. Available as Technical Report PRG-8-91, University of Oxford.

[42] A. Chin, "Locality-preserving hash functions for general purpose parallel computation," *Algorithmica,* submitted. Available as Technical Report PRG-9-91, University of Oxford.

[43] A. Chin, "Latency hiding for fault-tolerant PRAM computations," *Proc. Int. Conf. on Sets, Graphs and Numbers,* 1991, in press. Preliminary version available as Technical Report PRG-21-90, University of Oxford.

[44] A. Chin, "On the equivalence of asynchronous parallel models of computation," in preparation.

[45] A. Chin, "Uniformity of shared memory," *Proc. ALCOM Spring School on Parallel Computation,* (A. M. Gibbons and P. Spirakis, ed.), Cambridge, England: Cambridge University Press, 1991, to appear.

[46] A. Chin and W. F. McColl, "Virtual shared memory: algorithms and complexity," *Inf. Comput.,* to appear. Preliminary version available as Technical Report PRG-9-90, University of Oxford.

[47] F. Y. Chin, J. Lam and I.-N. Chen, "Efficient parallel algorithms for some graph problems," *Comm. ACM* **25** (1982), 659–665.

[48] R. Cole and U. Vishkin, "Deterministic coin tossing with applications to optimal parallel list ranking," *Inf. Control* **70** (1986), 32–53.

[49] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree and graph problems," *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science,* 1986, 478–491.

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

[51] R. Cole and U. Vishkin, "Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time," *SIAM J. Comput.* **17** (1988), 128–142.

[52] R. Cole and O. Zajicek, "The APRAM: incorporating asynchrony into the PRAM model," *Proc. First Annual ACM Symp. on Parallel Algorithms and Architectures,* 1989, 169–178.

[53] R. Cole and O. Zajicek, "The expected advantage of asynchrony," *Proc. 2nd Annual ACM Symp. on Parallel Algorithms and Architectures,* 1990, 85–94.

[54] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation* **19** (1965), 297–301.

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

[56] M. Cosnard, J. M. Muller, Y. Robert and D. Trystram, "Communication costs versus computation costs in parallel Gaussian elimination," in: *Parallel Algorithms and Architectures* (M. Cosnard *et al.*, ed.), Amsterdam: North-Holland, 1986.

[57] W. J. Dally *et al.*, "The J-machine: a fine-grain concurrent computer," *Information Processing 89* (G. X. Ritter, ed.), 1147–1153.

[58] J. Denes and A. D. Keedwell, *Latin Squares and Their Applications,* New York: Academic Press, 1974.

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

[60] P. van Emde Boas, "Machine models and simulations," in: *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Amsterdam: North-Holland, 1990, 1–66.

[61] S. Even, *Graph Algorithms,* Rockville, Md.: Computer Science Press, 1979.

[62] A. Fiat and A. Shamir, "Polymorphic arrays: a novel VLSI layout for systolic computers," *Proc. 25th Annual IEEE Symp. on Foundations of Computer Science,* 1984, 37–45.

[63] F. E. Fich, "New bounds for parallel prefix circuits," *Proc. 15th Annual ACM Symp. on Theory of Computing,* 1983, 100–109.

[64] R. A. Finkel, "Large-scale parallelism: three case studies," in: *The Characteristics of Parallel Algorithms* (D. Gannon, L. H. Jamieson and R. Douglas, ed.), Cambridge, Mass.: MIT Press, 1987.

[65] M. J. Flynn, "Very high speed computing systems," *Proc. IEEE* **54** (1966), 1901–1909.

[66] G. E. Forsythe and C. B. Moler, *Computer Solution of Linear Algebraic Systems,* Englewood Cliffs, N. J.: Prentice-Hall International, Inc., 1967.

[67] M. Furst, J. B. Saxe and M. Sipser, "Parity, circuits and the polynomial-time hierarchy," *Proc. 22nd Annual IEEE Symp. on Foundations of Computer Science,* 1981, 260–270.

[68] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "Cedar: a large scale multiprocessor," *Proc. Int. Conf. on Parallel Processing,* 1983, 524–529.

[69] Z. Galil and J. Seiferas, "Time-space optimal string matching," *Proc. 13th Annual ACM Symp. on Theory of Computing,* 1981, 106–113.

[70] K. Gallivan, W. Jalby, U. Meier and A. H. Sameh, "Impact of hierarchical memory systems on linear algebra algorithm design," *Int. J. Supercomputer Applications* **2** (1988), 12–48.

[71] H. Gazit, G. L. Miller and S. H. Teng, "Optimal tree contraction in the EREW model," *Proc. Princeton Workshop on Algorithm, Architecture and Technology Issues for Models of Concurrent Computation,* New York: Plenum Press, 1987, 139–156.

[72] N. Gehani, *Ada Concurrent Programming,* Englewood Cliffs, N. J.: Prentice-Hall International, Inc., 1984.

[73] A. M. Gibbons and M. S. Paterson, "Embedding balanced binary trees in the mesh," University of Warwick, 1991, to appear.

[74] A. M. Gibbons and W. Rytter, "An optimal parallel algorithm for dynamic expression evaluation and its applications," *Proc. Symp. on Foundations of Software Technology and Theoretical Computer Science,* 1986, 453–469.

[75] A. M. Gibbons and W. Rytter, *Efficient Parallel Algorithms,* Cambridge, England: University of Cambridge Press, 1988.

[76] P. B. Gibbons, "The Asynchronous PRAM: a semi-synchronous model for shared memory MIMD machines," Ph. D. thesis, University of California at Berkeley, 1989.

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

[78] A. Gottlieb, B. D. Lubachevsky and L. Rudolph, "Coordinating large numbers of processors," *Proc. Int. Conf. on Parallel Processing,* 1981.

[79] R. Greenlaw, H. J. Hoover and W. L. Ruzzo, "A compendium of problems complete for $\mathcal{P}$, part I: Theory and open problems," Technical Report, University of Washington, 1990.

[80] R. Greenlaw, H. J. Hoover and W. L. Ruzzo, "A compendium of problems complete for $\mathcal{P}$, part II: $\mathcal{P}$-complete problems," Technical Report, University of Washington, 1990.

[81] A. K. Gupta and S. E. Hambrusch, "Optimal three-dimensional layouts of complete binary trees, *Inf. Process. Lett.* **26** (1987), 99–104.

[82] T. Hagerup, "Constant-time parallel integer sorting," *Proc. 23rd Annual ACM Symp. on Theory of Computing,* 1991.

[83] G. Hains, personal communication, 1991.

[84] Y. Han, "Matching partition a linked list," *Proc. First Annual ACM Symp. on Parallel Algorithms and Architectures,* 1989, 246–253.

[85] F. Harary, *Graph Theory,* Reading, Mass.: Addison-Wesley, 1969.

[86] R. Hassin and D. B. Johnson, "An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks," *SIAM J. Comput.* **14** (1985), 612–624.

[87] J. Håstad, T. Leighton and M. Newman, "Fast computation using faulty hypercubes," *Proc. 21st Annual ACM Symp. on Theory of Computing,* 1989, 251–263.

[88] M. C. Herbordt, C. C. Weems and J. C. Corbett, "Message-passing algorithms for a SIMD torus with coteries," *Proc. 2nd Annual ACM Symp. on Parallel Algorithms and Architectures,* 1990, 11–20.

[89] T. Heywood and S. Ranka, "A practical hierarchical model of parallel computation: the model," Technical Report SU-CIS-91-06, Syracuse University, 1991.

[90] T. Heywood and S. Ranka, "A practical hierarchical model of parallel computation: binary tree and FFT graph algorithms," Technical Report SU-CIS-91-07, Syracuse University, 1991.

[91] W. D. Hillis, *The Connection Machine,* Cambridge, Mass.: MIT Press, 1985.

[92] D. S. Hirschberg, "Parallel algorithms for the transitive closure and the connected components problems," *Proc. 8th Annual ACM Symp. on Theory of Computing,* 1976, 55–57.

[93] D. S. Hirschberg, "Fast parallel sorting algorithms," *Comm. ACM* **21** (1978), 657–661.

[94] C. A. R. Hoare, *Communicating Sequential Processes,* Englewood Cliffs, N. J.: Prentice-Hall International, Inc., 1985.

[95] C. S. Iliopolous, "Call for participation: Combinatorial Pattern Matching School," 1991.

[96] L. H. Jamieson, "Characterizing parallel algorithms," in: *The Characteristics of Parallel Algorithms* (D. Gannon, L. H. Jamieson and R. Douglas, ed.), Cambridge, Mass.: MIT Press, 1987.

[97] D. S. Johnson, "A catalog of complexity classes," in: *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Amsterdam: North-Holland, 1990, 67–161.

[98] H. Jung, personal communication, 1990.

[99] P. Kanellakis and A. Shvartsman, "Efficient parallel algorithms can be made robust," *Proc. 8th Annual ACM Symp. on Principles of Distributed Computing,* 1989, 211–222.

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

[101] M. Karchmer and A. Wigderson, "Monotone circuits for connectivity require super-logarithmic depth," *Proc. 20th Annual ACM Symp. on Theory of Computing,* 1988, 539–550.

[102] R. M. Karp and R. E. Miller, "Parallel program schemata," *J. Comput. Syst. Sci.* **3** (1969), 147–195.

[103] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.* **31** (1987), 249–260.

[104] R. M. Karp and V. Ramachandran, "Parallel algorithms for shared-memory machines," in: *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Amsterdam: North-Holland, 1990, 869–942.

[105] Z. M. Kedem, K. V. Palem, A. Raghunathan and P. G. Spirakis, "Combining tentative and definite executions for very fast dependable parallel computing," *Proc. 23rd Annual ACM Symp. on Theory of Computing,* 1991.

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

[107] K. Kim and V. K. Prasanna Kumar, "Perfect Latin squares and parallel array access," *Proc. 16th Annual IEEE-ACM Int. Symp. on Computer Architecture,* 1989, 372–379.

[108] D. E. Knuth, J. H. Morris and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.* **6** (1977), 323–350.

[109] S. R. Kosaraju and M. J. Atallah, "Optimal simulations between mesh-connected arrays of processors," *Proc. 18th Annual ACM Symp. on Theory of Computing,* 1986, 264–272.

[110] S. R. Kosaraju and A. L. Delcher, "Optimal parallel evaluation of tree-structured computations by raking," *Proc. 3rd Aegean Workshop on Computing,* 1988, 101–110.

[111] V. Koubek and J. Kršňáková, "Parallel algorithms for connected components in a graph," *Proc. 5th Int. Conf. on Fundamentals of Computation Theory,* Springer-Verlag Lecture Notes in Computer Science vol. 199, 208–217.

[112] C. P. Kruskal, L. Rudolph and M. Snir, "The power of parallel prefix," *IEEE Trans. Comput.* **34** (1985), 965–968.

[113] C. P. Kruskal, L. Rudolph and M. Snir, "A complexity theory of efficient parallel algorithms," *Theor. Comput. Sci.* **71** (1990), 95–132.

[114] C. P. Kruskal and C. H. Smith, "Definitions of granularity," in: *Proc. Int. Symp. on High Performance Computer Systems* (E. Gelenbe, ed.), Amsterdam: North-Holland, 1988, 257–268.

[115] D. J. Kuck, "Illiac IV software and applications programming," *IEEE Trans. Comput.* **17** (1968), 758–770.

[116] D. J. Kuck, "A survey of parallel machine organization and programming," *Computing Surveys* **9** (1977), 29–59.

[117] D. J. Kuck, R. H. Kuhn, B. Leasure and M. Wolfe, "The structure of an advanced retargetable vectorizer," in: *Tutorial on Supercomputers* (K. Hwang, ed.), IEEE Press, 1984.

[118] H. T. Kung, "Synchronized and asynchronous parallel algorithms for multiprocessors," in: *Algorithms and Complexity: New Directions and Recent Results* (J. F. Traub, ed.), New York: Academic Press, 1976, 153–200.

[119] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM* **27** (1980), 831–838.

[120] T. W. Lam, "Some topics in communication complexity theory," Ph. D. thesis, University of Washington, 1988.

[121] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.* **24** (1975), 1145–1155.

[122] D.-L. Lee and Y. H. Wang, "Conflict-free access of arrays in a parallel processor," *Proc. First Annual ACM Symp. on Parallel Algorithms and Architectures,* 1989, 313–317.

[123] F. T. Leighton, *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graphs and Other Networks,* Cambridge, Mass.: MIT Press, 1983.

[124] F. T. Leighton, C. E. Leiserson and M. Klugerman, "Theory of parallel and VLSI computation," Lecture Notes 18.435/6.848, MIT Laboratory for Computer Science, 1991.

[125] F. T. Leighton, B. Maggs and S. Rao, "Universal packet routing algorithms," manuscript, 1987.

[126] F. T. Leighton and C. G. Plaxton, "A (fairly) simple circuit that (usually) sorts," *Proc. 31st Annual IEEE Symp. on Foundations of Computer Science,* 1990.

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

[128] J. Lenfant, "Parallel permutations of data: a Benes network control algorithm for frequently used permutations," *IEEE Trans. Comput.* **27** (1978), 637–647.

[129] M. Li and Y. Yesha, "New lower bounds for parallel computation," *Proc. 18th Annual ACM Symp. on Theory of Computing,* 1986, 177–187.

[130] C. Liddiard, personal communication.

[131] B. Lint and T. Agerwala, "Communication issues in the design and analysis of parallel algorithms," *IEEE Trans. Softw. Eng.* **7** (1981), 174–188.

[132] L. Lovász, "Communication complexity: a survey," Technical Report CS-204-89, Princeton University, 1989.

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

[134] W. F. McColl, "Some results on circuit depth," Ph.D. thesis, University of Warwick.

[135] W. F. McColl, "Parallel algorithms and architectures," *Proc. Shell Conf. on Parallel Computing,* New York: Springer-Verlag, 1988, 1–22.

[136] W. F. McColl, "General purpose parallel computing," *Proc. ALCOM Spring School on Parallel Computation* (A. M. Gibbons and P. Spirakis, ed.), Cambridge, England: Cambridge University Press, 1991, to appear.

[137] E. S. Maniloff, K. M. Johnson and J. Reif, "Holographic routing network for shared-memory parallel computers," Technical Report CSE-89-8, University of California at Davis, 1989.

[138] C. Martel, A. Park and R. Subramonian, "Optimal asynchronous algorithms for shared memory parallel computers," Technical Report CSE-89-8, University of California at Davis, 1989.

[139] Y. Matias and U. Vishkin, "On parallel hashing and integer sorting," *Proc. 17th Int. Conf. on Automata, Languages and Programming,* New York: Springer-Verlag, 1990, 729–743.

[140] K. Mehlhorn and U. Vishkin, "Granularity of parallel memories," Technical Report 89 (Ultracomputer Note 59), New York University, 1983.

[141] K. Mehlhorn and U. Vishkin, "Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories," *Acta Inf.* **21** (1984), 339–374.

[142] G. L. Miller, "Are pointer-based parallel algorithms realistic?" in: *Opportunities and Constraints of Parallel Computing* (J. L. C. Sanz, ed.), New York: Springer-Verlag, 1989, 85.

[143] "MSI researcher wins Cray Gigaflop Performance Award," *Minnesota Supercomputer Institute Research Bulletin* **7** (1991), 7.

[144] S. Miyano, S. Shiraishi and T. Shoudai, "A list of $\mathcal{P}$-complete problems," Technical Report RIFIS-CS-17, Kyushu University 33, 1989.

[145] J. K. Mullin, "A caution on universal classes of hash functions," *Inf. Process. Lett.* **37** (1991), 247–256.

[146] P. J. Narayanan, personal communication.

[147] D. Nassimi and S. Sahni, "An optimal routing algorithm for mesh-connected parallel computers," *J. ACM* **27** (1980), 6–29.

[148] N. Nishimura, "Asynchronous shared memory parallel computation," *Proc. Second Annual ACM Symp. on Parallel Algorithms and Architectures,* 1990, 76–84.

[149] D. Nussbaum and A. Agarwal, "Scalability of parallel machines," *Comm. ACM* **34** (1991), 57–61.

[150] B. Obrenic *et al.,* "Using emulations to construct high-performance virtual parallel architectures," *Proc. 3rd Annual IEEE Symp. on Frontiers of Massively Parallel Computation,* 1990.

[151] C. H. Papadimitriou and J. D. Ullman, "A communication-time trade-off," *SIAM J. Comput.* **16** (1987), 639–647.

[152] C. H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *Proc. 20th Annual ACM Symp. on Theory of Computing,* 1988, 510–513.

[153] I. Parberry, "Parallel speedup of sequential machines: a defense of the parallel computation thesis," *SIGACT News* **18** (1986), 54–67.

[154] I. Parberry, *Parallel Complexity Theory,* London: Pitman Publishing, 1987.

[155] M. S. Paterson, "Improved sorting algorithms with $O(\log n)$ depth, *Algorithmica* **5** (1990), 75–92.

[156] M. S. Paterson and U. Zwick, "Shallow multiplication circuits," Proc. 10th Annual IEEE Symp. on Computer Arithmetic, 1991, 28–34.

[157] M. C. Pease III, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.* **26** (1977), 458–473.

[158] G. F. Pfister *et al.,* "The IBM Research Parallel Prototype (RP3): introduction and architecture," *Proc. Int. Conf. on Parallel Processing,* 1985, 764–771.

[159] G. F. Pfister and V. A. Norton, " 'Hot spot' contention and combining in multistage interconnection networks," *Proc. Int. Conf. on Parallel Processing,* 1985, 790–797.

[160] N. J. Pippenger, "On simultaneous resource bounds," *Proc. 20th Annual IEEE Symp. on Foundations of Computer Science,* 1979, 307–311.

[161] N. J. Pippenger, "Parallel communication with limited buffers," *Proc. 25th Annual IEEE Symp. on Foundations of Computer Science,* 1984, 127–136.

[162] C. G. Plaxton, personal communication, 1991.

[163] F. P. Preparata, "Optimal three-dimensional VLSI layouts," *Math. Syst. Theory* **16** (1983), 1–8.

[164] M. O. Rabin, "Fingerprinting by random polynomials," Technical Report TR-15-81, Harvard University Center for Research in Computing Technology, 1981.

[165] M. O. Rabin, "Efficient dispersal of information for security, load balancing and fault tolerance," *J. ACM* **36** (1989), 335–348.

[166] P. Raghavan, "Probabilistic construction of deterministic algorithms: approximating packing integer programs," *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science,* 1986, 10–18.

[167] C. S. Raghavendra and V. K. Prasanna Kumar, "Permutations on Illiac IV-type networks," *IEEE Trans. Comput.* **35** (1986), 662–669.

[168] A. G. Ranade, "How to emulate shared memory," *Proc. 28th Annual IEEE Symp. on Foundations of Computer Science,* 1987, 185–194.

[169] A. G. Ranade, "Fluent parallel computation," Ph. D. thesis, Yale University, 1989.

[170] J. H. Reif, "An optimal parallel algorithm for integer sorting," *Proc. 26th Annual IEEE Symp. on Foundations of Computer Science,* 1985, 496–504.

[171] R. Rettberg and R. Thomas, "Contention is no obstacle to shared-memory multiprocessing," *Comm. ACM* **29** (1986), 1202–1212.

[172] T. D. de Rose, L. Snyder and C. Yang, "Near-optimal speedup of graphics algorithms using multigauge parallel computers," *Proc. Int. Conf. on Parallel Processing,* 1987, 289–294.

[173] V. Sarkar and J. Hennessy, "Partitioning parallel programs for macro-dataflow," *Proc. ACM Conf. on Lisp and Functional Programming,* 1986, 202–211.

[174] A. Schönhage and V. Strassen, "Schnelle Multiplikation grosser Zahlen," *Computing* **7** (1971), 281–292.

[175] A. Shamir, "How to share a secret," *Comm. ACM* **22** (1979), 612–613.

[176] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *J. Algorithms* **3** (1982), 57–67.

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

[178] R. Smolensky, "Algebraic methods in the theory of lower bounds for Boolean function complexity," *Proc. 19th Annual ACM Symp. on Theory of Computing,* 1987, 77–82.

[179] M. Snir, "On parallel searching," *SIAM J. Comput.* **14** (1985), 688–708.

[180] L. Snyder, "Introduction to the Configurable Highly Parallel computer," *Computer* **15** (1982), 47–56.

[181] L. Snyder, "An inquiry into the benefits of multigauge parallel computation," *Proc. Int. Conf. on Parallel Processing,* 1985, 488–492.

[182] C. Stanfill, "Communications architecture in the Connection Machine system," Technical Report HA-87-3, Thinking Machines Corp., Cambridge, Mass., 1987.

[183] L. Stockmeyer and U. Vishkin, "Simulations of parallel random access machines by circuits," *SIAM J. Comput.* **13** (1984), 409–422.

[184] H. S. Stone, "Parallel processing with perfect shuffle," *IEEE Trans. Comput.* **20** (1971), 153–161.

[185] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Comm. ACM* **20** (1977), 263–271.

[186] E. Upfal and A. Wigderson, "How to share memory in a distributed system," *Proc. 25th Annual IEEE Symp. on Foundations of Computer Science,* 1984, 171–180.

[187] L. G. Valiant, "A scheme for fast parallel communication," *SIAM J. Comput.* **11** (1982), 350–361.

[188] L. G. Valiant, "Bulk-synchronous parallel computers," Technical Report 08-89, Harvard University, Cambridge, Mass., 1989.

[189] L. G. Valiant, "A bridging model for parallel computation," *Comm. ACM* **33** (1990), 103–110.

[190] L. G. Valiant, "General purpose parallel architectures," in: *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Amsterdam: North-Holland, 1990, 943–971.

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

[192] J. H. van Lint, *Introduction to Coding Theory,* New York: Springer-Verlag, 1982.

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

[194] U. Vishkin, "An optimal parallel connectivity algorithm," *Discrete Appl. Math.* **9** (1984), 197–207.

[195] U. Vishkin, "Optimal parallel pattern matching in strings," *Inf. Control* **67** (1985), 91–113.

[196] U. Vishkin and A. Wigderson, "Dynamic parallel memories," *Inf. Control* **56** (1983), 174–182.

[197] J. S. Vitter and E. A. M. Schriver, "Optimal disk I/O with parallel block transfer," *Proc. 2nd Annual ACM Symp. on Parallel Algorithms and Architectures,* 1990, 159–169.

[198] J. S. Vitter and E. A. M. Schriver, "Algorithms for parallel memory I: Two-level memories," Technical Report CS-90-21, Brown University, Providence, R. I., 1990.

[199] J. S. Vitter and E. A. M. Schriver, "Algorithms for parallel memory II: Hierarchical multilevel memories," Technical Report CS-90-22, Brown University, Providence, R. I., 1990.

[200] J. von Neumann, *The Computer and the Brain,* New Haven, Conn.: Yale University Press, 1958.

[201] C. C. Weems *et al.,* "The Image Understand Architecture," *Int. J. Computer Vision* **2** (1987), 251–282.

[202] I. Wegener, *The Complexity of Boolean Functions,* Chichester, England: John Wiley and Sons, 1987.

[203] D. J. A. Welsh, *Codes and Cryptography,* Oxford, England: Clarendon Press, 1988.

[204] H. A. G. Wijshoff, "A phenomenology of current parallel computers," in: *Proc. ALCOM Spring School on Parallel Computation* (A. M. Gibbons and P. Spirakis, ed.), Cambridge, England: Cambridge University Press, 1991, to appear.

[205] H. A. G. Wijshoff and J. van Leeuwen, "The structure of periodic storage schemes for parallel memories," *IEEE Trans. Comput.* **34** (1985), 501–505.

[206] J. C. Wyllie, "The complexity of parallel computation," Ph. D. thesis, Cornell University, 1979.

[207] A. C.-C. Yao, "Some complexity questions related to distributed computing," *Proc. 11th Annual ACM Symp. on Theory of Computing,* 1979, 209–213.

[208] O. Zajicek, "The APRAM: a model for asynchronous parallel computation," Ph. D. thesis, New York University, 1990.

[209] Uri Zwick, personal communication, 1990.