The Asynchronous PRAM: A Semi-Synchronous Model for Shared Memory MIMD Machines

Phillip Baldwin Gibbons[1]

TR-89-062

December 11, 1989

Abstract

This thesis introduces the *Asynchronous PRAM* model of computation, for the design and analysis of algorithms that are suitable for large parallel machines in which processors communicate via a distributed, shared memory. The Asynchronous PRAM is a variant of the well-studied PRAM model which differs from the PRAM in two important respects: (i) the processors run asynchronously and there is an explicit charge for synchronization, and (ii) there is a non-unit time cost to access the shared memory.

Many new algorithms are presented for the Asynchronous PRAM model. We modify a number of PRAM algorithms for improved asymptotic time and processor complexity in the Asynchronous PRAM. We show general classes of problems for which the time complexity can be improved by restructuring the computation. We prove lower bounds that reflect limitations on information flow and load balancing in this model. Simulation results between the Asynchronous PRAM and various known synchronous models are presented as well.

We introduce a *post office gossip game* for studying the inherent synchronization complexity of coordinating processors using pairwise synchronization primitives. Results are presented that compare the relative power of various such primitives. These results and techniques are used to reduce the amount of synchronization in Asynchronous PRAM algorithms.

Furthermore, we discuss a programming model based on the Asynchronous PRAM. We introduce the notion of a *semi-synchronous* programming model, a model for repeatable asynchronous programs. *Repeatable* programs, in which the output and all intermediate results are the same every time the program is run on a particular input, greatly simplify the tasks of writing, debugging, analyzing, and testing programs.

Finally, we discuss hardware support for the Asynchronous PRAM model. In particular, we present a cache protocol suitable for the Asynchronous PRAM and a new technique for barrier synchronization.

---

1. International Computer Science Institute, Berkeley, CA.

The Asynchronous PRAM: A Semi-Synchronous Model for Shared Memory MIMD Machines

Phillip Baldwin Gibbons

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate Division of the University of California at Berkeley.

Acknowledgments

First of all, I would like to thank Dick Karp, my research advisor, for his encouragement and direction in my research. I am grateful for the many hours Dick spent listening to my work in progress and reading drafts of my papers. His quick mind and knowledge of the field was invaluable in helping to clarify my thinking, simplify my proofs, and extend my results. Dick has encouraged me throughout and, despite the many demands on his time, has always been available to discuss my work.

I thank the other two members of my thesis committee, Umesh Vazirani and Bob Solovay, for their interest in my work, their helpful comments, and their willingness to put the time and energy into serving on my committee.

I thank Jorge Sanz for his enthusiasm for my work, his interest in my professional future, and our many enjoyable technical discussions. Jorge encouraged me to become a student of all aspects of parallel computation and to address problems in software and hardware, in addition to theory. It has been a pleasure to work with Jorge.

I had the pleasure of working with Danny Soroker for my entire graduate career. Danny shares my interest in combining the theoretical with the practical in parallel computation. I have benefited greatly from his technical comments, his advice and his friendship. Much of chapters one, five, and six of this thesis resulted from discussions with Jorge and Danny.

David Anderson has been a constant source of encouragement. I thank David for many technical and non-technical discussions, his support for my work, and his help in revising papers.

This work has also benefited from interesting discussions with Bob Cypher, Tsahi Birk, Edith Cohen, Melanie Lewis, Vijaya Ramachandran, and Gary Miller.

I am grateful to John Wilkes, Steve Muchnick, and Mike Sipser for their encouragement early in my graduate school years. These three were tireless in their efforts to assist me in my research as a new graduate student.

I have enjoyed my contacts with the professors, students, and staff at Berkeley, especially with fellow students Luigi Semenzato, Valerie King, Yanjun Zhang, Lisa Hellerstein, Sally Floyd, Alice Wong and Marshall Bern.

Finally, I would like to thank my wife, Linda Moya, for her advice and support these past years (and for many years to come).

This research was supported by the International Computer Science Institute, Berkeley, California, by the IBM Almaden Research Center, San Jose, California, by an IBM pre-doctoral fellowship, and by IBM Micro grant #442427-57449.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction and Terminology

Parallel computers are becoming an important component of the computing world. Computers with many processors provide both a cost-effective way to achieve increased performance over conventional computers and a means to overcome fundamental limits on uniprocessor performance. In conjunction with these new machines and in anticipation of future machines, considerable research effort continues in the areas of parallel algorithms, programming support, machine architectures, and VLSI technology.

In this thesis, we introduce a new model, the *Asynchronous PRAM*, for the design and analysis of algorithms that are suitable for parallel machines with hundreds to thousands to millions of processors. We present many new algorithms on the Asynchronous PRAM, as well as give evidence that supports the practicality of the model. In particular, we argue that the Asynchronous PRAM supports an effective programming model for many application domains, serves as a good basis for studying algorithms and complexity issues, and can be implemented efficiently in hardware.

We begin by defining the terminology used in this thesis to describe parallel computers.

We view a parallel computer or multiprocessor as a network of processors, coprocessors, memory banks, switches, and links. Each node of the network contains one or more switches, which connect incoming links to outgoing links. As we shall see later, switches can be either very simple or quite elaborate. For example, they may have buffers (input and/or output queues) and an arithmetic unit (ALU). Some of the nodes of the network (possibly all) contain processors, and some contain memory banks. Some may contain several processors and/or several memory banks. For example, a node can contain a cluster of processors that share a bus for their intra-cluster communication and a common coprocessor for their machine-wide communication (e.g. [Gup89]). Each processor in the parallel computer has a unique identification number that distinguishes it from all other processors.

Each external link in the network is used for sending messages between two switches re-

siding at distinct nodes, one or more bits at a time. Processor-to-processor and/or processor-to-memory communication is accomplished by sending messages that are routed between the source and destination nodes across links in the network, often passing through multiple switches on the way. Typically, special coprocessors at each switch take care of interprocessor communication: both programmers and processors are relieved of the burden of routing messages through the network, including handling the intermediate hops of messages traveling to their destinations. The interconnection network is most often a member of a class of networks of similar structure, e.g. delta networks, mesh networks, shuffle-exchange networks, or hypercube networks (see, e.g., [WF84] for definitions).

The memory banks that comprise the memory of the parallel computer can be grouped according to their sharability and locality. Memory banks that can be accessed by all processors comprise the **shared (or global) memory**. In contrast, memory that can be accessed by only one processor is its **private memory**. Memory banks in the same node as a processor comprise its **local memory**; other memory banks comprise its **non-local** memory. (Note that a memory bank which is local to some processor can be part of shared or global memory.) A processor typically has a fast, relatively small, private memory which is local. The access time (**latency**) to a memory bank for a processor depends in part on the delays of the links traversed and the number of hops in the path from the processor to the bank. Thus different memory locations will have different access times for a given processor. In particular, access to a local memory bank will be faster than access to a non-local bank.

Either each processor fetches its own instructions from memory (a **MIMD** machine) or all processors execute the same instruction, broadcast from a central control processor (a **SIMD** machine). Examples of MIMD machines include the IBM RP3 [PBG+85], the Intel iPSC [Sei85][AS88], Cedar [Gaj83], and the BBN Butterfly [BBN86][LSB88]. The Connection Machine [Hil85] is an example of a SIMD machine. In a **synchronous** machine, all the processors execute in lock-step, i.e. each processor executes its instruction i before any processor proceeds to its instruction $i+1$. In an **asynchronous** machine, the processors are not constrained to operate in lock-step.

Parallel computers can be distinguished by whether they support a shared memory or a message passing model of processor communication. A **shared memory parallel computer** is one in which the processors communicate by reading and writing data words to a global memory. The "messages" sent from a processor to a memory location (and back) are short and simple, and the machine is streamlined accordingly. In contrast, in a **message passing parallel computer**, the processors communicate by passing messages that can have rich semantics (such as in an object-oriented programming style). The messages are viewed as sent between processors, and they can be long and invoke complicated actions (beyond simply loading and storing a data word). The similarities and differences between

the two models are discussed in more detail in section 6.4.

A parallel computer can be distinguished from a **distributed system** (e.g. a network of workstations) in several ways. First, a parallel computer is packaged as a single machine, whereas a distributed system is not so physically compact. User terminals and other peripherals may be at distant locations from the processors of the parallel computer, but the distances between its processors, memory banks, and switches are at most several meters. Second, interprocessor communication is implemented with less overhead in a parallel computer. Hardware mechanisms are used to support the communication, without interference by the operating system or other software mechanisms. In distributed systems, on the other hand, interprocessor communication is over greater distances and between processors that are potentially under different ownership domains. Thus communication in distributed systems is slower due to the overhead of dealing with issues of security, reliability, and failure detection. Finally, because communication is relatively fast, there is a higher degree of cooperation between processors working to solve a single problem. The processors communicate more frequently (the ratio of computing steps to communication steps is relatively small) and messages can be short.

Throughout this thesis, we will discuss various models for parallel computers. We distinguish between two potentially overlapping types of models for the parallel computer:

- **Models of computation**, which define the view of the computer presented to the algorithm designer for estimating running times and comparing algorithmic choices (e.g. the PRAM model [KR88]).
- **Programming models**, which define the view of the computer presented to the application programmer for programming (e.g. the IBM EPEX model [DRGNP86]).

1.2 Bottlenecks in Large Scale Parallel Computing

Three primary hindrances to effective large scale parallel computing are the latency to global memory, contention in the network and at the memory banks, and synchronization overheads. We add to these a fourth hindrance: asynchronous communication. The first three affect the performance of the machine, and can limit the extent to which good speedup is possible. The goal is that a machine with p processors will solve a problem p times faster than a comparable uniprocessor. We believe that the fourth is just as important, since it affects the usability/programmability of these machines and can have some performance impact. In this section, we discuss each of these four bottlenecks in some detail.

1.2.1 Latency to global memory

In existing parallel machines and all machines of the foreseeable future, communication between the processors takes considerably longer than an operation local to a processor. In shared memory machines, the delay in accessing a non-local memory location can be a serious bottleneck in programs for which a medium to high percentage of the instructions involve non-local memory accesses [AI88]. There is hope that optical communication mediums (e.g. [GLKA84][McA89]) will reduce the latency, but, even still, a communication step is likely to take considerably longer than a local computation step. Furthermore, as the number of processors increases, the latency must increase since the volume of the machine increases.

This being the case, techniques are needed to help tolerate the high latency to global memory. One such technique is to pipeline global memory accesses, i.e. to permit each processor to have multiple requests to memory pending at the same time. If the average latency in an otherwise empty network is l machine cycles, then the goal is for a processor to complete a batch of k global memory accesses in $l + k$ cycles. In order to support this type of full pipelining for all processors, a simple counting argument shows that the network of the machine must have l switches and links per processor. (There are k messages per processor, each occupying a total of l switches and l links, to be satisfied in $l + k$ cycles. Thus a total of $kl/(l + k)$ switches and links are needed [Cyp88].)

1.2.2 Contention

Contention occurs when two or more memory requests each attempt to use the same physical resource at the same time. These conflicts occur in existing machines for the following reasons.

- There are many memory locations per memory bank, and each memory bank can support only a constant number of accesses per cycle.
- Even if each memory bank is requested by exactly one processor, the paths to memory can overlap. Since a link in the network can transmit only one message per cycle, messages competing for a link are serialized.
- Furthermore, unless special *combining networks* (described below) are used, multiple requests to the same location will be serialized. Unfortunately, such combining networks are more expensive, more complex, and slower than regular networks [PN85].

A **combining network** is a network in which messages at a switch that are destined for the same location are grouped together (*combined*) so that only one such message is sent on to the next switch. Combining networks are used to reduce network traffic and avoid serial bottlenecks when multiple requests are destined for the same location. The switches in the

network come in pairs: for each switch handling messages destined for the memory banks, there is a dual switch for handling replies returning to the processors. As multiple requests to read the same memory location travel on the way to the destination, each switch along the way combines the messages and informs its dual switch. When a reply (to a read message) returns from memory to the dual switch, the information saved for the dual switch is used to broadcast the value to precisely those processors requesting it. Sophisticated combining networks [PBG+85][Ran89] can perform operations on the data fields of the messages being combined, e.g. sending on a message whose data field is the sum of the data fields in the individual incoming messages. In some networks (e.g. [Ran89]), multiple messages that are destined for the same location are guaranteed to be combined. In others (e.g. [PBG+85]), the combining may or may not occur, usually depending on whether or not the messages reach a common switch at roughly the same time.

Contention in the network can slow down the processors considerably. An **oblivious routing** scheme is one in which the route of a message depends only on its source and destination. Consider a network of n nodes in which each node has one processor, one memory bank, and at most c input links. Borodin and Hopcroft [BH85] proved that for any deterministic oblivious routing scheme for the network, there is a set of n requests, one per processor, which, due to contention in the network, will take $\Omega(\sqrt{n}/c^{1.5})$ cycles to reach their destinations. This lower bound holds even if no two processors request the same memory bank and the largest distance between any two node is $\log_c n$ links. The two ways of avoiding this potential bottleneck are to use non-oblivious techniques and/or randomization. Examples of non-oblivious schemes include routing-by-sorting (in which the requests are sorted as part of the routing operation [CS88][NS81][RV87]) and adaptive wormhole routing [NS89]. In this latter scheme, messages attempt to follow a shortest path to their destinations. However, if an incoming message to a switch can not immediately proceed along its shortest path, it may be routed along a longer path instead. Upfal [Upf89] has developed a deterministic non-oblivious scheme, which is not based on sorting and has provably good asymptotic performance. However, it requires a special network topology, and the large constants of the running time make it impractical. Recently, Leighton and Maggs [LM89] have improved Upfal's scheme to make it considerably more practical. Examples of using randomization include the Valiant and Brebner [VB81] scheme of routing first to a random location and then on to the destination, the randomized sorting algorithm of Reif and Valiant [RV87], and the use of random hashing.

In **random hashing**, a hash function mapping program addresses to shared memory locations is selected at random from some fixed set of hash functions, and used to map the program address space to the physical addresses in the machine. Random hashing serves to scatter the locations in a random fashion across the memory banks, thereby reducing contention in the network and at the memory banks. In multi-user environments,

minimizing this contention reduces the extent to which one user can slow down another. Random hashing has also been used to obtain simple routing strategies with provably good performance [Ran87]. However, computing a hash function for each access to memory may be expensive, and the programmer can no longer exploit localities of reference and other regularities of communication. Furthermore, in order to achieve the theoretical bounds, the entire memory will need to be rehashed occasionally. An important open question is whether techniques that use random hashing can be practical. Research on the proposed Fluent machine [Ran89], which uses random hashing, will provide insight into this question.

1.2.3 Synchronization overheads

Interprocessor synchronization mechanisms include semaphores, shared locks, full-empty bits, and barriers (see, e.g., [Din89] for definitions of these and other mechanisms). We distinguish between two functions for synchronization primitives.

- **Ordering.** Synchronization primitives are used to enforce a predetermined ordering between two sets of program events, e.g. that a particular write to a shared memory location occurs ahead of a set of reads of the same location by other processors.

- **Arbitration.** Synchronization primitives are used to select a winner among processors competing for a shared resource, e.g. selecting which processor can increment a shared counter among those trying to update it.

A particular mechanism can encapsulate both functions. For example, a shared lock is first granted to one of the processors requesting it (arbitration) and then used to enforce exclusive access to the data protected by the lock (ordering). The mechanism ensures that the reads and writes of the protected data by the selected processor occur before the reads and writes of any subsequent processor.

Another example of a synchronization mechanism/primitive is the implicit synchronization associated with the lock-step execution of synchronous machines, especially SIMD machines.

The run time costs associated with synchronization in parallel computers can be significant due to the following five reasons.

- **Synchronization may be frequent.** In order to ensure that the processors are doing useful work and that shared resources are accessed in a consistent manner, synchronization may be needed frequently for both ordering and arbitration. In synchronous machines, synchronization occurs at each instruction.

- **Processors are forced to wait.** With ordering primitives, program events in the second (i.e. delayed) set must wait for those in the first set to complete. If the first

set involves many processors, then all the processors involved in the second set must wait for the last one in the first set.

- **Synchronization mechanisms are a source of serialization.** A set of requests for a single resource are typically serialized, i.e. the requests are satisfied one at a time.

- **Arbitration typically involves concurrent access.** Even in mechanisms where all but one request for a particular resource is discarded, having multiple requests directed to its arbitration hardware can lead to contention problems.

- **Synchronization mechanisms can be slow.** Synchronization can not be performed without interprocessor communication. Moreover, a synchronization primitive may consist of many steps, e.g. a barrier primitive involves synchronizing each processor with all the others. (Synchronization barriers will be discussed in detail in section 5.3.) Third, the arbitration process for a mechanism can be quite involved.

For these reasons, synchronization must be a primary concern of the designers of parallel computers, parallel programming models, and parallel algorithms. Certain synchronization mechanisms/primitives that are useful in the sequential or distributed worlds may not be appropriate for large scale parallel computers.

1.2.4 Asynchronous communication

Most existing parallel computers are asynchronous machines. Asynchronous machines have an advantage over synchronous machines in that they avoid making worst case assumptions on instruction completion time in the presence of varying instruction times and clock skew. Instruction completion times can vary due to network congestion, memory bank contention, operating system interference, and the relative speeds of register vs. cache vs. local memory vs. global memory access. Further variation arises due to the relative speeds of instruction execution: an add is much faster than a floating point multiply or a global memory access.

Asynchronous machines, however, present difficulties for hardware, software, and algorithm designers not present in synchronous machines. Dealing with the complexity of communication between many independent processors is a nightmare for programmers. Programming and debugging are very difficult due to the subtleties of dealing with nondeterministic orderings of events during program execution and a lack of simple, global states. Any desired orderings among program events must be explicitly enforced, typically through programmer-controlled shared locks and other synchronization primitives. Testing the correctness of such programs is almost impossible, and proving their correctness can be extremely difficult as well. Debuggers must be left on at all times [MC88][FLMC88], since any bug that arises in one run of the program may not reoccur for thousands of subsequent

runs. Furthermore, there are difficulties in adequately analyzing the time complexity of programs written in these models [CZ89]. Finally, in order to support efficient access to shared synchronization variables, expensive combining networks are required.

We refer to this problem as the **asynchronous communication** bottleneck. In this thesis, we argue for imposing a semi-synchronous framework for interprocessor communication on asynchronous machines. In particular, we support **repeatable** programs, in which the output and all intermediate results are the same every time the program is run on a particular input.

1.3 Overview of the Thesis

This thesis introduces the Asynchronous PRAM model of computation, a variant of the PRAM model [FW78] in which the processors run asynchronously and there is an explicit charge for synchronization. The model is motivated by concerns with each of the four bottlenecks of the previous section.

The focus of this work is on synchronization in large scale parallel machines. In particular, the purpose of this work is to address the following open problems.

- Develop a model suitable for designing algorithms and writing programs for large scale shared memory MIMD machines, a model good for both theory and practice.
- Design algorithms suitable for these machines for important problems.
- Study synchronization from a complexity theory standpoint, including the computational power of various synchronization assumptions.
- Investigate the implications to algorithms, programming, and hardware of imposing a more structured framework for interprocessor communication on asynchronous machines. In particular, a framework in which programs are repeatable.

The outline of this thesis is as follows. Chapter 2 describes some of the limitations of the PRAM model, and then formally defines the Asynchronous PRAM model. A family of Asynchronous PRAMs are defined, varying in the types of synchronization steps and the costs for accessing the global memory. Chapter 3 focuses on Asynchronous PRAMs with *all processor synchronization*, in which all the processors synchronize at each synchronization step. This chapter contains the bulk of the theoretical work. We modify a number of PRAM algorithms for improved asymptotic time and processor complexity in this model. We show general classes of problems (such as those in NC [KR88]) for which the time complexity can be improved by restructuring the computation. In addition, we prove lower bounds that reflect limitations on information flow and load balancing in this model. Finally, we

have simulation results between the Asynchronous PRAM and various known synchronous models.

Chapter 4 focuses on the Asynchronous PRAM with subset synchronization, in which processors can synchronize in sets of arbitrary size. We present algorithms, lower bounds, and simulation results for this variant of the Asynchronous PRAM. We introduce a *post office gossip game* for studying the inherent synchronization complexity of coordinating processors using pairwise synchronization primitives. Results are presented that compare the relative power of various such primitives.

In chapter 5, we broaden our study of asynchronous parallel computation from the strictly theoretical to the more practical issues of programming models and hardware. We introduce the notion of a *semi-synchronous* programming model, a model for *repeatable* asynchronous programs. We then present methods for supporting the synchronization primitives of the Asynchronous PRAM efficiently in hardware, including a new technique for barrier synchronization. A cache protocol suitable for the Asynchronous PRAM is presented as well. Chapter 6 evaluates the Asynchronous PRAM model and compares it to related work. Finally, chapter 7 presents conclusions and areas for further research.

Preliminary versions of some of this work appeared in [Gib88] and [Gib89].

Chapter 2

Formal Definition of the Model

2.1 Limitations of the PRAM

The **PRAM model of computation** consists of a collection of p sequential processors, each with its own private local memory, communicating with one another through a shared memory. The processors execute in lock-step, although each processor does have its own local program. A PRAM computation is a sequence of time steps, alternating between three types of instructions: read, compute, and write. In a read step, each processor can read one shared memory location into a private memory location. In a compute step, each processor can execute a single RAM operation whose operands are in private memory, storing the result in a private memory location. In a write step, each processor can write the contents of one private memory location into a shared memory location. All three steps are assumed to take unit time in the model. Although an idealized model, the PRAM has proven to be a useful model for studying parallel computation (see [KR88] for a survey of results). The model is simple and relatively easy to use: most of the details of interprocessor communication, memory management, and synchronization are hidden in the model.

There are several difficulties that arise in mapping PRAM algorithms onto existing shared memory MIMD machines. First, realistic MIMD machines have more limited communication capabilities than the PRAM. The PRAM assumes that each processor can access any shared memory location in one step. As discussed in section 1.2, realistic machines are more limited in that (a) a shared, non-local memory access takes much longer than a local operation, and that (b) further delays can occur due to contention in the network or at the memory banks. Considerable research effort has been focused on finding efficient ways to satisfy the simultaneous shared memory accesses of a PRAM read or write step on realistic networks (e.g. [KU88][Ran87]). Despite the success of these efforts to minimize the contention bottleneck, at least theoretically, the latency bottleneck remains. Recently, several researchers have explored variants of the PRAM that take into account the high latency to shared memory (e.g. [AC88][PY88]).

Second, existing MIMD machines are asynchronous whereas the PRAM is a synchronous model. Supporting a synchronous model, such as the PRAM, on an asynchronous machine is inherently inefficient since the ability of the machine to run asynchronously is not fully exploited and there is a (potentially large) overhead in synchronizing the processors as part of each instruction. For example, in Ranade's scheme for simulating a PRAM [Ran87][Ran89], each switch holds any shared memory accesses for an instruction until all shared memory accesses of the previous instruction are finished with the switch. Thus there is a cost for synchronization at each level of the network, even for instructions that do not require it.

Surprisingly, this important limitation of the PRAM has been largely ignored by the theoretical community, even by those doing research into making the PRAM more practical.

With these limitations in mind, we introduce the Asynchronous PRAM model, a variant of the PRAM model more suited to shared memory MIMD machines. In this new model, accessing the shared memory will no longer be a unit time operation and the processors will no longer execute their instructions in lock-step with each other. The time to read or write a shared memory location on existing MIMD machines depends on many factors, e.g. the distance from the processor issuing the request to the memory location itself. However, to keep the model simple, we will use a single parameter d to quantify the **communication delay** (i.e. latency) to non-local memory. This parameter is intended to capture the average or median ratio of the time for a shared write operation to the time for a local operation. Our second modification will be to permit the processors to run asynchronously and then charge for any needed synchronization.

Both of these new features can have a large impact on algorithm design. Many examples will be given after the formal model is defined. For now, we will present two examples. The first will demonstrate the effect of considering the communication delay to memory; the second will demonstrate the impact of explicitly charging for synchronization.

Consider the problem of broadcasting a single value in shared memory to all the processors. The EREW PRAM algorithm for this problem consists of fanning out copies of the value in a binary tree fashion. This algorithm runs in $O(d \log n)$ time when we account for the communication delay to memory. (Unless stated otherwise, all logarithms in this thesis are base two logarithms.) However, if d writes can be pipelined to complete in $O(d)$ time, then an improved strategy is to use a d-ary tree to fan-out the copies. At each level of the tree, each active processor reads a copy of the value and makes $d-1$ new copies (see figure 2.1). Each level takes $O(d)$ time, so the total time is $O(d \log_d n)$.

Explicitly charging for synchronization can also have a large impact on the analysis of an algorithm. Consider the problem of determining which node in a linked list is the head of the list. The PRAM algorithm for this problem is simple.

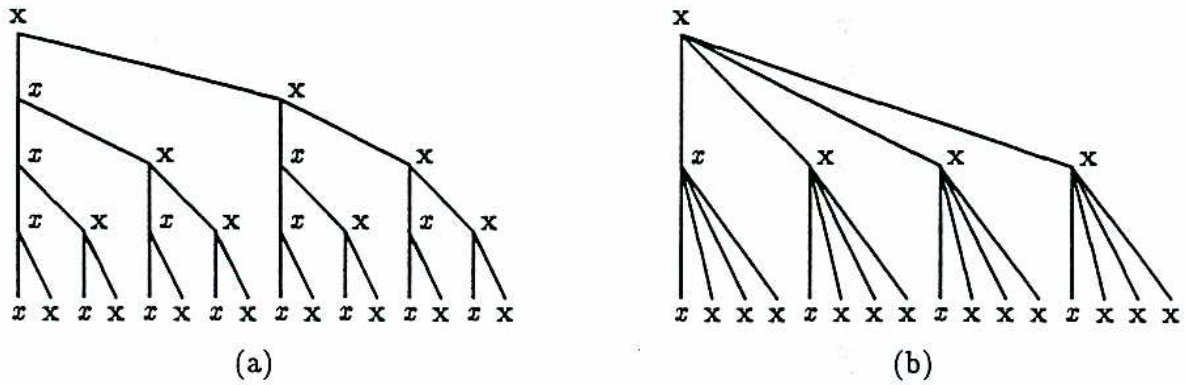


Figure 2.1: Accounting for communication delay when broadcasting a value. (a) The value x is broadcast to 16 processors in a binary tree fashion. (b) The value x is broadcast to 16 processors in a d-ary tree fashion, where $d = 4$.

Head_of_list algorithm:

This algorithm determines which node in a linked list has no predecessor.

1. Each node in the list notifies its successor.
2. The node who is not notified is at the head of the list.

◇

Pseudo-code for this algorithm is as follows.

Head_of_list program:

```
/*
inputs: A linked list of n nodes, where each processor is assigned to a node in the list.
outputs: The node at the head of the list is marked.
description: This program implements the algorithm described above. M_1, ..., M_n are
locations in the shared memory.
*/
for all processors i in parallel do {
    M_i := 0;
(1) if node i has a successor j
        M_j := 1;
(2) if M_i = 0
        mark node i as being the head of the list;
}
```

The Head_of_list algorithm runs in $O(1)$ time on a PRAM with n processors. However, if the processors run asynchronously and are subject to varying delays, then a processor

may not know the progress of its fellow processors. Let processor h be assigned to the node at the head of the linked list. In order for processor h to be sure that, indeed, no other processor will be writing to shared location M_h, processor h must synchronize with all its fellow processors before it can correctly test whether it is the head of the list. Without the synchronization, processor h can not distinguish the case where there is no writer from the case where the writer exists but has been delayed. Thus, in a model that charges, say, $\log p$, for synchronizing p processors, the running time is $O(\log n)$ when n processors are used.

2.2 Formal Definition of the Asynchronous PRAM Model

In this section, we formally define the Asynchronous PRAM model of computation.

The **Asynchronous PRAM model of computation** consists of a collection of p sequential processors, each with its own private local memory, communicating with one another through a shared memory. Each processor has its own local program. Unlike the PRAM, the processors of an Asynchronous PRAM run asynchronously, i.e. each processor executes its instructions independently of the timing of the other processors. Any desired timing dependencies between processors must be explicitly incorporated into the programs of the processors. There is no global clock.

A processor can issue up to one instruction per tick of its local clock. An instruction completes after some unbounded, but finite, number of ticks.

There are four types of instructions.

- **Global read.** Read the contents of a shared memory location into a private memory location.
- **Local operation.** Perform any RAM operation [AHU83] where the operands are in private memory and the result is stored in private memory.
- **Global write.** Write the contents of a private memory cell into a shared memory cell.
- **Synchronization step.** A synchronization step among a set S of processors is a logical point in a computation where each processor in S waits for all the processors in S to arrive before continuing in its local program.

The local program for a processor consists of a series of **phases** in which the processor runs independently, separated by synchronization steps. *All* instructions for processors in S prior to a synchronization step complete before *any* processor in S issues an instruction from its next phase. The set S may not be known at the beginning of the phase, e.g. it may be data dependent.

Processors can read and write to the shared memory asynchronously, but no processor may read the same memory location that another one writes unless there is a synchronization step involving both processors between the two accesses. Likewise, two writes of different values to a location by different processors must be separated by a synchronization step among the two processors. Thus there are no race conditions possible in the model. The varying delays for instruction completion do not affect the program computation. In particular, if a program that obeys this synchronization requirement is correct when all processors experience the same delays, then it is always correct.

2.2.1 Computation costs in the model

An Asynchronous PRAM program is correct only if it works regardless of any delays that may occur. As argued in section 1.2.4, varying delays are to be expected in MIMD machines. Nevertheless, these same machines are often tightly-coupled multiprocessors with regular networks and identical processors. Thus a reasonable first approximation to the behavior of one of these machines, for the purpose of measuring the cost of a computation, is to assume that the local clocks all run at the same speed. Given very similar local programs, the processors will progress through their programs at roughly the same rate. This motivates the following accounting scheme for our model.

We will estimate running times of programs assuming a global clock and a fixed cost for each instruction, independent of the processor. In particular, a local operation at a processor takes unit time. The cost for a global read or write instruction depends on the variant of the model, as will be described in section 2.2.2. A synchronization step among a set S of processors costs $B(x)$, a nondecreasing function of x, where $x = |S|$.

Recall that a local program for a processor consists of a series of phases separated by synchronization steps. The processor can not begin its next phase until the synchronization step has completed. The completion time for a synchronization step depends on the *last* processor to reach the step.

The completion time for an algorithm is defined inductively as follows. Initially, all processors begin their local programs at time zero. Inductively, consider a phase that is followed by a synchronization step among a set S of processors. The completion time for the phase for a processor in S (not counting the synchronization step) is defined to be the completion time for the processor's prior synchronization step plus the cost for its instructions this phase. The completion time for the synchronization step is the maximum completion time for the phase over all the processors in S plus the cost, $B(|S|)$, of the synchronization step itself. In this way, the completion time for a local program can be defined. The running time for an algorithm is defined to be the maximum, over all processors j, of the completion time for processor j's local program.

Remark. There are a wide variety of schemes for implementing synchronization steps on

MIMD machines, with varying run time overheads. (Some of these will be discussed in chapter 5.) This variability in machines requires us to define our algorithms parametrically, if we wish to have algorithms that are suitable for a wide range of machines. Given a particular target machine, a parameterized algorithm can be tailored to the machine by simply plugging in for $B(x)$ the estimated cost of synchronizing x processors on the machine.

2.2.2 A family of models

The Asynchronous PRAM defines a family of models that differ in the types of synchronization steps permitted, the cost of accessing the shared memory, and the extent to which concurrent reads or writes to a location are permitted.

- In an Asynchronous PRAM with **subset synchronization**, multiple disjoint sets of processors can synchronize independently and in parallel. The cost for a synchronization step among the processors in a set S is charged only to those processors in S. In an Asynchronous PRAM with **all-processor synchronization**, in contrast, synchronization steps must include all the processors. Multiple, parallel synchronization steps are not permitted. Three options are possible: (a) the set S must be all the processors in the machine, (b) the set S is all the processors assigned to the program, or (c) the set S is all processors currently active in the program. In this thesis, we will consider only the second case.

- An Asynchronous PRAM can either account for a communication delay to the shared memory or not. For the purpose of estimating execution time, we consider fixed communication delays: a global read takes $2d$ time and a global write takes d time. If communication delays are ignored, then both global reads and writes take unit time.

- An Asynchronous PRAM can either permit concurrent read and/or write or not. We have described thus far the **CRCW Asynchronous PRAM**. The other variants are more restrictive. In the **CREW Asynchronous PRAM**, any two accesses to a location by different processors, where at least one is a write, must be separated by a synchronization step among the two processors. In the **EREW Asynchronous PRAM**, any two accesses to a location by different processors must be separated by a synchronization step among the two processors.

In all cases, for the purpose of estimating execution times, we will assume that a processor can pipeline its instructions, i.e. it may issue instructions $i+1, i+2$, and so forth, of its local program before its instruction i has completed. (This assumption is irrelevant in Asynchronous PRAM's in which all instructions – other than synchronize – are assumed to take unit time.) The pipelining of instructions in a phase is limited only by the dependencies (if any) between the instructions. Interdependencies between instructions in a local

program arise in a sequence of reads if, for example, the value returned by one global read dictates the location to be read by the next global read (as in the case of traversing a linked list). The cost in the model to complete a sequence of r global read instructions with no interdependencies, issued one after another by a processor, is $2d + r - 1$. Likewise, the cost to complete w write instructions, issued one after another by a processor, is $d + w - 1$.

We make two further assumptions that simplify the model. First, any sequence of reads and writes issued by a processor to a location read and write the memory in the same order as they are dispatched. Second, any sequence of reads issued by a processor to different locations return to the processor in order. The former is true of machines that use FIFO buffers and links, while the latter can be simulated by buffering requests as they return, if necessary. Further discussion on support for such assumptions will be presented in section 6.3.7.

As discussed in section 2.1, d is a parameter quantifying the communication delay, i.e. the time to access (write to) memory. The communication delay d will increase with the number of processors in the machine. Note that on most machines, $B(x)$ is proportional to d or $d \log x$. Let p be the number of machine processors used by a program. In designing algorithms for the Asynchronous PRAM model, we assume only that $2 \leq d \leq B(x) \leq p$, where $2 \leq x \leq p$, unless otherwise noted.

Although only a simple variant of the PRAM, the Asynchronous PRAM is considerably more practical. Its primary advantages are that it permits asynchronous execution and it reflects some of the costs associated with synchronization and/or communication delay in real machines. Algorithms designed for the PRAM model tend to be far too fine-grained for real machines; algorithms designed for the Asynchronous PRAM model tend to be less fine-grained. Further discussion of the practicality of the Asynchronous PRAM model will be presented in chapter 6.

2.3 Comparison with Related Models

The literature to date reflects an overwhelming predominance of synchronous models for parallel computation over asynchronous ones. A large body of parallel algorithms and complexity theory results have been developed, particularly in the past ten years, entirely for synchronous models (e.g. the PRAM). There have been asynchronous models in the world of distributed computing for many years, but in these models, the parameters are typically the number of processes and the number of messages. The number of messages in a program is not a suitable complexity measure for tightly-coupled multiprocessors (this will be discussed further in section 6.3.10). Typical programming models for MIMD machines (e.g. for the IBM RP3 [PBG+85] and for the Sequent Balance [Seq86]) are asynchronous, but have no notion of costs.

The work presented in this thesis represents one of the first attempts to design an asynchronous model suitable for parallel computers and study it in detail. Other such models were developed independently by Kruskal, Rudolph, and Snir, and by Cole and Zajicek.

Kruskal, Rudolph, and Snir [KRS88] studied an asynchronous model for parallel computers based on an accounting scheme for asynchronous computation due to Lynch and Fischer [LF81]. In this scheme, each processor completes an instruction in one *local* clock cycle. Time is measured using the slowest processor clock. In one time unit or "round", each processor executes at least one instruction (the slowest executes one, faster processors execute more). Kruskal, Rudolph, and Snir show that a PRAM-like model with this accounting scheme can simulate a CRCW PRAM of p processors with $O(\log p)$ loss.

Later, Cole and Zajicek introduced the APRAM model [CZ89], based on the Lynch and Fischer accounting scheme as well. The APRAM permits multiple sets of processors to synchronize independently and in parallel, does not account for communication delay, and permits concurrent reads and writes. The goal in the APRAM model is to redesign algorithms so that processors synchronize in constant-size sets only: when this can be achieved, it leads to algorithms with the same time complexity as their PRAM counterparts. In addition, they define a measure, the *synchroneity* of an algorithm, that captures the extent to which slowing down a subset of the processors slows down the overall running time of the algorithm. They present APRAM algorithms for several basic problems such as the prefix problem (defined in section 3.2), and devise a sophisticated algorithm for computing the connected components of an undirected graph. This latter algorithm demonstrates the subtleties of devising correct algorithms in the APRAM model.

Very recently, Martel, Park, and Subramonian introduced another asynchronous model for parallel computers [MPS89]. In their model, the processors can have arbitrary asynchronous behavior, including arbitrary *unbounded* delays in executing instructions. Martel, Park, and Subramonian show how these delays can be overcome through the use of randomized allocation of work, as follows. A directed acyclic graph representing the tasks to be performed and the dependencies between them is placed in the shared memory. This graph has a single root node. Each processor selects a task at random, performs the task if its predecessors in the graph have been completed, and repeats. A processor halts when the root node indicates that the entire graph has been completed. In this way, processors that are delayed, or even fail, do not unduly slow down the computation: the faster processors will simply evaluate more nodes in the graph. In their more sophisticated algorithms, the graph is divided into phases in which the processors coordinate at the end of each phase and, in some cases, use binary search to find an unevaluated task within a block of tasks selected at random. The complexity measure for the model is the expected amount of *work* done, i.e. the total number of instructions executed by the processors. Their model permits

concurrent reads and writes. In fact, these operations are inherent in the model for two reasons. First, each processor frequently reads the value of the root node to determine if the computation has completed. Second, since tasks are selected at random without consulting other processors, more than one processor can select the same node and perform its task concurrently. The model also does not account for communication delay.

Many asynchronous algorithms have been developed for particular problems. Most of this work is tailored to specific machines and does not present a formal treatment of asynchronous parallel computation as is found in this thesis, e.g. the work does not include a formal model, general algorithmic techniques, lower bounds, and complexity comparisons with other models. Some formal work has been done, however. For example, Greenberg, Lubachevsky, and Odlyzko [GLO88] introduced a model for asynchronous parallel machines suitable for analyzing algorithms in which the inputs arrive staggered in time. In this context, the important metric is the average *response time* for a processor, namely the time from when its input is first ready until it completes its participation in the computation. They study the problem of finding the maximum of a set of inputs that arrive staggered in time. Algorithms are presented that achieve provably optimal response times. Their model does not account for communication delay and assumes that concurrent reads and writes can be performed in unit time (write conflicts are resolved by having a random processor succeed in writing its value).

In the area of iterative numerical algorithms, models for asynchronous parallel machines have been defined, but in these models, the important performance metric is the convergence rate of the iterative process. Other complexity measures are typically not addressed. For example, Lubachevsky and Mitra [LM86] present a model for iterative numerical algorithms where arbitrary delays occur, but are uniformly bounded by some finite value. Because of the delays, processors typically receive data computed several iterations before. Their main result is to prove fast convergence of an iterative algorithm (to compute the fixed point of an important class of matrices) despite the varying delays.

A model related to the Asynchronous PRAM, but synchronous, is a message-passing model with unbounded messages [Sni88]. In this model, each processor can pack a collection of values into a single message which it then sends to some other processor. Sending such a message is similar to issuing the set of global writes by a processor in an Asynchronous PRAM phase, since all such writes complete before any values can be read. However, in the Asynchronous PRAM model, the individual values that make up the set can be accessed by *different* processors in the very next phase, in contrast to the message-passing model.

The effect of communication delay on algorithm design has also been studied, in the context of synchronous models, by Aggarwal, Chandra, and Snir [AC88][ACS89] and Papadimitriou and Yannakakis [PY88]. Aggarwal, Chandra, and Snir introduced two synchronous models that are based on the PRAM, the *LPRAM* and the *BPRAM*. The LPRAM does not

permit the pipelining of memory requests, while the BPRAM permits blocks of consecutive memory locations to be accessed in pipelined fashion. In the model of Papadimitriou and Yannakakis, pipelining of memory requests is permitted. A more detailed model was studied by Gannon and Van Rosendale [GR84] in the context of numerical algorithms (e.g. algorithms for solving systems of equations). Variants of their model account for communication delay, pipelining rates, network bandwidth, and network topology.

Chapter 3

All Processor Synchronization: Algorithms and Lower Bounds

3.1 Introduction

In this chapter, we will focus on Asynchronous PRAMs with *all-processor synchronization.* A computation in this variant of the model is a series of global, program-wide phases in which the processors run asynchronously. These phases are separated by *synchronization barriers*, i.e. synchronization steps that are among all the processors (see figure 3.1).

We will refer to such Asynchronous PRAMs as either **Phase PRAMs** or **Phase LPRAMs**. The two models are identical except that the Phase PRAM charges unit time for global reads and writes, while the Phase LPRAM charges $2d$ for global reads and d for global writes. Recall, however, that an Asynchronous PRAM charges only $2d + k - 1$ ($d + k - 1$) for a sequence of k global read (write) instructions with no interdependencies, issued one after another by a processor.

Since all processors participate in each synchronization step, we can count time on a global phase-by-phase basis. Thus the time cost for a phase is the maximum, over all processors i, of the cost of the instructions executed by processor i during the phase. The running time for a program is simply the sum of the time costs for each phase plus $B(p)$ times the number of synchronization steps, where p is the number of processors used by the program. Since all processors participate in each synchronization step, for convenience we will denote $B(p)$ as simply B. In the results presented in this chapter, the only assumptions on the parameters d and B are that $2 \leq d \leq B \leq p$.

As discussed in section 2.2.2, various concurrent read/write policies are possible, which we summarize here.

- **EREW Phase PRAM or Phase LPRAM.** No two processors access the same location in a phase.

	processor 1	processor 2	...	processor p
phase 1	read x_1 read x_2 * *write to A*	read x_3 * write to B write to C		read x_n * * write to D
phase 2	read B * write to B	*read A* * write to D		read C *
phase 3	* read D *	write to C		read B read A * write to B

Figure 3.1: An example computation for an EREW Asynchronous PRAM with all processor synchronization. There are three phases, terminated by synchronization barriers. Each column contains the instructions executed by one processor. An asterisk (*) represents a local operation (i.e. an operation other than a global read, global write, or synchronize). A horizontal line represents a barrier synchronization. In phase one, for example, processor 1 reads shared memory locations x_1 and x_2, performs some local operation, and then writes to shared memory location A. Note that no two processors access the same location in the same phase. For example, the read of A is separated from the write of A by a synchronization barrier.

- **CREW Phase PRAM or Phase LPRAM.** Any number of processors can read the same location (as long as no processor writes to the location), but no two processors may write to the same location in a phase.

- **CRCW Phase PRAM or Phase LPRAM.** Any number of processors can read (write) the same location, as long as no processor writes to (reads from) the location in the same phase. For concurrent writes, the processors must all write the same value.

In what follows, we present algorithms, lower bounds, and simulation results for the Phase PRAM and Phase LPRAM. In section 3.2, we present a few preliminary lemmas and an example Phase LPRAM program. Section 3.3 presents algorithms for important primitive operations such as list ranking and multiprefix. Results for on-line load balancing are presented in section 3.4. Finally, section 3.5 presents techniques for simulating known synchronous models of parallel computation on the Phase LPRAM.

3.2 Preliminary Lemmas

We now present a few lemmas and an example program.

First we observe that any algorithm for the EREW PRAM can be adapted to run on the Asynchronous PRAM as follows: insert a synchronization barrier after each read or write step in the PRAM program. This forces the Asynchronous PRAM to execute in lock-step. A single PRAM instruction involving a read step, a compute step, and a write step can be simulated in $2d + B + 1 + d + B$ time on the Asynchronous PRAM, i.e. $O(B)$ time, since $d \leq B$. Thus a PRAM algorithm running in time t using p processors can be run on an Asynchronous PRAM in $O(Bt)$ time. We can simulate a PRAM using fewer Asynchronous PRAM processors as follows.

Lemma 1 *An EREW (CREW, common-CRCW) PRAM algorithm running in time t using p processors can be simulated by an EREW (CREW, CRCW) Phase PRAM or Phase LPRAM running in time $O(Bt)$ with p/B processors.*

The idea is to have each Asynchronous PRAM processor simulate B PRAM processors for a single PRAM instruction and then synchronize. This balances the time spent synchronizing with the time spent accessing memory and computing.

Lemma 2 *Lemma 1 is tight. There is a problem that requires $\Omega(Bt)$ time on a CRCW Phase PRAM or Phase LPRAM, where t is the time to solve the problem on a PRAM. Moreover, p/B processors are required by the Phase PRAM or Phase LPRAM to achieve this time, where p is the number of processors used by the PRAM.*

Proof. Consider the problem of determining which node in a linked list of n elements is the head of the list. The argument presented in section 2.1 shows that the Phase PRAM or Phase LPRAM requires $\Omega(n/p_0 + B(p_0))$ time to solve this problem with p_0 processors. The time is minimized when $p_0 = n/B(p_0)$. On the other hand, the Head_of_list algorithm of section 2.1 runs in $O(1)$ time on a PRAM with n processors. The lemma follows by setting $t = O(1)$ and $p = n$. □

As we shall see, for many problems, algorithms exist that achieve better results than the results obtained by synchronizing after each step of a PRAM algorithm.

3.2.1 Prefix sum

We begin with the prefix problem. Let $\oplus$ be an associative binary operation that can be computed by a processor in constant time. The *prefix problem on* $\oplus$ is, given n inputs $x_0, \ldots, x_{n-1}$, compute $y_i = x_0 \oplus \cdots \oplus x_i$ for each i, $0 \leq i \leq n-1$. The prefix problem on addition is also called the *prefix sum problem.*

For simplicity, we will describe only the first half of the prefix sum computation, in which the summation of the n input numbers is computed. We can compute the sum on an EREW PRAM in $O(\log_2 n)$ time, in a binary tree fashion. Using lemma 1 yields a Phase PRAM or Phase LPRAM algorithm that runs in $O(B \log n)$ time. This can be improved to $O(B \log n / \log B)$ time using a B-ary tree, where at each level of the tree, each active processor reads $B-1$ values, computes their sum, writes the result, and then synchronizes.

Summation program:

```
/*
inputs: The n input numbers are stored in shared memory.
outputs: The summation of the n numbers is stored to shared memory.
description: This program computes the summation using a B-ary tree, where at each
level of the tree, each active processor reads B − 1 values, computes their sum, writes the
result, and then synchronizes.
*/
for all processor in parallel do (
    for level := 1 to log_B n do (
        if left-most among your siblings at the current level of the B-ary tree (
            read from shared memory the values of siblings 2, 3, ..., B;
            sum the values of all B siblings;
            write the sum to shared memory;
        )
        barrier;
    )
)
```

Since the global reads can be pipelined, each level of the tree takes $2 + (2d + B - 2) + (B - 1) + d + B + 1$ time, i.e. $O(B)$ time. There are $\log n / \log B - 1$ levels, so the Summation program runs in $O(B \log n / \log B)$ time with n processors. We can achieve an optimal processor-time product as follows. Let $\tau = B \log n / \log B$. By initially having each processor sum τ inputs without synchronizing and then using a B-ary tree, n/τ Phase PRAM or Phase LPRAM processors suffice to achieve $O(\tau)$ time. This is the optimal number of processors to use. In fact, in the typical case where B is a strictly increasing function of p, using $p = n$ or $p = n/\log n$ processors would result in a slower algorithm. This reflects the realities of most real machines.

Similarly, this B-ary tree approach (the same algorithm as in [PY88]), can be used to solve any prefix problem.

3.2.2 Brent's scheduling principle

Before continuing to present new algorithms, we present two lemmas for simulating an Asynchronous PRAM with many processors on an Asynchronous PRAM with fewer processors. These lemmas correspond to known lemmas for the PRAM. We will refer to the processors of the Asynchronous PRAM being simulated as **virtual processors**, and the processors of the simulating Asynchronous PRAM as **machine processors.**

Lemma 3 *A Phase PRAM (Phase LPRAM) program using p_0 processors and running in time $t + B(p_0)s$, where s is the number of synchronization steps, can be simulated by a Phase PRAM (Phase LPRAM) using $p < p_0$ processors in time $O((p_0/p)t + B(p)s)$.*

Proof. For each phase, each machine processor simulates the instructions in the phase for p_0/p virtual processors and then synchronizes. □

In the remainder of this section, we prove a more general result, corresponding to Brent's scheduling principle for the PRAM [Bre74]. Brent's scheduling principle applies to synchronous models with no communication delay/pipelining. In the context of the Asynchronous PRAM, we need a revised definition of the *work* of an algorithm, a revised statement of the theorem, and a new proof.

Definition 1 *The* **work** *of a Phase PRAM or Phase LPRAM algorithm is the sum over all processors of the cost of the instructions performed by a processor in the algorithm, not counting synchronization barriers.*

Theorem 1 *A Phase PRAM (Phase LPRAM) program using p_0 processors, s synchronization steps, a total of x work, and $t + B(p_0)s$ time can be simulated by a Phase PRAM (Phase LPRAM) using $p < p_0$ processors in $O(x/p + t + B(p)s)$ time.*

As in lemma 3, the proof of this theorem involves simulating the virtual processors on a phase-by-phase basis. We begin with the following lemma.

Lemma 4 *Let n_w (h_w) be the work (time) for phase w of a Phase PRAM (Phase LPRAM) program using p_0 processors. Then the phase can be simulated in $\leq n_w/p + h_w$ time on a Phase PRAM (Phase LPRAM) using $p < p_0$ processors.*

Proof. We prove the lemma for the Phase PRAM. The extension to the Phase LPRAM is straight-forward since each machine processor simulates the instructions in a phase one virtual processor at a time.

Let I_j be the instructions performed in phase w by the virtual processor with the j^{th} largest cost in the phase. Thus $\text{cost}(I_1) \geq \text{cost}(I_2) \geq \cdots \geq \text{cost}(I_{p_0})$. Each sequence of instructions I_j in phase w can be viewed as a series of unit time steps. Let x_c be the number of virtual processors with at least c steps in phase w. Thus $n_w = \sum_{c=1}^{h_w} x_c$.

Let machine processor i be assigned to perform the instructions in I_i, I_{i+p}, I_{i+2p}, etc. Processor 1 has the most work and it performs $\lceil x_c/p \rceil$ unit-time steps which are the c^{th} step for some virtual processor. Thus the time for the phase on the Phase PRAM using p processors is

$$\sum_{c=1}^{h_w} \lceil x_c/p \rceil \leq \sum_{c=1}^{h_w} (x_c/p + 1) \leq n_w/p + h_w$$

□

Now we can prove theorem 1.

Proof. (of theorem) For each phase, each machine processor simulates the instructions in the phase for p_0/p virtual processors and then synchronizes. Let n_w (h_w) be the work (time) for phase w. Then $x = \sum_{w=1}^{s} n_w$ and $t = \sum_{w=1}^{s} h_w$. Let T be the total time for the simulating Phase PRAM or Phase LPRAM. By lemma 3.4,

$$T \leq \sum_{w=1}^{s} (n_w/p + h_w + B(p)) \leq x/p + t + B(p)s$$

□

Note that, as in Brent's scheduling principle, this theorem does not account for scheduling costs.

3.2.3 Phase PRAM vs. Phase LPRAM

Now we turn to the relationship between the Phase PRAM and the Phase LPRAM. We will need the following lemma.

Lemma 5 *A Phase PRAM (Phase LPRAM) program can be simulated with constant overhead by a Phase PRAM (Phase LPRAM) program in which each processor, in each of its phases, first performs all its global reads and local operations for the phase, then performs its global writes.*

This lemma follows from the observation that (a) the timing of the reads and writes within a phase does not affect other processors, and (b) the processor's local state can be simulated even if the writes by a processor are queued until the end of the phase.

Clearly a Phase PRAM can simulate a Phase LPRAM with no loss. The following lemma gives a sufficient condition for a Phase LPRAM to simulate a Phase PRAM with only constant overhead. Recall that a sequence of global reads with no interdependencies can be pipelined by a processor. Denote such a sequence of reads as an **oblivious sequence.** An **intraphase oblivious** algorithm is one in which, in each phase, each processor first issues an oblivious sequence of global reads (and waits for them to complete), then performs a sequence of local compute steps, and finally issues a sequence of global writes.

Lemma 6 *An intraphase oblivious Phase PRAM program running in time t with p processors can be simulated by a Phase LPRAM program running in time $O(t)$ with p processors.*

Proof. We show that the Phase LPRAM can simulate each phase of the Phase PRAM with constant overhead. Suppose the worst case processor for a Phase PRAM phase performs (an oblivious sequence of) r global reads, then l local operations, and finally w global writes. The time on the Phase PRAM for this phase and the subsequent synchronization step is $r+l+w+B$. The reads and the writes can be pipelined, so the time on the Phase LPRAM for the phase is $(2d+r-1)+l+(d+w-1)+B$, i.e. $O(r+l+w+B)$ since $d \leq B$. □

Remark. With this in mind, for convenience, we can focus on the simpler Phase PRAM, and not the Phase LPRAM. Most of our algorithms will be intraphase oblivious, and so the results will apply to the more realistic Phase LPRAM.

3.3 Algorithms for Important Primitive Operations

In this section, we analyze the complexity of various primitive operations that, along with parallel prefix, are used as building blocks for many parallel algorithms.

3.3.1 A lower bound for summing

We begin this section with a lower bound. Our result of the previous section for summing n numbers is optimal in the following sense:

Theorem 2 *Given n numbers, stored one per shared memory location, and the following four types of instructions: $L \leftarrow G, L \leftarrow L+L, G \leftarrow L$, and "synchronize", where L is a private cell and G is a shared cell, then the sum of n numbers on a CRCW Phase PRAM with this instruction set requires $\Omega(B \log n / \log B)$ time, regardless of the number of processors.*

Proof. Fix the number of processors p. We will show the running time for computing the sum is $\Omega(B(p) \log n / \log B(p))$. Let the fastest algorithm have s phases, of time $t_1, t_2, \ldots, t_s$.

In a phase of time t_i, each processor can at best compute the sum of t_i numbers, thus the number of partial sums reduces by a factor of t_i at best. So in order to produce the sum, we must have $n/(t_1 t_2 \cdots t_s) \leq 1$. It can be readily proved that the time $t = \sum_{i=1}^{s} t_i$ is minimized when $t_i = n^{1/s}$ for all i. Thus $t \geq s n^{1/s}$.

The running time T to produce the sum is $t + Bs$. Let $s = \alpha \log n / \log B$. If $\alpha \geq 1$, then

$$T \geq Bs \geq B \log n / \log B$$

If $\alpha < 1$, then

$$T \geq t \geq s n^{1/s} \geq \alpha B^{1/\alpha} \log n / \log B > B \log n / 2 \log B$$

In either case, the theorem is proved. □

This argument can be applied to any n input, m output associative function f, where (1) at least one of the outputs of f depends on all the inputs, and (2) the basic step permitted is combining two partial results to get one. The factor of $B/\log B$ occuring in this lower bound reflects the extent to which expensive synchronization (or communication delay) hinders information flow to a processor in our model.

3.3.2 Fast Fourier Transform

Consider the Fast Fourier Transform (FFT) problem. It is a well known fact that the FFT can be computed quickly in parallel using a communication pattern that is a butterfly graph of n rows and $\log n$ columns, where n is the number of inputs (see figure 3.2) [Ull84]. By simulating one column at a time, a PRAM can compute the FFT in $O(\log n)$ time with n processors. Likewise, an Phase PRAM or Phase LPRAM can simulate one column at a time, synchronizing after each column, to solve an FFT problem in $O(B \log n)$ time with n processors.

We can improve upon the running time as follows. We partition the columns of the butterfly into $\log n / \log B$ stages of $\log B$ consecutive columns each. By the structure of the butterfly, the value of each node in the last column of its stage depends on the values of B nodes in the last column of the previous stage. In particular, the value of each such node can be computed by a binary tree whose B leaves are in the last column of the previous stage and whose interior nodes are in the same stage as the node (see figure 3.2).

This leads to the following algorithm for n processors.

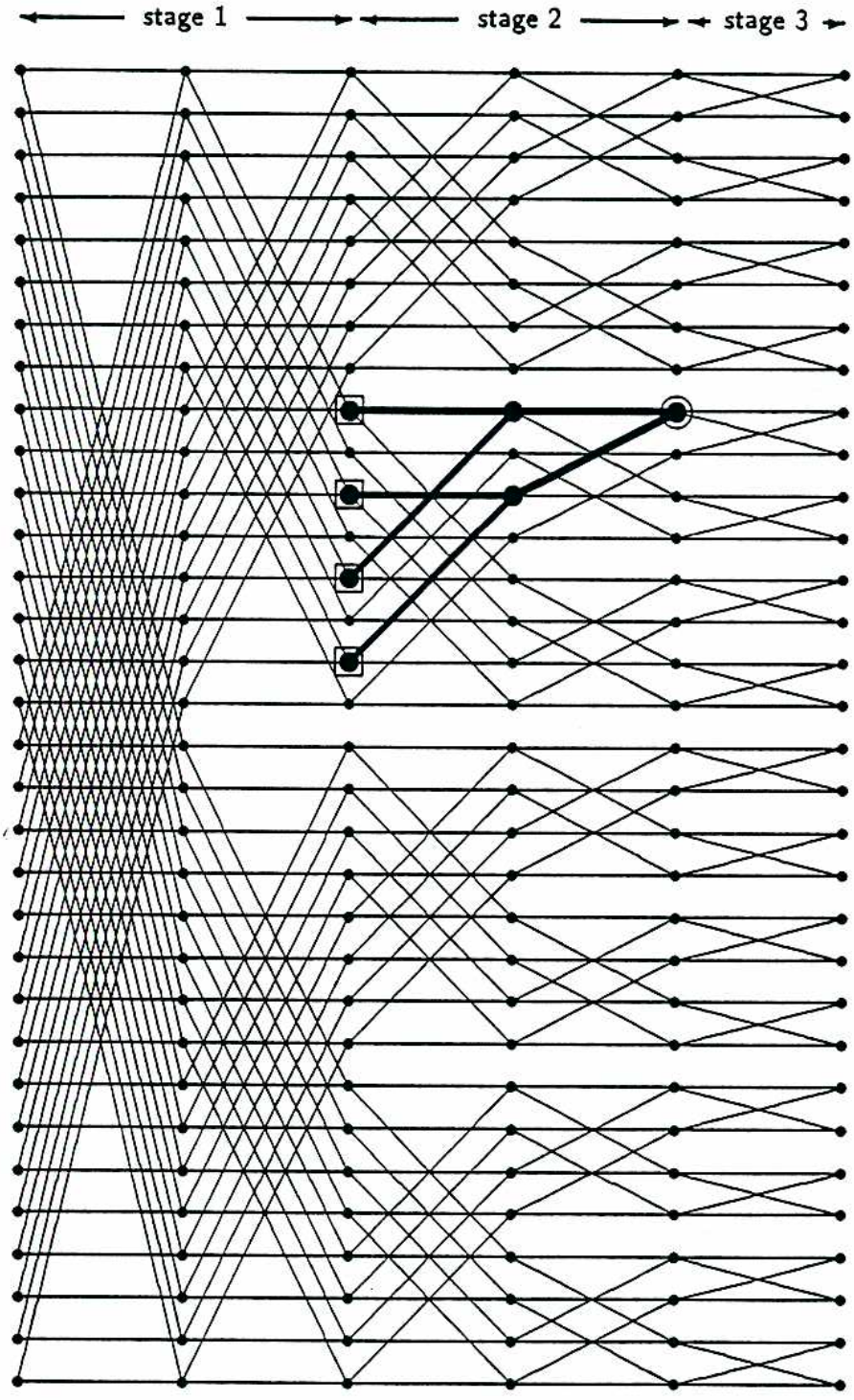


Figure 3.2: A butterfly graph on 32 nodes. The FFT algorithm for the Phase LPRAM divides the graph into stages of $\log B$ columns. Shown here are the stages when $B = 4$. The circled node is in the last column of stage 2. The binary tree for computing the value of the circled node is shown in bold, with its leaf nodes enclosed in squares.

FFT program:

/*

inputs: The n inputs are stored in shared memory.

outputs: The n result values from computing the FFT on the inputs are stored in shared memory.

description: This program computes the Fast Fourier Transform by having each processor, for each of $\log n/\log B$ stages, simulate $\log B$ consecutive columns of the butterfly graph. At each stage, processor i computes the value for the row i node in the last column of the stage.

*/

for all processors i **in parallel do** ⟨

  for level := 1 **to** $\log n/\log B$ **do** ⟨

    read from shared memory the B leaves for the row i node;

    compute the value at the node by simulating the binary tree for the node;

    write B copies of the value to the shared memory;

    barrier;

  ⟩

⟩

The extra copies are needed to avoid concurrent access to a location in a phase, since B processors read the value of any node that is in the last column of the node's stage. Each phase (i.e. each iteration of the while loop) takes $O(B)$ time, so the FFT program runs in $O(B\log n/\log B)$ time using n processors.

This algorithm can be improved to use only $n\log B/B$ processors with the same time complexity as follows. We partition the columns of the butterfly into $\log n/\log(B/\log B)$ stages of $\log(B/\log B)$ consecutive columns each. By the structure of the butterfly, the value of each node in the last column of its stage depends on the values of $B/\log B$ nodes in the last column of the previous stage. Moreover, the values of a set of $B/\log B$ nodes in the last column of their stage depend on the values of a set of $B/\log B$ nodes in the last column of the previous stage (see figure 3.2). A processor can mimic the butterfly graph of $B/\log B$ rows and $\log(B/\log B)$ columns in $O(B)$ time. There are $n\log B/B$ sets in a stage, so $n\log B/B$ processors suffice to achieve $O(B\log n/\log(B/\log B))$ time, i.e. $O(B\log n/\log B)$ time, on an EREW Phase PRAM or Phase LPRAM. This algorithm is similar to the algorithm due to Papadimitriou and Yannakakis [PY88] for computing an FFT on a synchronous model with communication delay d.

3.3.3 Bitonic merge

A sequence of elements over a totally ordered set is *bitonic* if it is the cyclic shift of a monotonically increasing subsequence followed by a monotonically decreasing subsequence.

For example, the sequence

$$\{12, 16, 23, 36, 45, 48, 42, 34, 28, 25, 20, 9, 8, 4, 6, 10\}$$

is a bitonic sequence. The *bitonic merge problem* is to sort a bitonic sequence. An algorithm for this problem can be used to merge two sorted lists, or as a subroutine for a bitonic sorting algorithm.

As with the FFT, a bitonic merge on n inputs can be performed quickly in parallel using a communication pattern that is a butterfly graph of n rows and $\log n$ columns. Thus the technique described above for the FFT problem can be used to solve the bitonic merge problem on a Phase PRAM or Phase LPRAM. For the bitonic merge problem, however, we can reduce the number of processors needed to n/B as follows.

We will partition the graph into $\log n / \log B$ stages, each consisting of $\log B$ columns. For each stage $j, 1 \leq j \leq \log n / \log B$, the rows of the butterfly graph can be partitioned into sets $S_{j,1}, S_{j,2}, \ldots, S_{j,n/B}$ of size B with the following property: for each stage j, the outputs of the rows in $S_{j,i}$ are the result of sorting the inputs to these same rows. Each stage will have a different partitioning of the rows. The rows in a set will be evenly spaced, but not adjacent (except for the final stage). All comparisons in a stage are between elements of rows that are in the same set (see figure 3.3).

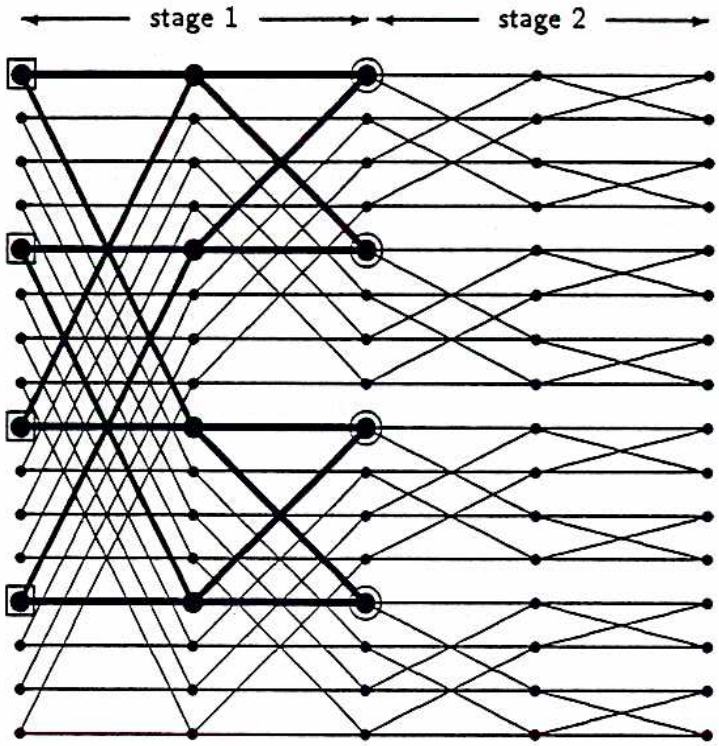
This leads to the following Asynchronous PRAM algorithm.

Bitonic_Merge program:

```
/*
inputs: A bitonic sequence of n elements is stored in shared memory.
outputs: The n elements are stored in shared memory in sorted order.
description: This program sorts a bitonic sequence by having each processor, for each
of log n/ log B stages, simulate log B consecutive columns of the butterfly graph. At each
stage j, processor i sorts the B elements corresponding to the "rows" for set S_{j,i}
*/
for all processors i in parallel do (
    for level := 1 to log n/ log B do (
        read from shared memory the B elements in S_{j,i};
        sort these B elements using a sequential sorting algorithm;
        barrier;

        write back to shared memory the B elements in sorted order;
        barrier;
    )
)
```

If the B elements in a set were in arbitrary order, the sorting step above would require



(a)

12	12	**8**	6	4
16	16	4	4	6
23	20	6	8	8
36	9	9	9	9
45	8	**12**	12	10
48	4	16	10	12
42	6	20	20	16
34	10	10	16	20
28	28	**28**	23	23
25	25	25	25	25
20	23	23	28	28
9	36	34	34	34
8	45	**45**	42	36
4	48	48	36	42
6	42	42	45	45
10	34	36	48	48

(b)

Figure 3.3: A bitonic merge computation. (a) The computation graph for a bitonic merge problem with 16 inputs. The graph is divided into stages of $\log B$ columns, where $B = 4$. The subgraph corresponding to the rows in the set $S_{1,4}$ is shown in bold. (b) The progression of intermediate values in the bitonic merge computation. The first column is the inputs and the last is the sorted outputs.

$\Omega(B \log B)$ time. However, the sorting step can in fact be done in $O(B)$ time, since the B elements in a set form a bitonic sequence and hence can be sorted using a linear time sequential merging algorithm. Thus the bitonic_merge algorithm runs in $O(B \log n / \log B)$ time on an EREW Phase PRAM or Phase LPRAM using only n/B processors.

This is an example of a general paradigm for saving processors in the Phase PRAM and Phase LPRAM by (1) using a known parallel algorithm for the problem to structure the computation, while (2) performing the individual steps by reading B values, running a sequential algorithm on these values, and then writing the results.

3.3.4 List ranking

Now consider the list ranking problem. The *list ranking problem* is, given a linked list of n elements, compute the distance of each element from the end of the list. Unlike the prefix problem of the previous section, it is no longer trivial to partition the work among the processors such that each active processor can do B useful operations (and avoid concurrent read) without synchronizing. However, the following pointer jumping approach achieves $O(B \log n / \log B)$ time on an EREW Phase PRAM with n processors.

Pointer Jumping algorithm:

Given a linked list of n elements, this algorithm reduces the list to a single element. In the statement of the algorithm, we omit the special handling required for processors that have encountered the tail of the list prior to the last round.

1. For each of $\log n / \log B$ rounds, repeat steps (2)-(4).
2. Processor i makes B copies in shared memory of the name of (pointer to) the current successor of element i, then synchronizes.
3. Processor i pointer jumps for B steps, visiting the successor of i (copy 1 from step (2)), then the successor of the successor of i (copy 2 from step (2)), etc., and then synchronizes.
4. Processor i writes in shared memory the new successor of i, i.e. the B^{th} such element visited, then synchronizes.

$\diamond$

This list-reducing algorithm can be used for list ranking by summing as it chases pointers. This shows how duplicating computation (here, B processors chase the same pointer – although a different copy – each phase) can reduce the running time, and duplicating the contents of selected memory locations can avoid concurrent read. From the proof of theorem 2, it is easy to see that the list-reducing (list ranking) problem has a lower bound of $\Omega(B \log n / \log B)$ time on a Phase PRAM.

Remark. This is not an intraphase oblivious algorithm. The Phase LPRAM running time is $O(B \log n / \log(B/d))$ on n processors.

A more sophisticated algorithm is needed in order to use fewer processors and achieve the same time bound on an EREW Phase PRAM. Known processor-efficient list ranking algorithms (e.g. [CV86][CV88][AM88]) communicate too frequently to run efficiently on this model, so a new algorithm is needed. Our new algorithm runs in three stages, and applies variants and/or generalizations of three known list ranking algorithms. Let $\tau = B \log n / \log B$. We will use $p = n/\tau$ processors.

Processor Efficient List Ranking algorithm:

Given a linked list of n elements, this algorithm reduces the list to a single element. There are three stages to the algorithm.

1. This first stage reduces the list from n elements to pB elements. The ideas behind this stage (we omit the full details) are as follows. The Anderson and Miller [AM88] EREW PRAM (deterministic) algorithm reduces a list from n to $n/\log n$ elements in $O(\log n)$ time using $n/\log n$ processors. The list elements are partitioned into $n/\log n$ queues of $\log n$ elements each. Each processor works on the elements in its queue, removing elements from the list according to a fixed arbitration scheme. Using a clever weighting scheme to analyze the progress of the algorithm, Anderson and Miller show that at most $n/\log n$ elements remain after $5 \log n$ rounds of their algorithm.

   Their analysis depends strongly on the ability to reallocate work each of $O(\log n)$ times, and hence their algorithm is too slow for our purposes. However, a careful examination of their algorithm and proof reveals the following generalization. By partitioning the list elements into n/x queues of x elements each, where $1 \leq x \leq \log n$, then n/x processors can reduce a list from n to n/x elements in $5x$ rounds of their algorithm. For our purposes, we take $n/x = pB$, i.e. $x = \tau/B$. At each of $O(x)$ rounds of the Anderson and Miller algorithm, each Phase PRAM processor simulates the work of B PRAM processors and then synchronizes. This takes $O(\tau)$ time.

   Finally, we *compact* the list of at most pB elements into a block of adjacent elements. This can be done in $O(\tau)$ time using the parallel prefix algorithm described in section 3.2.1.

2. The second stage reduces the list from pB elements to p elements. We use the deterministic coin tossing technique of Cole and Vishkin [CV86], with each Phase PRAM processor simulating B PRAM processors. We run the technique for $\log\log^*(pB)$ rounds (synchronizing after each round), and then compact. Then we run the technique for at most $\log B$ additional rounds (synchronizing after each round) until

we are left with at most p elements, and then compact. This takes $O(B \log B + B \log^* n \log\log^*(pB))$ time.

3. The third stage reduces the list from p elements to one element. This is done in $O(\tau)$ time using the Pointer Jumping algorithm given above.

$\diamond$

These ideas lead to an EREW Phase PRAM algorithm that runs in time $O(\tau + B \log B)$ with n/τ processors (the second term from stage (2) can be made not to dominate). This is an optimal algorithm (i.e. has an optimal processor-time product) for any parallel machine where $B \in O(2^{\sqrt{\log p}})$, in which case the running time for p processors is $O(n/p + B \log n / \log B)$, the same as for prefix sum.

Remark. The three stages in the above algorithm are used in an "accelerating cascade" manner [CV86], where earlier stages are processor efficient but make smaller progress, while later stages are less processor efficient but make greater progress. In order to achieve $O(B \log n / \log B)$ time, we could only "compact" the list a constant number of times.

Remark. For certain functions $B(p)$, e.g. $B(p)$ bounded above by $\log p$, the second stage can be omitted.

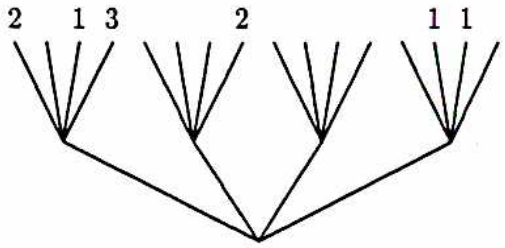
3.3.5 Multiprefix, integer sorting, and Euler tours

In the *multiprefix problem*, we are given n elements consisting of a value and one of L labels, and we want to simultaneously perform a prefix computation for each label. The parallel prefix algorithm of section 3.2.1 can be used to compute the prefixes for each possible label. If the labels are integers in the range of 1 to L, then Ln/τ processors suffice to achieve $O(\tau)$ time for a multiprefix problem of n data items, where $\tau = B \log n / \log B$. However, we can improve on this result for the case where $L \gg \tau$ using the following algorithm. To simplify the description of our algorithm, we will consider the case where the associative operation to be performed is addition. We will use n processors in all.

Consider a forest of L trees, one for each possible label. Each tree is an (implicit) complete B-ary trees on n leaves. As in the parallel prefix algorithm, the algorithm below computes a sum by progressing level-by-level up these trees. The difference here is that we have only p_j processors working on the tree for label j, where p_j is the number of data items with label j. Processor i starts at leaf i of the tree for its label and works its way up the tree until it encounters a smaller-numbered processor with the same label. The smallest numbered processor with a label l succeeds in reaching the root of its tree and thereby computing the sum of all the data items with label l (see figure 3.4).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
label	**L3**	L7	**L3**	**L3**	L4	L7	L1	**L3**	L1	L6	L9	L9	L4	**L3**	**L3**	L9
value	2		1	3				2						1	1	
sum	2		3	6				8						9	10	

(a)



(b)

P0		P2	P3
2	0	1	3
2	0	1	3
2	0	1	3
2	0	1	3

P7			
0	0	0	2
0	0	0	2
0	0	0	2
0	0	0	2

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

P13	P14		
0	1	1	0
0	1	1	0
0	1	1	0
0	1	1	0

P0	P7		P13
6	2	0	2
6	2	0	2
6	2	0	2
6	2	0	2

(c)

Figure 3.4: A multiprefix computation. (a) A multiprefix problem on $n = 16$ inputs, and its desired output. For simplicity, only the values and prefix sums relevant to label L3 are shown. (b) The multiprefix algorithm for the Phase LPRAM uses implicit B-ary trees for each label. Shown here is the tree for label L3, where $B = 4$. Each of six processors start at the appropriate six leaves and work together to compute the prefix sums for label L3. (c) The prefix sum for L3 is computed according to the algorithm. The intermediate results at both levels are shown.

Multiprefix algorithm:

This algorithm computes the multiprefix of n numbers stored in shared memory.

1. We repeat steps (2)-(3) once for each level in the forest of L trees, a total of $\log n / \log B$ times. Initially, all processors are active and the cumulative sum for a processor is simply the value of its element.

2. Consider a $B \times B$ matrix for each node of the next level in the forest. Each active processor writes its current cumulative sum in each row of a *column* of the matrix for its parent, and then synchronizes. Columns such that no processors have the appropriate label will remain untouched (all zero).

3. Each active processor sums its *row* in the matrix and continues to the next level of the tree (i.e. remains an active processor) if and only if it is the left-most processor with this label among its $B - 1$ siblings at the current level of the tree.

4. Reverse the process (steps 1-3) to yield the partial sums for each label.

$\diamond$

The Multiprefix algorithm runs in $O(B \log n / \log B)$ time on an EREW Phase PRAM or Phase LPRAM using n processors and $O(LnB)$ memory cells. Further refinements reduce the memory requirements to $O(\lceil L/B \rceil n)$ cells and include initialization costs, while maintaining the same time and processor bounds for any integer value of L. We omit the details here.

An algorithm for multiprefix yields an algorithm for sorting n integers of $k \log n$ bits each [Ran89]. It runs in $O(kB \log n / \log B)$ time on an intraphase oblivious EREW Phase PRAM with n processors. Given an n node forest represented by each node having a pointer to its parent, a multiprefix algorithm can be used to compute the index of each node among the nodes with the same parent. From this, we can construct an Euler tour of a tree and then compute many tree functions such as preorder and postorder numberings [TV85], all within the same resource bounds (using the List Ranking algorithm of section 3.3.4).

Generalized multiprefix

We now consider a generalized multiprefix problem in which we are given n elements consisting of a value, one of L labels, and a tag, where tags are integers in the range 0 to $r-1$, $r \geq 2$, and no two elements with the same label have the same tag. The problem is to compute a prefix computation for each label. We will focus on the case where $L \gg B \log n$ and there are $p \leq n$ processors.

As before, we consider a forest of L trees, one for each possible label. The time to progress level-by-level up a k-ary tree of r leaves is $O(k \log r / \log k)$, not counting synchronization steps or communication delay. If $n/p \geq B$, then binary trees ($k = 2$) are used.

Each processor spends n/p time performing one step for each of the items it represents and then synchronizes. This yields an $O((n/p)\log r)$ time EREW Phase LPRAM algorithm. If $n/p \leq B$, then we choose $k = B/(n/p)$, so that each level of the tree, each processor performs a total of B steps and then synchronizes. This yields an $O(B\lceil \log r/\log(Bp/n)\rceil)$ time EREW Phase LPRAM algorithm. This generalized result will be used in the proof of theorem 7 in section 3.5.2.

3.4 Upper and Lower Bounds for Load Balancing

In this section, we present a scheduling problem that arises in the context of on-line load balancing in the Phase PRAM, and which has more general applications. Given p processors and k jobs of unknown (nonzero) duration, find a preemptive on-line schedule such that cost B is charged each time the processors are scheduled, or preempted and rescheduled. Let h_i be the (unknown) number of unit-time steps in job i, let $n = \sum_{i=1}^{k} h_i$, and let $h = \max(h_1, \ldots, h_k)$. A lower bound on the completion time is $\max(n/p, h, B)$.

The generalization of Brent's scheduling principle to the Asynchronous PRAM presented in section 3.2.2 can be viewed as an off-line strategy for this problem. In particular, if the durations of the jobs are known (and sorted: $h_1 \geq h_2 \geq \cdots \geq h_k$), then, by theorem 1, the following *nonpreemptive* schedule achieves $n/p + h + B$ completion time: assign processor i to complete jobs $i, i+p, i+2p$, etc.

When the durations of the jobs are unknown, then the following simple strategy achieves within an $O(\log B/\log\log B)$ multiplicative factor of optimal. As we shall see later in this section, this strategy is (asymptotically) the best possible deterministic strategy.

We will view each job as a linked list of nodes, where each node represents a unit-time step. In unit time, each processor can "visit" the first node in some list, i.e. remove the node from its list, perform the associated unit-time step, and label the list as now *dead* (empty) or still *live* (not empty). We permit at most one processor to be assigned to any one list at a time.

Balanced algorithm:

This algorithm gives an on-line, preemptive schedule for completing a collection of jobs on p processors. There are three stages to the algorithm, each potentially involving many steps.

1. Repeat the following while the number of live lists is $\geq pB$: partition the live lists evenly among the processors, and have each processor visit one node in each of its lists.

2. Repeat the following while the number of live lists is $> p$. Partition the live lists evenly among the processors. Each processor cycles through its lists, visiting one node from a list at a time, until it has visited B nodes or emptied all its lists.

3. The number of live lists is now at most p. Assign one processor per live list and have it visit all the nodes remaining in the list.

$\diamond$

Theorem 3 *The Balanced algorithm achieves an* $O(\max(n/p, h, B) \log B / \log\log B)$ *completion time.*

Proof. We focus on the second stage and ignore floors and ceilings (we omit the extensions to include the first and third stages, which are within a constant factor of optimal).

Let m be the number of rounds in the second phase. Let k_i be the number of live lists per processor at the start of round i, $1 < k_i < B$. Thus the following restrictions on the values of t, n, and h must hold.

t: Each round is completed in $2B$ time (counting the B for rescheduling). Thus the completion time is $2mB$.

n: In round i, each processor visits at least B/k_i nodes in each list that remains live for round $i+1$. Thus the total number of nodes visited in round i, $i < m$, is at least pBk_{i+1}/k_i. At the end of round m, there are at most p live lists.

h: The number of nodes in the longest list is at least $B \sum_{i=1}^{m} 1/k_i$.

The worst case ratio r of the completion time to the optimal time (i.e. $\max(n/p, h, B)$) is

$$\begin{aligned} r \;\leq\; & \max_{k_i,m}\{t/\max(n/p,h,B)\} \\ & \max_{k_i,m}\{\min(t/(n/p), t/h, t/B)\} \\ & \max_{k_i,m}\{\min(\; 2mB/(B\textstyle\sum k_{i+1}/k_i), \\ & \qquad\qquad 2mB/(B\textstyle\sum 1/k_i), \\ & \qquad\qquad 2mB/B)\} \\ =\; & \max_{k_i,m}\{2m \min(1/(\textstyle\sum k_{i+1}/k_i), 1)\} \end{aligned}$$

where

$$B > k_1 \geq k_2 \geq \cdots \geq k_m > 1$$

For any fixed m, r is maximized when

$$k_{i+1}/k_i = 1/x$$

for all i. Thus $m = \log_x k_1$ and so

$$m/(\textstyle\sum(k_{i+1}/k_i)) = x$$

Thus r is maximized when $x = \log_x k_1$, i.e. when

$$r = x = O(\log k_1 / \log\log k_1)$$

Since $k_1 < B$, the theorem holds. $\square$

Theorem 4 *Any deterministic strategy will be* $\Omega(\log B/\log\log B)$ *from optimal on some set of jobs.*

Proof. Let there be pB lists and let $\alpha = \log\log B/\log B$. We assume that all processors visit up to B nodes and then reschedule (we omit the simple extension of the proof to the case where a processor visits more than B nodes before rescheduling). Let m be the number of rounds, and let k_i be the number of live lists per processor at the start of round i.

To foil the strategy, the adversary decides when to kill off a list (i.e. make a list have one node remaining). The adversary kills off lists according to the following three rules.

1. Each round, the adversary will kill off all lists assigned to the $(1-\alpha)p$ processors with the fewest lists assigned.

2. In addition, the adversary will kill off any list that is visited αB times in one round. A processor can cause at most $1/\alpha$ lists to be killed this way.

3. As soon as $k_i < 2/\alpha$, the adversary kills off all the remaining live lists.

First observe that

$$k_{i+1} \geq \alpha(k_i - (1/\alpha)) \geq \alpha k_i/2$$

Thus (after some arithmetic) we see there will be $\Omega(\log_{2/\alpha} B) = \Omega(1/\alpha)$ rounds.

We use the following accounting scheme for counting the total number of nodes visited in a round: the final node on a list is *not* charged to the round in which it was visited; instead, we will add a one time charge of pB to account for these "final" nodes. Then the number of nodes visited in a round is at most αpB (since there are only αp active processors). Thus the following restrictions on the parameters t, n, and h must hold.

t: The completion time is $\Omega(B/\alpha)$.

n: The number of nodes visited is $O((1/\alpha)\alpha pB + pB) = O(pB)$.

h: The number of nodes in the longest list is $O((1/\alpha)\alpha B) = O(B)$.

Hence the strategy is at least $t/(\max(n/p, h, B))$ from optimal, i.e. a multiplicative factor of

$$\Omega(1/\alpha) = \Omega(\log B/\log\log B)$$

from optimal. □

In contrast, if the Balanced algorithm is modified to *randomly* partition the lists among the processors, it achieves within a constant factor of optimal [Kar88].

Remark. This "list" scheduling problem generalizes to scheduling on a tree of unknown shape. This latter problem has applications in parallel branch-and-bound. We will not discuss the tree scheduling problem in this thesis, except to remark that an adversary can

force any deterministic or randomized strategy (that assigns at most one processor to a node) to be $\Theta(B)$ from optimal as follows. At each round, the adversary kills off all the frontier nodes of the tree but one, and then has this one remaining frontier node branch out to pB children.

3.5 Comparisons with Synchronous Models

In this section, we study the computational power of the Phase LPRAM relative to existing synchronous models of parallel computation. In particular, we present techniques for the Phase LPRAM to simulate DAG-based computations (formally, uniform circuit families of bounded-fanin arithmetic circuits), straight-line code, the CRCW PRAM model, and the BPRAM model. Each of these simulations improves upon the time complexity of the brute force technique (namely, the technique of synchronizing the Phase LPRAM after every step of the synchronous model).

3.5.1 Simulation of bounded-fanin circuits

Our first simulation can be used to improve the time complexity of DAG-based computations. We will begin with a less formal description of the technique and conclude with a more formal description.

Our informal description makes use of the following definition.

Definition 2 *A* **computation graph** *is a directed graph G such that (a) the nodes of G are labeled with either input values, unary operators, or binary operators, and (b) there is a directed edge in G from one node to another if and only if the former node computes an operand of the operation at the latter node.*

Thus input nodes have indegree zero, unary nodes have indegree one, and binary nodes have indegree two.

Recall that the natural computation graph for the prefix sum problem was a binary tree of n leaves fanning into one node followed by a binary tree fanning out to n leaves. The natural computation graph for the FFT problem was a butterfly graph.

In both cases, we were able to restructure the computation to give improved time complexity on the Phase LPRAM. These were examples of a general technique that can be applied to known (synchronous) algorithms that can be viewed as a family of computation graphs with the following properties:

- there is one computation graph for each input size,
- the computation graph is acyclic (a DAG), and
- each node has (indegree and) outdegree at most two.

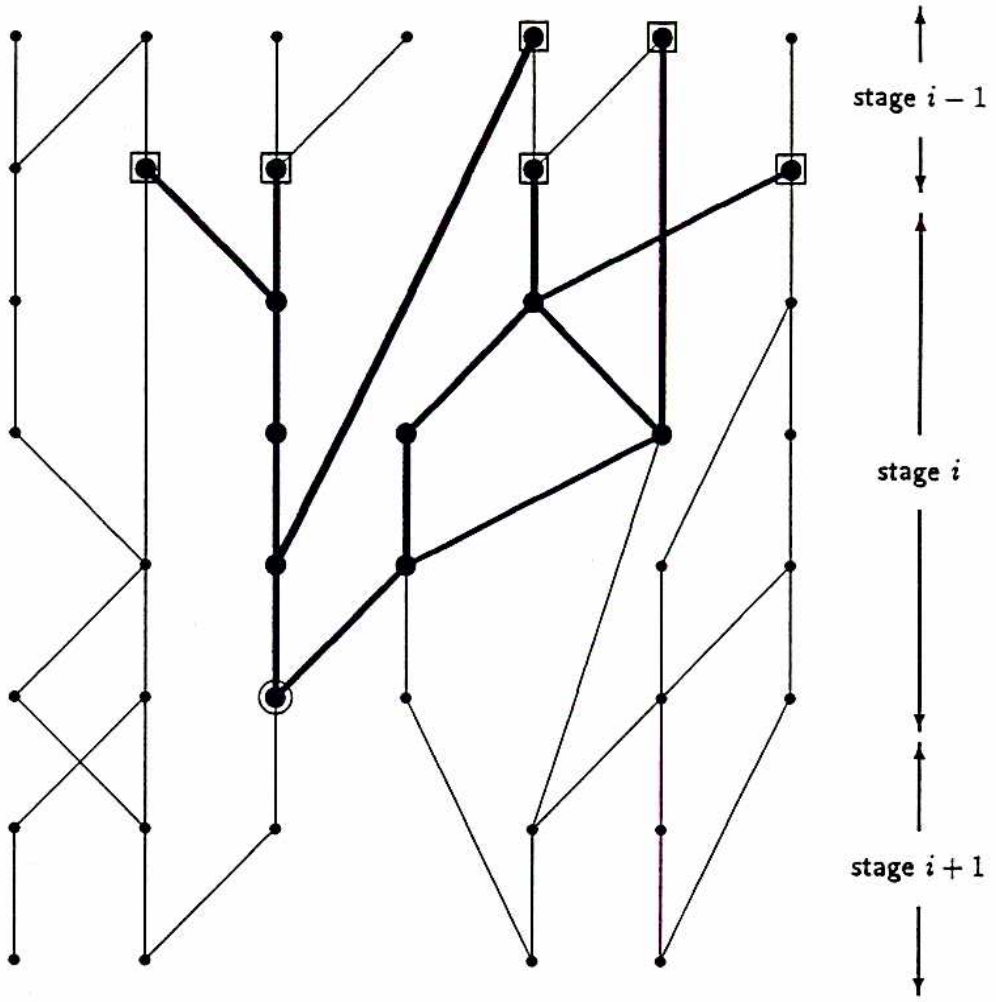


Figure 3.5: Simulating a bounded-fanin circuit. Stage i of an example computation graph is shown, where $B = 16$. The edges of the graph are directed downward (not shown). The value of the circled node is computed by evaluating the subgraph shown in bold. The Phase LPRAM processor for this node reads from global memory the values of the six leaves for the node (shown here as square nodes), and then computes the value using local operations that mimic the subgraph.

The *level* of a node in a computation graph is the number of arcs in the longest path from an input to the node. The *depth* of a computation graph is the maximum level of any node in the graph. The *width* of a computation graph is the maximum number of nodes at any one level.

As in the FFT algorithm, in the general technique we will partition the computation graph into stages of $\log B$ consecutive levels each. The value of each node in a stage depends on at most B predecessors in the graph from earlier stages (since there are $\log B$ levels in a stage and the fanin is at most two). This value can be computed by a DAG whose (up to) B leaves are in previous stages and whose interior nodes are all in the same stage as the node (see figure 3.5).

This leads to the following algorithm.

DAG Evaluation program:

```
/*
inputs: The values of the inputs to the DAG are stored in shared memory.
outputs: The values of the outputs computed by the DAG are stored in shared memory.
description: This algorithm gives a general technique for evaluating a DAG.
*/
for all processor i in parallel do (
    while more stages of log B consecutive levels in the DAG (
        /* Processor i computes the value for the i^th node in the current stage. */
        read from shared memory the up to B leaves for your node;
        compute the value at the node using local operations that mimic the DAG;
        write B copies of the value to the shared memory;

        barrier;
    )
)
```

If the computation graph for inputs of size n has depth $D(n) \geq \log n$ and width $W(n)$, then the DAG Evaluation program above runs in $O(B\ D(n)/\log B)$ time with $W(n)\log B$ processors. This represents an improvement by a factor of $\log B$ over the running time resulting from simply synchronizing at each level of the computation graph.

The above technique can be formalized as follows. (See [KR88] for a discussion of uniform circuit families.)

Theorem 5 *Any function computable by a log-space uniform circuit family of bounded-fanin arithmetic circuits of depth $D(n) \geq \log n$, maximum width $W(n)$, and polynomial size can be computed by a log-space uniform EREW Phase LPRAM family running in $O(B\ D(n)/\log B)$ time with $W(n)\log B$ processors.*

Proof. In what follows, we will first present the proof for simulating the circuit family by a CREW Phase LPRAM family that has one processor per circuit gate. The approach will parallel the technique used in the DAG Evaluation program. Then we will show how the technique can be enhanced to use fewer processors and avoid concurrent reading.

By a result of Hoover, Klawe, and Pippenger [HKP84], we can assume w.l.o.g. that the circuit has both indegree and outdegree bounded by two. Let M_c be a log-space Turing machine that generates a description of the arithmetic circuit, starting with the output nodes and proceeding level-by-level back to the inputs. The description for a node in the circuit is a four-tuple, (*id, op, left_child, right_child*), where the i^{th} node generated has id

i, *op* is the unary or binary operation for the node, *left_child* is the id of the first operand, and *right_child* is the id of the second operand (if any).

A Phase LPRAM program that computes the same function as the circuit can be generated uniformly in logarithmic space as follows. On input n, the following log-space Turing machine M_p can generate a description of the Phase LPRAM program, one processor at a time, starting with its final program instructions. Encode the transition function for M_c into the transition function for M_p in such a way that M_p can simulate M_c. We will handle the nodes in order. We maintain a counter of the level of the current node in the circuit, where the *level* of a node i is the length of the longest path from node i to an output. We view the circuit as divided into *slices,* or stages, of $\log B$ levels.

For each gate in the current slice, we simulate M_c repeatedly to perform a depth first search starting at the node and proceeding to the first node on each path which is outside the slice (a frontier node). We output the program for the processor (in reverse order) as (1) synchronize, (2) global write of its final value, (3) series of local operations which mimic gates within the slice, and (4) global reads of the values for the frontier nodes. Finally, if the gate is in the j^{th} slice (from the final slice), we output $j-1$ additional *synchronize* instructions in order that the processor starts at the proper time.

Note that some operations may be repeated in the resulting Phase LPRAM program, even by the same processor. Memory locations used in the Phase LPRAM program correspond to the unique node id's.

This can all be done in logarithmic space as follows. A depth first search from a node x is performed by maintaining a bit vector indicating, at each level, whether a left or right child was used to reach the current node in the search. This vector requires $O(\log B)$ space, which is $O(\log n)$. To search forward on a child y, simulate M_c until the node for y is generated. To backtrack from a node z at level l, simulate M_c starting from x and following the first $l-1$ bits of the vector to reach the parent of z. We maintain the minimum id of a child of a node in the current level. When this child is generated, we increment the level counter. The level counter is used to detect frontier nodes. It is also used to keep track of the current slice number, in order to output the correct number of *synchronize* instructions per processor.

The resulting CREW Phase LPRAM program will run in time $O(B\ D(n)/\log B)$, since there are $D(n)/\log B$ slices and each slice takes $O(B)$ time.

To reduce the number of processors to $W(n)\log B$, we use processor i to simulate the i^{th} node of each slice. The Phase LPRAM program can be generated one processor at a time. If there are fewer than $W(n)\log B$ nodes in a slice, then the code generated for a slice of a processor who has no work in the slice is just a *synchronize* instruction. Alternatively, the Phase LPRAM program can be generated slice-by-slice, where the width $W(n)$ is computed prior to the first slice.

Each value computed for a node will be used by up to $2B-2$ processors. To avoid this concurrent reading in the Phase LPRAM program, we have each processor generate $2B-2$ copies of its output value. Consider a node r in the circuit and the processor p assigned to r. For each frontier node w of r, have processor p read copy i of the output value of node w if node r is the i^{th} lowest numbered node that has w for a frontier node. The number i can be determined in logarithmic space by simulating M_c beginning with node 0 and continuing to node r, and counting the number of nodes that have w for a frontier node. In this way, each reader of a node w can be assigned a unique copy to read. □

A corollary to this theorem relates the well-studied complexity classes NC^k to the Phase LPRAM. A language is in $\mathbf{NC}^k$ if it is recognized by a log-space uniform circuit family of polynomial size and $\log^k n$ depth [Pip79][KR88].

Corollary 1 *Any language in $NC^k, k \geq 1$, can be recognized by a log-space uniform EREW Phase LPRAM family running in $O(B \log^k n / \log B)$ time with a polynomial number of processors.*

Remark. By theorem 5, we see that the Ajtai, Kolmos, and Szemeredi (AKS) sorting network [AKS83] can be simulated on a Phase LPRAM in $O(B \log n / \log B)$ time using $n \log B$ processors.

Remark. Theorem 5 also yields an algorithm for matrix multiplication, based on the fast matrix multiplication algorithm of Coppersmith and Winograd [CW87], that runs in time $O(B \log n / \log B)$ on an EREW Phase LPRAM with $O(n^{2.376})$ processors.

There is also a simple algorithm for matrix multiplication on an EREW Phase LPRAM: fanout n copies of each entry of both input matrices (using a B-ary tree), perform the n^3 multiplications, then fanin the n partial terms that sum to yield an entry in the resulting product matrix (using a B-ary tree). This runs in $O(B \log n / \log B)$ time using n^3 processors.

3.5.2 Simulation of the BPRAM and related models

The **BPRAM model of computation** [ACS89] consists of a collection of p sequential processors, each with its own private local memory, communicating with one another through a shared memory. The processors execute in lock-step, although each processor does have its own local program. There are three types of instructions.

- **Block read.** Read a contiguous block of memory words from the shared memory into private memory. If k words are read, the operation takes $l + k$ time, where l is a parameter in the model intended to capture the latency to shared memory.
- **Local operation.** Perform any RAM operation where the operands are in private memory and the result is stored in private memory. This operation takes unit time.

- **Block write.** Write the contents of a block of private memory locations into a contiguous block of shared memory. As in the block read operation, this operation takes $l + k$ time to write a block of k words.

A processor can perform at most one instruction at a time. For example, a processor issuing a block read of k words at time t can not issue another instruction until time $t + l + k$. Blocks that are accessed in parallel can not overlap. For example, if processor i reads a block b_i, then prior to the completion of this block read, no processor j may read or write a block b_j such that b_i and b_j share a common word.

3.5.2.1 BPRAM simulation

We now proceed to compare the BPRAM model with the Phase LPRAM model. By comparing the latency parameter, l, with the communication delay parameter, d, in the Phase LPRAM model, we observe that $l = 2d - 1$. In the BPRAM model, at any time step, some processor may issue a read to a block in which another processor wrote at the previous time step. In simulating the BPRAM on the Phase LPRAM, we must ensure that the write completes before this read, using a synchronization barrier. A BPRAM algorithm running in time t can be simulated by an EREW Phase LPRAM running in time $O(Bt)$ by synchronizing after each step.

The brute force method above can be improved upon. We accomplish this improvement by exploiting the fact that a block of reads takes more than l time steps to complete, and thus we can synchronize less often.

Consider slices of the BPRAM program of time l. Since a block operation can not complete in the same slice as it was issued, each slice for a processor consists of (1) waiting for a block operation to complete, (2) performing local operations once the block operation has completed, and then (3) issuing a new block operation and begin waiting again. Any of these three may be omitted, but the ones that are in a slice must be in the order specified.

BPRAM Simulation algorithm:

In this algorithm, a Phase LPRAM simulates a BPRAM algorithm with the same number of processors.

1. Each Phase LPRAM processor keeps track of BPRAM time in a local variable. For each slice of l BPRAM steps, repeat steps (2)-(4).

2. Processor i wait for any reads or writes pending from the previous slice to complete.

3. Processor i performs the local operations of BPRAM processor i, and then synchronizes.

4. If BPRAM processor i issues a read (write) of a block of k words, the Phase LPRAM processor i issues $\max(k,l)$ consecutive read (write) instructions, one per word. If $k > l$, then for each slice until all the reads (writes) in this block have been issued, processor i issues a batch of l reads (writes) and waits for them to complete.

◇

The correctness of this algorithm follows from the fact that all simulated block reads or writes in slice j complete in the Phase LPRAM before any reads or writes from slices $\geq j$ are issued. Thus the necessary synchronization is enforced.

The number of processors can be reduced to pd/B, by having each Phase LPRAM processor simulate B/d BPRAM processors.

Theorem 6 *A BPRAM algorithm running in time t using p processors can be simulated by an EREW Phase LPRAM running in $O(tB/d)$ time with pd/B processors.*

Proof. In the algorithm given above, each slice of the BPRAM is simulated in less than $2d + B + l$ time, and there are t/l slices. Having each Phase LPRAM processor simulate B/d BPRAM processors increases the time per slice to at most $(2d + l)B/d + B$, which is still $O(B)$. □

3.5.2.2 Arbitrary-BPRAM simulation

Let an **arbitrary-BPRAM** be a BPRAM that can issue requests for a non-contiguous block of memory, i.e. a block read or write instruction in the model can access an arbitrary set of k memory words in $l + k$ time. As before, sets that are accessed in parallel can not overlap.

Corollary 2 *An arbitrary-BPRAM algorithm running in time t using p processors can be simulated by an EREW Phase LPRAM running in $O(tB/d)$ time with pd/B processors.*

Proof. The proof of theorem 6 did not make use of the fact that only contiguous blocks of memory were being accessed. □

Lemma 7 *Corollary 2 is tight. There is a problem that requires $\Omega(tB/d)$ time on a CRCW Phase PRAM or Phase LPRAM, where t is the time to solve the problem on an arbitrary-BPRAM. Moreover, pd/B processors are required by the Phase PRAM or Phase LPRAM to achieve this time, where p is the number of processors used by the arbitrary-BPRAM.*

Proof. Consider the problem of determining which node in a linked list of n elements is the head of the list. An arbitrary-BPRAM can solve this problem in $O(d)$ time with n/d processors as follows. Each arbitrary-BPRAM processor reads in d nodes, sends a notification to the d successors of these nodes, then tests to see if any of its d nodes have not been notified. The lower bound for the Phase LPRAM or Phase PRAM is given in lemma 2. The lemma follows by setting $t = O(d)$ and $p = n/d$. □

3.5.2.3 Synchronous-LPRAM simulation

Unlike in the Asynchronous PRAM, a BPRAM or arbitrary-BPRAM processor can perform at most one instruction at a time. Another natural model to consider is a synchronous model in which this restriction is removed. Let a **synchronous-LPRAM** consist of a collection of p sequential processors, each with its own private local memory, communicating with one another through a shared memory. The processors execute in lock-step, although each processor does have its own local program. There are three types of instructions.

- **Global read.** Read the contents of a shared memory location shared memory into a private memory location. A global read issued at time t accesses its memory location at time $t + d$, and returns the value for use at time $t + 2d$.
- **Local operation.** Perform any RAM operation where the operands are in private memory and the result is stored in private memory. This operation takes unit time.
- **Global write.** Write the contents of a private memory location into a shared memory cell. A global write issued at time t updates its memory location at time $t + d$.

A processor can issue one instruction per time step. At each time step, at most one processor may be accessing or updating a particular memory location. This corresponds to an EREW version of the model. A CREW synchronous-LPRAM permits multiple processors to issue a read request to a location in the same time step. A CRCW synchronous-LPRAM permits multiple processors to issue a write request to a location in the same time step as well, as long as all processors attempt to write the same value. In each case, a read and a write to the same location may not be issued in the same time step. The model is related to the LPRAM model of Aggarwal and Chandra [AC88], hence its name.

A memory location in the synchronous-LPRAM can be used for communication every step of an algorithm. Consider, for example, d processors numbered 1 to d, where at each odd-numbered step i, processor i issues a write of location x, and at each even-numbered step $i + 1$, processor $i + 1$ issues a read of location x. Then a total of $d/2$ values will be exchanged in a slice of time d. The technique described in section 5.3 can be used to support this model: the routing mechanism ensures that global memory accesses at time step j stay behind those at step $j - 1$ and ahead of those at step $j + 1$.

The Phase LPRAM can be simulated on the synchronous-LPRAM with no loss.

Lemma 8 *An EREW (CREW, CRCW) Phase LPRAM algorithm running in time t using p processors can be simulated by an EREW (CREW, CRCW) synchronous-LPRAM running in t time with p processors.*

On the other hand, there are difficulties in simulating the synchronous-LPRAM on the Phase LPRAM. The main difficulty is that a memory location in the synchronous-LPRAM

can be used for communication every step of the algorithm, as described above. In the Phase LPRAM, synchronization is needed whenever there is an exchange, thus it may appear difficult to improve upon the brute force $O(Bt)$ simulation.

However, we can improve upon the brute force simulation as follows. We will divide the synchronous-LPRAM computation into slices of $2d$ steps. Each synchronous-LPRAM processor can issue up to $2d$ memory requests in a slice. Because read requests take at least $2d$ steps to complete, however, the location used or the value written by a request issued in a slice can not depend on any other request by any processor within the same slice. (The *contents* of a location read in a slice, on the other hand, does depend on the writes issued this slice and the previous slice.) For each location l, we will match each read of l in the current slice to the most recent write of l. Since, in the synchronous-LPRAM model, shared memory requests access a location exactly d steps after they are issued, it suffices to match reads with writes based on *issue* times.

Synchronous-LPRAM Simulation algorithm:

In this algorithm, an EREW Phase LPRAM simulates an EREW synchronous-LPRAM algorithm with p processors. We divide the synchronous-LPRAM computation into slices of $2d$ steps. For each slice, in the first phase, we simulate the effect of the completion of read requests and local operations, ignoring any memory requests issued this slice. Then, in the second phase, we perform all the memory requests. Since multiple reads and writes to the same location may occur in a slice, we use a multiprefix computation to match each read to its appropriate write.

1. Each Phase LPRAM processor keeps track of synchronous-LPRAM time in a local variable. For each slice of $2d$ synchronous-LPRAM steps, repeat steps (2) and (3).

2. We first simulate the completion of read requests and the execution of local operations. At the start of this iteration of the loop, due to step (3) of the previous iteration, any synchronous-LPRAM read operation that completes in step k of this slice is in slot k of a buffer for its processor. Consider each step j, $j = 1$ to $2d$, in turn. If a read completes in the synchronous-LPRAM at step j and the value read is stored in private memory location y, copy the value from buffer slot j into private location y. If a local operation is performed by the synchronous-LPRAM at step j, perform the same local operation.

3. We now consider the memory requests issued this slice. Each synchronous-LPRAM processor can issue up to $2d$ memory requests in a slice. For each location l, we need to match each read of l this slice to the most recent write of l that was issued prior to the step in which the read was issued.

This matching of reads to writes is done using a generalized multiprefix computation (see section 3.3.5) as follows. There are $n = dp$ elements, each labeled with the location to be read or written, whose "value" is the value to be written for a write operation and "$*$" for a read operation, and whose "tag" is the step number of the operation within the slice. For example, a write of x to location l in step k of the slice corresponds to an element with label l, value x, and tag k. At the start of the multiprefix computation, the value x will be written to the k^{th} leaf of tree l. Uninitialized leaves are assumed to have value "$*$". The appropriate associative operation $\oplus$ for the multiprefix computation is

$$a \oplus b = \begin{cases} b & \text{if } b \neq *, \\ a & \text{otherwise.} \end{cases}$$

Two boundary cases need to be handled during the course of the multiprefix computation. If an element corresponding to a read of location l discovers that it has the smallest tag of any element with label l, it reads memory location l to get the contents of the cell prior to the start of this slice. If an element corresponding to a write of location l discovers that it has the largest tag of any *write* element with label l, it is designated the "last writer" for label l. After the multiprefix computation has completed, the last writer for each location l (written to this slice) writes its value to memory location l. At this point, each reader in the slice has obtained the correct value and the memory locations have been updated to reflect the completion of the slice. We synchronize and continue on to the next slice.

◇

If the synchronous-LPRAM algorithm permits concurrent reading of a location in a step, i.e. the model is CREW, then in the multiprefix computation, more than one processor may be marching up the same path in a tree. If a CREW Phase LPRAM is used, the algorithm above is still correct. Likewise a CRCW synchronous-LPRAM can be simulated by a CRCW Phase LPRAM: the concurrent writing will result in more than one processor marching up the same path in a tree.

The synchronous-LPRAM Simulation algorithm yields the following theorem.

Theorem 7 *An EREW (CREW, CRCW) synchronous-LPRAM algorithm running in time t using p processors can be simulated by an EREW (CREW, CRCW) Phase LPRAM running in $O(t(B/d)\log d/\log \alpha)$ time with $\alpha p/(B/d)$ processors, for $1 \leq \alpha \leq d$.*

Proof. The multiprefix computation each slice dominates the running time for the slice. We have $n = dp$ items, $p_0 = \alpha p/(B/d)$ processors, and $r = 2d$ possible tags. Thus $n/p_0 = B/\alpha$. Since $n/p_0 < B$ when $1 \leq \alpha \leq d$, then from section 3.3.5, the time for the slice

is $O(B\lceil \log r/\log(B/(n/p_0)) \rceil)$. Plugging in for r and n/p_0 yields $O(B \log d/\log \alpha)$. The theorem follows since there are $O(t/d)$ slices in the synchronous-LPRAM program. □

From this theorem, we see that using p processors yields an $O(t(B/d)\log d/\log(B/d))$ time algorithm, while $O(t(B/d))$ time can be achieved using pd^2/B processors.

3.5.3 Other simulation results

Theorem 5 saves a $\log B$ factor off the brute force $O(B \cdot D(n))$ simulation time for computations represented as bounded fan-in circuits. Likewise, we can save a $\log B$ factor off the time (using the PRAM algorithm in [MRK88]) to evaluate a straight-line program:

Lemma 9 *Let C be an arithmetic circuit over a commutative semiring $(R, +, *, 0, 1)$ with n nodes and degree δ with values assigned to each input node from the domain R. The value for each node of the circuit can be computed in $O(B \log n \log(n\delta)/\log B)$ time on an EREW Phase LPRAM using $O(n^{2.376})$ processors.*

Proof. See [MRK88] for a precise definition of *degree*. This lemma follows from the Miller, Ramachandran, and Kaltofen PRAM algorithm for this problem, the fact that the time for matrix multiplication dominates their algorithm, and our observation that matrix multiplication can be performed in $O(B \log n/\log B)$ time on a EREW Phase LPRAM using $O(n^{2.376})$ processors. □

Lemma 10 *A CRCW PRAM algorithm running in time t using p processors can be simulated by an EREW Phase LPRAM running in time $O((B \log n/\log B)t)$ with p processors.*

This lemma follows from Eckstein's simulation of a CRCW PRAM by an EREW PRAM [Eck79] and our result for the multiprefix problem.

Chapter 4

Subset Synchronization: Algorithms and Lower Bounds

4.1 Introduction

This chapter focuses on the Asynchronous PRAM with *subset synchronization*. This variant of the Asynchronous PRAM model permits multiple disjoint sets of processors to synchronize independently and in parallel. The local program for a processor consists of a series of phases in which the processor runs independently, separated by synchronization steps involving at least one other processor. Throughout this chapter, we will assume a global read costs $2d$, a global write costs d, a local operation costs 1, and a synchronization step among a set S of processors costs $B(|S|)$. For convenience, we will refer to such Asynchronous PRAMs as **Subset LPRAMs**. There are EREW, CREW, and CRCW variants of the model, as discussed in section 2.2.2.

The processors in a set S synchronize using shared memory locations. Any two processors that agree upon a common memory location can synchronize using that location. Thus a set S can be data dependent and the processors in S need not know the identities of the other processors in S. For example, consider a graph G on p nodes with unique labels on the edges and a clique C in G. In a Subset LPRAM algorithm, a particular synchronization step may involve synchronizing all processors in C using memory locations based on the labels on the edges in C. A concrete example will be given in section 4.3 in the context of the list ranking problem.

In the results presented in this chapter, we will assume that $2 \leq d \leq B(x) \leq p$, where p is the number of processors used by the program and $2 \leq x \leq p$. Note that $B(2) \geq d$. The model charges the same cost for all global reads (global writes, pairwise synchronization steps) regardless of any physical proximity between the processor and the memory bank accessed. We will also assume that synchronization steps, like global reads and writes, can

be pipelined. In fact, we will assume that $B(d) \in \Theta(d)$. This assumption will be motivated in section 4.2.4.

Although the cost measure for the Subset LPRAM is relatively complicated, many algorithms are fairly easy to analyze. Consider, for example, the following algorithm for computing the sum of n numbers.

Summation2 program:

```
/*
inputs: The n input numbers are stored in shared memory.
outputs: The summation of the n numbers is stored to shared memory.
description: This program computes the summation using a d-ary tree, where at each
level of the tree, each active processor reads d − 1 values, computes their sum, writes the
result, and then synchronizes with its siblings.
*/
for all processors in parallel do (
    for level := 1 to log n/ log d do (
        if left-most among your siblings at the current level of the d-ary tree (
            read from shared memory the values of siblings 2, 3, ..., d;
            sum the values of all d siblings;
            write the sum to shared memory;

            synchronize with your d siblings at the next level of the tree;
        )
    )
)
```

All processors that remain active perform the same amount of work. Each active processor takes $3+(2d+d-2)+(d-1)+d$ time to complete a phase. Thus k levels of the tree are completed in $k(5d+B(d))$ time, i.e. $O(kd)$ time, since $B(d) \in \Theta(d)$. Therefore, the algorithm runs in $O(d \log n/\log d)$ time using n processors on an EREW Subset LPRAM. The processor bound can be improved by initially having each processor sum $O(d \log n/\log d)$ inputs without synchronizing. With this modification, $n \log d/(d \log n)$ processors suffice.

This chapter is organized as follows. In section 4.2, we present a *post office gossip game* for studying the inherent synchronization complexity of coordinating processors using pairwise synchronization primitives. In section 4.3, we present new algorithms and simulation results for the Asynchronous PRAM with subset synchronization.

4.2 Post Office Gossip Problems

In this section, we study complexity questions involving the use of pairwise synchronization operations to implement arbitrary synchronization patterns among processors. Upon completion of a pairwise synchronization between two processors, each processor knows that the other has arrived at this particular synchronization point. A collection of pairwise synchronization operations among the processors can be used to satisfy the synchronization requirements of a particular point in a program. We view each processor that is ready for a synchronization step as having a set of other processors with which it wishes to synchronize. Denote such a set as the goal set for a processor. In the Subset LPRAM, synchronization is among processors in a set S such that all processors in S have the same goal set S. That is, any two goal sets are either identical or disjoint. In this section, however, we will permit goal sets to overlap in arbitrary ways. We will assume, though, that if the goal set for processor i contains j then the goal set for processor j contains i.

We can represent the goal sets as a graph G on p nodes. There is an edge between node i and node j in G if processors i and j are in each other's goal sets, i.e. they each need to know that the other has arrived at the synchronization point. For example, if the objective is to have all processors synchronize with each other, then G is a complete graph.

Two observations need to be made. First, we need not perform a pairwise synchronization for each edge in G, since we can take advantage of transitivity. Consider, for example, a complete graph on four nodes, w, x, y, and z. We can satisfy the goal sets represented by this graph in two rounds. In round one, w synchronizes with x, and y with z. In round two, w synchronizes with y, and x with z. Node w does not need to explicitly synchronize with node z, since it is easily seen that w already knows that z has arrived at the synchronization point and z already knows that w has arrived. Second, we are permitted to synchronize two nodes that are not connected by an edge in G. Thus it becomes an interesting question to determine, for a given graph G, the "minimum" synchronization needed to satisfy the graph, according to some reasonable cost measure. As we will see, the minimum depends greatly not only on the cost measure but on the types of pairwise synchronization steps permitted and the information about the graph available to each processor in advance.

In what follows, we introduce a model for studying the costs of satisfying the synchronization requirements specified by a particular graph, and present results for this model. Our cost function is designed to model the following (idealized) scenario.

- No global clock.
- Two-way (nonblocking) synchronization, in which processors synchronize in pairs via the shared memory using "issue exchange" and "complete exchange" messages.
- No charge for communication or computation.

- No communication delay for synchronization messages.
- No reuse of a shared memory location.

We have chosen an idealized scenario in order to simplify the model and focus in on the "synchronization" aspects of a solution. The basic operation is a synchronization among two processors, called an **exchange**. We have divided the exchange operation into two primitives, issue exchange and complete exchange, in order to permit each processor to have multiple exchanges in progress at the same time. As we will see, this feature is quite useful in the model. Moreover, instead of being concerned with distinguishing a sender from a receiver (or a reader from a writer), we permit a symmetry between the two processors in an exchange: both perform an issue exchange, then a complete exchange to the same memory locations. To focus solely on the synchronization aspects, i.e. the extent to which processors wait for each other, we do not charge for any communication or computation. Finally, to simplify the model, we ignore issues of the communication delay of a synchronization message or the safe reuse of memory locations. Later, in section 4.2.4, we will consider the effect of communication delay.

4.2.1 The gossip model

Consider a game being played by $p \geq 2$ boys, numbered 0 to $p-1$. A large number of post office boxes that are numbered $0, 1, 2, 3, \ldots$ have been reserved at the local post office for use in the game.

The boys communicate by exchanging letters in pairs using these P.O. boxes. There are two actions: (a) mail a letter to a box and (b) request delivery from a box. Two boys exchange letters as follows. Boy b_1 mails a letter l_1 to a box x, and later requests delivery from box x. Meanwhile, boy b_2 mails a letter l_2 to box x and later requests delivery from box x. The result of this exchange is that letter l_1 is delivered to boy b_2 and letter l_2 is delivered to boy b_1. Two boys exchanging letters are called **partners**.

The rules of the game are as follows.

1. Letters of any finite length are permitted.
2. A boy must mail a letter to a box before requesting delivery from that box.
3. Each box is either not used during the game or it is the target of a letter and a delivery request by each of exactly two boys.
4. A boy who mails a letter can proceed to his next action without waiting for any action by another boy.
5. A boy who makes a request for delivery can neither mail a letter nor request delivery of another box until he receives his partner's letter.

More formally, the boys are p finite state machines. A boy's next action, including the contents of any message sent and the box used, is a function of (a) his "inputs" (described in the next section), and (b) the contents of all messages he has received.

The complexity measure for the game is based on a global clock, where the unit of time is a "day". The game takes place over a series of days. Each day, each boy who has not finished his participation in the game is either waiting for a letter from a partner, mailing a letter, or requesting delivery from a box. When a boy makes a request for delivery, there are two cases. If the partner has mailed his letter on an earlier day, then the boy receives his partner's letter the same day that his request for delivery is made. Otherwise, the boy receives his partner's letter the day after the partner sends it.

A game is completed on the day that the final boy finishes his participation in the game. The worst case time for a game is the maximum over all inputs of the number of days to complete the game.

Although the complexity measure for the game is based on a global "day", the boys in the game do not have access to any global or local clocks. A boy never knows what day it is.

A more restrictive version of the gossip game forces the boys to request delivery of a box the next action after mailing a letter to the box. In this case, we will say that the game is using **blocking exchanges.**

4.2.2 The exchange graph

Given the setup described above, the game is as follows.

Each boy has a secret. An instance of the game is defined by an **exchange graph** G on p nodes. The nodes of G are numbered 0 to $p-1$, and its edges are uniquely labeled with nonnegative integer ids. Boy i is assigned to node i.

The goal of the game is for each boy to learn the secret of each of his neighbors in G.

A particular **gossip problem** is defined by the local knowledge of the boys at the start of the game, the set of permissible exchange graphs, and whether or not blocking exchanges are required. Unless otherwise specified in the statement of the problem, each boy i knows the following at the start of the game, i.e. the following are "inputs" to finite state machine i:

- his secret,
- the id of each of the edges incident to his node, and
- the number of boys (which equals the number of nodes).

Depending on the type of exchange graphs permitted, there may be additional "default" inputs to the boys. The exchange graph can either be undirected or directed. In the latter case, another input to the boys is the direction of each of its edges.

The boys' algorithm for a gossip problem will reflect the set of exchange graphs permitted. Unless otherwise specified in the statement of a particular game, the boys know no other information about a particular problem instance.

An algorithm for a gossip problem is **valid** if, on any problem instance, no game rules are violated and, at the completion of the game, each boy knows (at least) the secrets of all his neighbors.

The difficulty of the game is that each boy must determine the order with which to mail letters and request deliveries starting with only local information.

Our model differs from the gossip model in [EM89] and earlier work (e.g. [BS72]) in at least three respects. First, the game is asynchronous, i.e. there is no global clock that can be used to signal the end of a round. Second, we study arbitrary exchange graphs, whereas earlier work studied only (a) exchanging all secrets and (b) broadcasting a single secret. Third, we consider communication delay and pipelining in a variant of our model.

4.2.3 Upper and lower bounds for gossip problems

In this section, we present upper and lower bounds for gossip problems. We will examine the effect of various restrictions on the local information of the boys at the start of the game, the exchange graphs permitted, and the type of exchanges permitted. For simplicity in describing the algorithms for gossip problems, we will assume that an exchange graph has no self-loops. Since a boy can always test all his edges for self-loops in zero days, the results presented in this section can easily be extended to the case where the exchange graph has self-loops.

4.2.3.1 Preliminary results

Lemma 11 *Any gossip problem (as defined in section 4.2.2) can be solved in* $2\lceil \log p \rceil$ *days, with blocking exchanges.*

Consider the following algorithm for boy 0 to learn all the secrets.

Fan-in program:

/*
inputs: Let s_i be the secret for boy i.
output: All secrets are collected into boy 0's local variable "my_secrets".
description: The secrets are collected using a binary tree of p leaves.
*/

for all boys i **in parallel do** ⟨
(1) my_secrets := $\{s_i\}$; /* secrets known by boy i */
(2) **for** $j := 0$ **to** $\lceil \log p \rceil - 1$ **do** ⟨
(3) **if** i is divisible by 2^{j+1} ⟨ /* left sibling */

```
(4)          mail to box i + 2^j;        /* use the box of the right sibling */
(5)          deliver box i + 2^j returning siblings_secrets;
(6)          my_secrets := my_secrets ∪ siblings_secrets;
         )
(7)      else if i is divisible by 2^j (     /* right sibling */
(8)          mail my_secrets to box i;
(9)          deliver box i;
         )
     )
)
```

A symmetric, "fan-out" algorithm has all boys learn the secret(s) of boy 0. Both these algorithms use (only) blocking exchanges.

Combining the fan-in and fan-out algorithms yields an algorithm in which all boys learn all secrets in $4\lceil \log p \rceil$ days. An improved algorithm, which we call the **complete-exchange** algorithm, exchanges secrets using a butterfly graph of p rows and $\lceil \log p \rceil$ stages. (Figure 3.2 contains a butterfly graph on 32 nodes.) For simplicity, assume that p is a power of two. Each *stage* of the butterfly consists of a set of $p/2$ complete 2-by-2 bipartite graphs. In the complete-exchange algorithm, each 2-by-2 corresponds to an exchange between partners. At each exchange, each boy sends all the secrets he knows to his partner. Consider two boys, j and k, $j < k$. In round i of the algorithm (stage i of the butterfly), $1 \leq i \leq \log p$, these two boys are partners if $k = j + p/2^i$. By performing these stages of exchanges in order, all boys can learn all secrets in $2\lceil \log p \rceil$ days, and lemma 11 follows.

Theorem 8 *Consider a set of k boys each of whom has a secret. Suppose each boy knows the ids of all the others. Then all the boys in the set can each learn all the secrets in $2\lceil \log k \rceil$ days using blocking exchanges. Moreover, $2\lceil \log k \rceil$ days are required (even with nonblocking exchanges) for one boy to learn the secrets of k boys, k boys to learn the secret of one boy, or k boys each to learn the secrets of all k boys.*

Proof. The upper bound uses the complete-exchange algorithm. For the lower bounds, it takes two days (one for mailing, one for requesting delivery) to pass a secret from one boy to another. Thus, even with nonblocking exchanges, the number of boys that know a secret (or the number of secrets that are known by a boy) can at most double every two days. □

Corollary 3 *The gossip problem requires $\Omega(\log \delta)$ days in the worst case, where δ is the maximum degree of a node in the exchange graph.*

Lemma 12 *Let δ, the maximum degree of a node in the exchange graph, be an input to the boys. Then the gossip problem can be solved in $O(\min(\delta, \log p))$ days.*

Proof. If $\delta \geq \log p$, use the complete-exchange algorithm. Otherwise, have each boy mail his secret to box i for each incident edge i (a total of δ letters are sent), and then request delivery of those boxes. All letters will have been mailed by day δ, thus all delivery requests will have completed by day 2δ. Each box corresponding to an edge is used by exactly two boys for two letters, so the algorithm is valid. □

Lemma 13 *Consider a gossip problem where the boys do not have ids. This gossip problem can be solved in $O(\delta)$ days, where δ is the maximum degree of a node in G.*

Proof. As in the previous lemma, have each boy mail his secret to box i for each incident edge i, and then request delivery of those boxes. □

In contrast, theorem 9 in the next section shows that if the boys are restricted to blocking exchanges, $\Omega(p)$ days are required even for a restricted set of exchange graphs.

4.2.3.2 Directed cycles

Theorem 9 *Consider a gossip problem P where the boys do not have ids, only blocking exchanges are permitted, and an exchange graph must be a collection of directed cycles, i.e. each node in the graph has indegree one and outdegree one. Then P has a matching upper and lower bound of $2p$ days.*

Proof. The upper bound can be achieved (without deadlock) by having each boy first perform an exchange for his edge with the smaller label. For example, a boy with secret s and with edges labeled i and j, where $i < j$, (a) mails a letter containing s to box i, (b) requests delivery of box i, (c) mails a letter containing s to box j, and then (d) requests delivery of box j. In the worst case (a cycle of all the edges in increasing order), this algorithm takes $2p$ days.

Now we turn to the lower bound. The first two actions of each boy are to mail a letter to a box and then to request delivery of the box. In this anonymous game, we can assume without loss of generality that all boys have the same function f for deciding what box to use first. This function can depend only on e_i and e_o, where e_i is the incoming edge and e_o is the outgoing edge, and its output must be either a function $g(e_i)$ or a function $h(e_o)$. The latter is true since a boy b can never be his own partner, i.e. he must wait for another boy to mail a letter to the same box, and no other boy may have the same pair of adjacent edges or any other fact in common with b. In other words, a boy's first exchange must be either with his predecessor (using a box $g(e_i)$) or with his successor (using a box $h(e_o)$), since he can not locate any other boy. Furthermore, since adjacent boys must use the same box to communicate, we can assume w.l.o.g. that $g(e_i) = e_i$ and $h(e_o) = e_o$.

Suppose $f(a,b) = a$ and $f(b,a) = b$. Then if the two edges a and b form a two-cycle in the exchange graph, the two boys in the cycle will deadlock, since each will be stuck waiting for each other at different boxes. By symmetry, $f(a,b) = b$ and $f(b,a) = a$ also results in deadlock. Thus for each pair a and b, either $f(a,b) = f(b,a) = a$ or $f(a,b) = f(b,a) = b$.

Consider a graph A based on f where the edge ids are the nodes and there is a directed edge from node a to node b if $f(a,b) = a$. The above argument shows that A is a tournament. Furthermore, A must be acyclic, since any cycle in A leads to a deadlock when the exchange graph contains a cycle of those same edges. It is easy to see by induction that an acyclic tournament must be a totally ordered chain of nodes.

Given any fixed f corresponding to a totally ordered chain, consider the case where the exchange graph is a single cycle of its edges in order of the chain. Then only one exchange can complete on the first two days, and the exchanges will complete, in order, at a rate of one per every two days. Thus the gossip problem requires $2p$ days. □

Lemma 14 *Consider a gossip problem where an exchange graph must be a collection of directed cycles, only blocking exchanges are permitted, and the edge ids are numbered 0 to* $p-1$. *Then this gossip problem can be done in* $O(1)$ *days.*

Proof. (Shamir [Sha89]) The cycle sharing algorithm below runs in 12 days using blocking exchanges. □

The **successor id** of an arc is the id of the boy into whom the arc is directed. The **predecessor id** of an arc is the id of the boy out of whom the arc is directed. Consider a directed graph where incoming arcs to a node are numbered with consecutive nonnegative integers starting with 0. Then the (successor) **sibling number** of an arc is i if the arc is number i among those arcs directed into the same node.

Remark. With exchange graphs of indegree one, the following are equivalent, i.e. any gossip problem under one scenario can be solved under the other scenario in the same number of days.

- The edge ids are numbered 0 to $p-1$, not necessarily corresponding to the ids of the boys at either endpoint of an edge.
- Each boy knows the ids of his successors in the exchange graph.

If the first is given, each boy can assume the id of his incoming arc. If the second is given, each arc can be assumed to be labeled with its successor id. This remark also holds for graphs of outdegree one, and for directed trees. In the latter case, there are only $p-1$ edges. The equivalence holds if the root of the tree can determine the unused edge label, e.g. if the edges are numbered 0 to $p-2$, the root takes label $p-1$.

Cycle Sharing algorithm:

Label each boy by the id of his incoming edge (thus the boys are labeled from 0 to $p-1$). Two boys i and j (by this new labeling) are "brothers" in this algorithm if and only if $i = j \bmod (p/2)$. A boy with label less than $p/2$ is "low", otherwise he is "high". Note that brothers will typically not be adjacent in the graph. There are four steps to the algorithm.

1. Each boy exchanges his secret and the id of his outgoing edge with his brother.

2. Each high boy h sends his secret to box $h+p$ and then requests delivery of this box.

   In the meantime, each low boy sends up to two letters as follows. (a) If his outgoing edge i is at least $p/2$, then he sends his secret to box $i+p$ and then requests delivery of this box. (b) If his brother's outgoing edge j is at least $p/2$, then he sends his brother's secret to box $j+p$ and then requests delivery of this box.

   In this way, the boys have completed exchanges for all edges with ids at least $p/2$, i.e. into high boys.

3. Each low boy l sends his secret to box $l+p$ and then requests delivery of this box.

   In the meantime, each high boy sends up to two letters. (a) If his outgoing edge i is less than $p/2$, then he sends his secret to box $i+p$ and then requests delivery of this box. (b) If his brother's outgoing edge j is less than $p/2$, then he sends his brother's secret to box $j+p$ and then requests delivery of this box.

   In this way, the boys have completed exchanges for all edges with ids less than $p/2$, i.e. into low boys.

4. At this point, a boy may have the secret needed by his brother. If a boy received a secret on behalf of his brother (step 2b or 3b), then he sends this secret to his brother, using an exchange with his brother.

◇

For clarity, and to help justify the correctness claims, we next give a pseudo-code program for this gossip problem.

Cycle Sharing program:

/*

inputs: Let "my_secret", "my_name", and "my_successor" be local variables for each boy containing the boy's secret, incoming edge id, and outgoing edge id, respectively.

outputs: At the end of the algorithm, the local variables "predecessors_secret" and "successors_secret" for each boy will contain the boy's predecessor's secret and successor's secret, respectively.

```
description: This program implements the cycle sharing algorithm described above. The
numbers beside program statements correspond to steps in the description of the algorithm.
*/
for all boys in parallel do (
    case my_name < p/2 (                    /* low boy */
        brothers_name := my_name+p/2;
        for_brother := empty_msg;
(1)     mail my_secret, my_successor to box my_name;
        deliver box my_name returning brothers_secret, brothers_successor;
(2a)    if my_successor ≥ p/2 (
            mail my_secret to box my_successor+p;
            deliver box my_successor+p returning successors_secret;
        )
(2b)    if brothers_successor ≥ p/2 (
            mail brothers_secret to box brothers_successor+p;
            deliver box brothers_successor+p returning brothers_successors_secret;
            for_brother := brothers_successors_secret;
        )
(3)     mail my_secret to box my_name+p;
        deliver box my_name+p returning predecessors_secret;
(4)     mail for_brother to box brothers_name;
        deliver box brothers_name returning from_brother;
        if my_successor ≥ p/2
            successors_secret := from_brother;
    )

    case my_name ≥ p/2 (                    /* high boy */
        brothers_name := my_name−p/2;
        for_brother := empty_msg;
(1)     mail my_secret, my_successor to box brothers_name;
        deliver box brothers_name returning brothers_secret, brothers_successor;
(2)     mail my_secret to box my_name+p;
        deliver box my_name+p returning predecessors_secret;
(3a)    if my_successor < p/2 (
            mail my_secret to box my_successor+p;
            deliver box my_successor+p returning successors_secret;
        )
(3b)    if brothers_successor < p/2 (
```

```
            mail brothers_secret to box brothers_successor+p;
            deliver box brothers_successor+p returning brothers_successors_secret;
            for_brother := brothers_successors_secret;
        )
(4)     mail for_brother to box my_name;
        deliver box my_name returning from_brother;
        if my_successor ≥ p/2
            successors_secret := from_brother;
    )
)
```

The proof of correctness is left to the interested reader. The analysis of the cycle sharing algorithm is as follows. All boys complete step 1 on day two. A low boy completes step 2a on day four and step 2b on day six in the worst case. Thus a high boy completes step 2 on day six in the worst case. A high boy completes step 3a on day eight and step 3b on day ten in the worst case. Thus a low boy completes step 3 on day ten in the worst case. All boys complete step 4 on day twelve in the worst case. Thus the cycle sharing algorithm runs in 12 days as claimed.

4.2.3.3 Directed trees

Lemma 14 can be generalized as follows.

Theorem 10 *Consider a gossip problem where an exchange graph must be a tree directed towards its leaves, only blocking exchanges are permitted, and the edge ids are numbered 0 to $p-2$. Then this gossip problem can be done in $\Theta(\log \delta)$ days, where δ is the maximum outdegree of a node in the exchange graph. Likewise, the gossip problem corresponding to the symmetric case of a tree directed towards its root can be done in $\Theta(\log \delta)$ days.*

Proof. The tree sharing algorithm below solves the problem for trees directed towards their leaves in $20 + 6 \log \delta$ days. A symmetric algorithm solves the case where the tree is directed towards its root. These results match the $\Omega(\log \delta)$ lower bound (corollary 3). □

Tree Sharing algorithm:

Label the boy at the root of the tree $p-1$, and label each other boy by the id of his incoming edge (thus the boys are labeled from 0 to $p-1$). Two boys i and j (by this new labeling) are "brothers" if and only if $i = j \bmod (p/2)$. A boy with label less than $p/2$ is "low", otherwise he is "high". In the first half of the algorithm, we will broadcast a boy's secret to all his successors. Figure 4.1 gives an example and shows the intermediate steps of this

boy "0"	{10}	{10},{}	$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	{0,4}	{0}
boy "1"	{}	{},{}	$\longrightarrow$	{1}			
boy "2"	{1}	{},{}	$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	{2,6}	{2}
boy "3"	{8,9,12,14}	{8,9,12,14},{}	$\longrightarrow$	{3}			
boy "4"	{}	{},{11,13}	$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	{4}
boy "5"	{7}	{},{}	$\longrightarrow$	$\longrightarrow$	{5}		
boy "6"	{}	{},{}	$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	{6}
boy "7"	{}	{},{}	$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	{7}	
boy "8"	{}	{},{}	{8,9,12,14}	{8,9}	{8}		
boy "9"	{2,6}	{2,6},{}	$\longrightarrow$	$\longrightarrow$	{9}		
boy "10"	{}	{},{1}	{10}				
boy "11"	{}	{},{}	{11,13}	{11}			
boy "12"	{0,4,11,13}	{0,4},{}	$\longrightarrow$	{12,14}	{12}		
boy "13"	{5}	{5},{7}	$\longrightarrow$	{13}			
boy "14"	{}	{},{}	$\longrightarrow$	$\longrightarrow$	{14}		
boy "15"	{3}	{3},{}	(root)				

Figure 4.1: Applying the tree sharing algorithm to an example exchange graph. The intermediate steps of the tree sharing algorithm are shown for an example where $p = 16$ and $\delta = 4$. Only the first half of the algorithm is shown, in which each boy broadcasts his secret to all his successors (the secrets are not shown). The first column contains the boys listed according to their labels. The first eight boys are *low*, the second eight are *high*. The second column is the original list of outgoing edges for each boy. The third column contains, for each low boy, the outgoing high edges for the boy and his brother, and for each high boy, the outgoing low edges for the boy and his brother. Columns 4-6 show the steps in distributing each high edge h to (high) boy h. Columns 5-8 show the steps in distributing each low edge l to (low) boy l. Low edges are sent from a high boy to a low boy as soon as the high boy has finished its distribution of high edges.

first half of the algorithm. In the second half, we will return to the boy the secrets of all his successors.

1. Each low (high) boy sends his secret and the ids of his outgoing low (high) edges to his high (low) brother.

2. Each high boy h (except the root boy) sends an empty message to box $h+p$ and then requests delivery of this box. This box will contain his predecessor's secret and a set S of one or more high edges that share the same predecessor ($h \in S$).

   In the meantime, each low boy sends up to 2 letters. (a) If he has at least one outgoing high edge, he sends his secret and the ids of his outgoing high edges to box $i+p$, where i is, say, the smallest of the outgoing high edges, and then he requests delivery of this box (receiving an empty message). (b) If his brother has at least one outgoing high edge, he sends his brother's secret and the ids of his brother's outgoing high edges

to box $j + p$, where j is, say, the smallest of the outgoing high edges, and then he requests delivery of this box (receiving an empty message).

3. Each high boy continues for up to $\log d$ rounds as follows. If the set S received by a high boy h contains more than one member, send his predecessor's secret and half of the set S (say, the larger half) to box $i + p$, where i is the smallest member of S that was sent. Then, request delivery of this box and repeat this step until but one member of S remains, namely h.

   In this way, each high boy receives his predecessor's secret.

4. Step 2 is repeated with the roles of the high and the low boys reversed.

5. Step 3 is repeated with the roles of the high and the low boys reversed.

   In this way, each low boy receives his predecessor's secret.

6. In the second half of the algorithm, each boy will receive all his successor's secrets.

   We first unwind step 3. Consider a high boy h (not the root boy) and let $h_1 + p, h_2 + p, \ldots, h_k + p$ be the boxes he used in step 3, where k is the number of rounds he sent messages in step 3. If $k = 0$, i.e. the boy received a set S of size 1, then boy h sends his secret to box $h + 2p$ and then requests delivery of this box. Else boy h does the following for k rounds. In round i, boy h sends an empty message to box $h_i + 2p$, requests delivery of this box, and adds the set of secrets received to those secrets received in earlier rounds. At the end of k rounds, boy h sends all the secrets he has received to box $h + 2p$ and then requests delivery of this box.

7. To unwind step 2, each low boy receives up to 2 messages as follows. (a) If he has at least one outgoing high edge, he sends an empty message to box $i + 2p$, where i is the smallest of the outgoing high edges, and then he requests delivery of this box (receiving the set of secrets for his high successors). (b) If his brother has at least one outgoing high edge, he sends an empty message to box $j + 2p$, where j is the smallest of the outgoing high edges, and then he requests delivery of this box (receiving the set of secrets for his brother's high successors).

8. Step 6 is repeated with the roles of the high and the low boys reversed, in order to unwind step 5.

9. Step 7 is repeated with the roles of the high and the low boys reversed, in order to unwind step 4.

   In this way, each high boy receives the set of secrets for his low successors and his brother's low successors.

10. At this point, a boy may have secrets needed by his brother. If a boy received secrets on behalf of his brother (step 7b or 9b), then he sends these secrets to his brother, using an exchange with his brother.

◇

The worst case analysis of the tree sharing algorithm is as follows. All boys complete step 1 on day two. A low boy completes step 2a on day four and step 2b on day six (in the worst case). If a high boy receives a set S of δ members in step 2, then he performs $\log \delta$ rounds in step 3, completing the first on day eight and the last on day $6 + 2\log\delta$. A high boy (in this worst case) completes step 4b on day $10 + 2\log\delta$ and begins the second half of the algorithm. Meanwhile, if a low boy receives a set S of δ members in step 4, then he performs $\log \delta$ rounds in step 5, completing the first on day $12 + 2\log\delta$ and the last on day $10 + 4\log\delta$.

Unwinding step 3 takes $2\log\delta$ days. Thus both the high boys and the low boys complete step 7 on day $14 + 4\log\delta$. Then unwinding step 5 takes $2\log\delta$ days, so all boys complete step 9 on day $18 + 6\log\delta$. Step 10 takes two days. Thus the tree sharing algorithm runs in $20 + 6\log\delta$ days as claimed.

The tree sharing algorithm can be further generalized as follows.

Corollary 4 *Consider a gossip problem where an exchange graph must be a directed graph of maximum indegree (or outdegree) one, only blocking exchanges are permitted, and the edge ids are numbered 0 to $m-1$, where each boy knows the number of edges m. Then this gossip problem can be done in $\Theta(\log\delta)$ days, where δ is the maximum outdegree (or indegree) of a node in the exchange graph.*

Proof. The problem in extending the tree sharing algorithm to this gossip problem is that we need a way to divide the boys into two groups, low and high, such that (a) a predecessor knows whether his successor is low or high and (b) low boys can be paired with high boys as brothers. In the tree sharing algorithm, we used the id of the incoming edge as the label of a boy and $p-1$ as the label of the root boy. If there is more than one boy of indegree zero, however, this simple technique no longer works.

The solution in the case where the number of edges is known by the boys, is as follows. Consider the set of m boys of indegree one and label each one by the id of its incoming edge. Let the boys labeled less than $m/2$ be low, the rest high. (If m is odd, it poses no problems to the algorithm to have two low boys share a high brother.) Root boys fall into a third category. A root boy does not need the help of a brother since it has only successors.

The algorithm proceeds as in the tree sharing algorithm, except that a root boy does not participate in any exchange between brothers and a root boy may send his secret in both steps 2 and 4. To elaborate, in step 2 (step 4), if a root boy has at least one outgoing high (low) edge, he sends his secret and the ids of his outgoing high (low) edges to box

$i+p$, where i is the smallest of the outgoing high (low) edges, and then he requests delivery of this box. Likewise, in the second half of the algorithm, a root boy may receive a set of secrets in both steps 7 and 9. □

Corollary 5 *Consider a gossip problem where an exchange graph must be a directed graph of maximum indegree (or outdegree) one, only blocking exchanges are permitted, and each arc is labeled with its successor (or predecessor) id. Then this gossip problem can be done in* $\Theta(\log \delta)$ *days, where* δ *is the maximum outdegree (or indegree) of a node in the exchange graph.*

If each arc is labeled with its successor id, then we can divide the boys into high and low by their ids and each predecessor will know whether his successor is low or high.

4.2.3.4 General graphs

Theorem 11 *Consider a gossip problem where an exchange graph is a fixed directed graph and only blocking exchanges are permitted. Then this gossip problem can be done in* $O(\min(\delta, \log p))$ *days, where* δ *is the maximum indegree or outdegree of a node in the exchange graph.*

Proof. In this off-line problem, the exchange graph is known in advance, but not the secrets. If $\delta \geq \log p$, use the complete-exchange algorithm. Otherwise, decompose G into δ subgraphs, not necessarily edge disjoint, whose union is G such that each of the subgraphs is indegree and outdegree one. Each of these graphs can be edge colored with three colors, and hence solved in 6 days using blocking exchanges (two days per color). □

Theorem 12 *Consider a gossip problem where an exchange graph is a directed graph, only blocking exchanges are permitted, and each arc is labeled with the pair* (b, s)*, where* b *is its successor id and* s *is its sibling number. Then this gossip problem can be done in* $O(\delta_i \log \delta_o)$ *days, where* δ_i *(*δ_o*) is the maximum indegree (outdegree) of a node in the exchange graph.*

Proof. The algorithm consists of δ_i rounds. At round i, consider the subgraph G_i induced by all arcs with sibling number i. Each G_i is a directed graph of maximum indegree one. Thus, by corollary 5, each round can be performed in $O(\log \delta_o)$ days. Note that each round needs to use its own set of boxes, but this can be accomplished by using box numbers that are offset an amount based on the round number. □

By symmetry, the lemma holds when the roles of the indegree and outdegree are reversed. In either case, for graphs of high degree, the complete-exchange algorithm (lemma 11), which solves this problem in $O(\log p)$, can be used instead.

4.2.4 Adding communication delay to the gossip model

In the post office gossip model, a boy can send his secret to another boy in two days. Now consider the case where a letter takes d days to reach the post office once mailed and $2d$ days to be delivered once requested. If each boy can have at most one letter or delivery request in transit to the post office at a time, we simply multiply the results of the previous section by d to get results for this new model. If, however, each boy can have multiple letters and delivery requests in transit on the same day, then these new results can be improved upon. We refer to this latter post office gossip model as the **pipelined gossip model.** These modifications are intended to capture the communication delay and pipelining of the Asynchronous PRAM model. Note that, in order to pipeline, pairwise exchanges must be nonblocking.

The pipelined gossip model will be used in section 4.3 to prove results about the Subset LPRAM. In what follows, we first present two lemmas concerning the complete exchange of secrets among a set of boys in the pipelined gossip model. Then we present two results on solving pipelined gossip problems on arbitrary graphs.

Lemma 15 *Consider a set of k boys, $2 \leq k \leq d$, each of whom has a secret. Suppose each boy knows the ids of all the others. Then all the boys in the set can each learn the secrets of all k boys in the pipelined gossip model in $\Theta(d)$ days.*

Proof. Consider the boys to be numbered 0 to $k-1$. Each boy i sends his secret to box $i+kj$, for $0 \leq j \leq k-1$, $j \neq i$. Each boy i then requests delivery of box $ki+j$, for $0 \leq j \leq k-1$, $j \neq i$. This completes in $O(d)$ days since both the sends and the delivery requests can be pipelined.

A lower bound of $2d$ follows from the fact that it takes $2d$ days to pass a secret from one boy to another. □

This result motivates our assumption in this chapter that $B(d) \in \Theta(d)$ in the Subset LPRAM.

Lemma 16 *Consider a set of k boys, $k \geq d$, each of whom has a secret. Suppose each boy knows the ids of all the others. Then all the boys in the set can each learn the secrets of all k boys in the pipelined gossip model in $\Theta(d \log k / \log d)$ days.*

Proof. We use a variant of the complete-exchange algorithm (described in section 4.2.3) in which the exchanges are performed according to a butterfly graph of base d. For simplicity, assume that k is a power of d. Each *stage* of the base d butterfly consists of a set of p/d complete d-by-d bipartite graphs. For each of the $\log k / \log d$ stages, each d-by-d corresponds to a complete exchange between a set of d boys. By the previous lemma, this can be done in $O(d)$ days, so all boys can learn all secrets in $O(d \log k / \log d)$ days.

For the lower bound, consider the problem of k boys learning the secret of one boy. Initially one boy, i, knows the secret. After $2d-1$ days, boy i will have mailed at most $2d-1$ letters and no other boy will have received the secret yet, and hence no other boys will have mailed a letter containing the secret yet. Assume that all letters in transit magically arrive the next day (this can only help the algorithm). Then there are $2d$ boys with the secret and no letters with the secret in transit. After $4d-1$ days, at most $2d$ boys will have mailed a letter containing the secret, to a total of at most $2d(2d-1)$ boys. Assume that all letters in transit magically arrive the next day. Then there are at most $4d^2$ boys with the secret and no letters with the secret in transit. In general, after $t(2d)-1$ days, there are at most $(2d)^t$ boys with the secret. Thus, in order for k boys to know the secret, we need $(2d)^t \geq k$, i.e. $t \geq \log k/\log(2d)$. Therefore $\Omega(d \log k/\log d)$ days are required. □

Corollary 6 *The gossip problem requires $\Omega(d \log \delta/\log d)$ days in the worst case, where δ is the maximum degree of a node in the exchange graph.*

Lemma 17 *Let δ, the maximum degree of a node in the exchange graph, be an input to the boys. Then the gossip problem can be solved in $O(\min(d+\delta, d \log p/\log d))$ days.*

Proof. If $\delta \geq d \log p/\log d$, apply the approach used in lemma 16. Otherwise, have each boy mail his secret to box i for each incident edge i, and then request delivery of those boxes. All letters will have been mailed by day δ, reached their boxes by day $\delta + d$, and delivered by day $\delta + 2d$ (using pipelining). Each box corresponding to an edge is used by exactly two boys for two letters, so the algorithm is valid. □

4.3 Algorithms and Simulation Results

In this section, we present new algorithms and simulation results for the Subset LPRAM.

4.3.1 Subset LPRAM vs. Phase LPRAM

We begin with a comparison between the Subset LPRAM and the Phase PRAM or Phase LPRAM.

Theorem 13 *An EREW (CREW, CRCW) Subset LPRAM algorithm running in time t using p processors can be simulated by an EREW (CREW, CRCW) Phase PRAM or Phase LPRAM running in $O(tB(p)/B(2))$ time with $pB(2)/B(p)$ processors.*

Proof. Let $q = pB(2)/B(p)$. We assume that $B(p) \in \Theta(B(q))$. Each Phase LPRAM processor simulates $B(p)/B(2)$ Subset LPRAM processors as follows. Consider slices of the Subset LPRAM computation of $B(2)$ time steps each slice. Each processor can participate in at most one synchronization step during a slice. For $1 \leq i \leq t/B(2)$, let $C(i,j)$ be the set

of instructions issued by processor j in slice i prior to its synchronization step (if any). Let $D(i,j)$ be the set of instructions issued by processor j in slice i after its synchronization step (if any). To simulate slice i, each Phase LPRAM processor first performs all the instructions in $C(i,k)$, where k is one of its $B(p)/B(2)$ simulated processors, then synchronizes with all the other processors. Second, it performs all the instructions in $D(i,k)$, where k is one of its $B(p)/B(2)$ simulated processors, and continues to the next slice.

For each slice, the first half of the simulation can be done in $O(d + B(2)(B(p)/B(2)) + B(q))$ time and the second half can be done in $O(d + B(2)(B(p)/B(2)))$ time. Thus each slice takes time x, $x \in O(d + B(p) + B(q)) \in O(B(p))$. Since there are $t/B(2)$ slices, the total time is $O(tB(p)/B(2))$. □

Corollary 7 *An EREW (CREW, CRCW) Subset LPRAM algorithm running in time t using p processors can be simulated by an EREW (CREW, CRCW) Phase PRAM or Phase LPRAM running in $O(t \log p / \log d)$ time, with $p \log d / \log p$ processors.*

Proof. This follows since $B(2) \in \Omega(d)$ and $B(p) \in O(d \log p / \log d)$ by lemmas 15 and 16. □

4.3.2 Algorithms for important primitive operations

In this section, we present Subset LPRAM algorithms for many of the important primitive operations discussed in chapter 3. We will refer to the following lower bound.

Lemma 18 *Let f be an n input, m output associative function such that at least one of the outputs of f depends on all the inputs. Then $\Omega(d \log n / \log d)$ time is required for a CRCW Subset LPRAM to compute f, regardless of the number of processors.*

Proof. By an argument similar to that used in theorem 2, $\Omega(d \log n / \log d)$ time is needed to fanin the n inputs to an output that depends on all the inputs. □

The algorithms presented here are based on the Phase LPRAM algorithms for these problems. Many of the Phase LPRAM algorithms presented in chapter 3 consist of a series of stages in which each processor performs $O(B)$ steps, separated by synchronization barriers. The barrier synchronization in these Phase LPRAM algorithms can safely be replaced in the Subset LPRAM by synchronization steps involving disjoint subsets of size $f(B) < p$. For example, in the $O(B \log n / \log B)$ time, $n \log B / B$ processor Phase LPRAM algorithm for the FFT problem, subsets are size $f(B) = B / \log B$. The set size $f(B)$ in the Phase LPRAM algorithms is selected to ensure that each processor performs $O(B)$ steps each stage. In designing Subset LPRAM algorithms for these problems, we aim to have each processor perform $O(d)$ steps per stage and synchronize in sets of size $f(d) \leq d$. An example is the summation algorithm for the Subset LPRAM presented in section 4.1, in which we use a d-ary tree and synchronize in sets of size d. In contrast, the summation

algorithm for the Phase LPRAM in section 3.2 uses a B-ary tree and synchronizes in sets of size p. By lemma 18, each of these problems has a matching $\Omega(d \log n / \log d)$ lower bound.

- The prefix problem can be solved in $O(d \log n / \log d)$ time on an EREW Subset LPRAM with $n \log d / d \log n$ processors.
- The FFT and bitonic merge problems can be solved in $O(d \log n / \log d)$ time on an EREW Subset LPRAM with n processors.
- Matrix multiplication can be solved in $O(d \log n / \log d)$ time on an EREW Subset LPRAM with n^3 processors.

We omit any further details.

Now consider the list ranking problem. In the list ranking problem, we assume the head and tail of the list are marked. During the course of the algorithm, the processors perform a series of pointer jumping steps. To avoid concurrent reading of the tail element, we keep track of which elements currently point to the tail element so that we do not pointer jump to the tail element. We use one processor per element. For each of $\log n$ rounds, each processor performs a pointer jumping step for its element. This takes $O(d)$ time. Then each processor synchronizes with its new successor. We can view each round as separated by a synchronization point whose exchange graph has outdegree and indegree at most one. Thus by corollary 5, this synchronization can be accomplished in $O(1)$ gossip game days.

In applying gossip game results to the Subset LPRAM, we must first ensure that the synchronization can be done within the rules of the more restrictive Subset LPRAM model (see the discussion at the beginning of section 4.2). In this case, there is no problem since corollary 5 is a result for *blocking* exchanges. Thus each day consists of a collection of synchronizations among sets of size two. Second, in the Subset LPRAM, we must account for communication and computation costs not reflected in the gossip game results. For the gossip game algorithm used to prove corollary 5, each gossip game day takes at most $2d$ time on the Subset LPRAM. Therefore, we have the following result.

Lemma 19 *The list ranking problem can be solved in $O(d \log n)$ time on an EREW Subset LPRAM with n processors.*

It is interesting to note that if the Subset LPRAM model were modified to permit the less restrictive synchronization steps of the gossip model, then the list ranking problem can be solved in $O(d \log n)$ time with only n/d processors. The algorithm is as follows. Each processor is responsible for d elements in the list. For each of $\log n$ rounds, each processor performs a pointer jumping step for each of its elements. This takes $O(d)$ time, since the global accesses can be pipelined. Then each processor synchronizes with d others, where the set of d is data dependent. The exchange graph at each synchronization point has maximum

indegree and outdegree d. By lemma 17, this synchronization can be accomplished in this less restrictive model in $O(d)$ time. Thus the total time is $O(d \log n)$ as claimed. Similarly, the FFT and bitonic merge problems can be solved using fewer processors in this less restrictive model. Both problems can be solved in $O(d \log n / \log d)$ time, with the FFT problem using $n \log d / d$ processors and the bitonic merge problem using n/d processors. We give a very brief sketch of the algorithms used. As in the Phase LPRAM algorithms for these problems described in section 3.3, each processor is assigned multiple nodes in a stage. For each stage, each processor synchronizes with the d processors from the previous stage, reads in the values from shared memory, computes the outputs, and then writes the outputs to shared memory. Since the exchange graph at each synchronization point has maximum indegree and outdegree d, each processor can complete its synchronization for a stage in $O(d)$ time (lemma 17). Thus each stage can be performed in $O(d)$ time, so the total time for the two problems is $O(d \log n / \log d)$ as claimed.

4.3.3 Comparisons with synchronous models

In this section, we compare the Subset LPRAM to existing synchronous models of parallel computation. We will use the following lower bound.

Lemma 20 *Consider the problem of determining which node in a linked list of n elements is the head of the list. This problem requires $\Omega(d \log p_0 / \log d)$ time on a CRCW Subset LPRAM, where $p_0 = n \log d / d \log p_0$ is the number of processors needed to achieve this time.*

Proof. The argument presented in section 2.1 shows that a synchronization step among all the processors is required. Thus the Subset LPRAM requires $\Omega(n/p + d \log p / \log d)$ time to solve this problem with p processors. The time is minimized when $p = n/(d \log p / \log d)$. □

Note that more processors can be available, but only p_0 processors should be used to achieve the given time bound.

The following results follow directly from the simulation results presented in chapter 3. Definitions of the models are given in that chapter. The upper bounds presented below are trivially true since the Subset LPRAM can simulate the Phase LPRAM with no loss (in the results below, we have replaced $B(p_0)$ by $d \log p_0 / \log d$ in accordance with lemma 16). The lower bounds follow from lemma 20. In many cases, the lower bound matches the upper bound. This list of results is presented here for completeness.

- An EREW (CREW, common-CRCW) PRAM algorithm running in time t using p processors can be simulated by an EREW (CREW, CRCW) Subset LPRAM running in $\Theta(td \log p_0 / \log d)$ time with $p_0 = p \log d / (d \log p_0)$ processors.

- A BPRAM algorithm running in time t using p processors can be simulated by an EREW Subset LPRAM running in $O(t \log p_0 / \log d)$ time with $p_0 = p \log d / \log p_0$ processors.
- An arbitrary-BPRAM algorithm running in time t using p processors can be simulated by an EREW Subset LPRAM running in $\Theta(t \log p_0 / \log d)$ time with $p_0 = p \log d / \log p_0$ processors.
- An EREW (CREW, CRCW) Subset LPRAM algorithm running in time t using p processors can be simulated by an EREW (CREW, CRCW) synchronous-LPRAM running in t time with p processors.
- An EREW (CREW, CRCW) synchronous-LPRAM algorithm running in time t using p processors can be simulated by an EREW (CREW, CRCW) Subset LPRAM running in $O(t \log p_0 / \log \alpha)$ time with $p_0 = \alpha p \log d / \log p_0$ processors, for $1 \leq \alpha \leq d$.

We conclude this section with a simulation of bounded fanin circuits by the Subset LPRAM. The **arc-width** of a circuit is the maximum, over all levels i, of the number of arcs from a node of level $\leq i$ to a node of level $> i$. Corresponding to theorem 5 for the Phase LPRAM, we have the following theorem.

Theorem 14 *Any function computable by a log-space uniform circuit family of bounded-fanin arithmetic circuits of depth $D(n) \geq \log n$, maximum arc-width $W(n)$, and polynomial size can be computed by a log-space uniform EREW Subset LPRAM family running in $O(d\, D(n)/\log d)$ time with $d\, W(n)$ processors.*

Proof. The only synchronization inherent in a circuit is along arcs in the circuit. The simulation outlined in the proof of theorem 5 can be adapted to the Subset LPRAM as follows. We consider slices in the circuit of $\log d$ levels and output the instructions for one slice at a time. We assign one processor per node in the slice. At each slice i, each "node" processor (a) reads from global memory the values of its up to d frontier nodes, (b) mimics the circuit in order to compute its output value, and then (c) writes d copies of the value to the shared memory for each arc that is outgoing to a node in a different slice.

The necessary synchronization is accomplished as follows. We assign d "arc" processors to each outgoing arc for the slice. A node processor has zero, one, or two outgoing "inter-slice" arcs. For each outgoing inter-slice arc, the node processor synchronizes with the d arc processors for the arc. This results in there being a unique arc processor for each copy of a value that will be needed in slice $i+1$. At slice $i+1$, each node processor for the slice synchronizes with the up to d "arc" processors corresponding to the values of its frontier nodes (one arc processor per value).

Note that an arc may be both an incoming and an outgoing inter-slice arc for a slice. In this case, the d processors for the arc are idle this slice and remain assigned to the arc

in the next slice. The synchronization for a slice can be done in $O(d)$ time. Thus each slice can be done in $O(d)$ time. We use $O(d\,W(n))$ processors at a time, so the theorem follows. □

Chapter 5

Semi-Synchronous Programming and Hardware Support

5.1 Introduction

In previous chapters, we defined the Asynchronous PRAM model of computation and studied various questions in complexity theory on this and related models. Each model studied, although an asynchronous model, has the property that programs in the model are *repeatable*. This property is sometimes called *determinacy* [KM69]. Certainly less restrictive models are possible and are beginning to be studied in some detail (e.g. the APRAM [CZ89]). However, we argue in what follows that this "structured" asynchrony is useful in many application domains and may be practical as a basis for an effective programming model.

In this chapter, we broaden our study of asynchronous parallel computation from the strictly theoretical to the more practical issues of programming models and hardware. In section 5.2, we introduce the notion of a "semi-synchronous" programming model, a hybrid of existing synchronous and asynchronous programming models. This model is based on enforcing "structured" asynchrony, so that programs are repeatable.

In section 5.3, we present a new method for supporting fast synchronization barriers in hardware. In order to effectively support a semi-synchronous model such as the Asynchronous PRAM, the target multiprocessor must provide support for fast implementation of the synchronization steps defined in the model. For the case of an Asynchronous PRAM with all-processor synchronization, it is essential that a barrier synchronization primitive require minimal run time overhead. This new method minimizes the synchronization overhead for certain common classes of MIMD machines.

In section 5.4, we discuss a method for supporting pairwise synchronization in hardware that minimizes the overhead. This permits an Asynchronous PRAM with subset synchro-

nization to be supported efficiently in hardware.

Together, we believe these results validate the Asynchronous PRAM as a practical model:

- it supports an effective programming model for many application domains,
- it serves as a good basis for studying algorithms and complexity issues, and
- it can be implemented efficiently in hardware.

These claims will be discussed in detail in chapter 6.

5.2 Semi-Synchronous Parallel Programming

A **programming model** defines the view of the computer presented to the applications programmer. The purpose of a parallel programming model (language, environment) is to provide a framework and tools for expressing parallelism in programs. In some cases, the parallelism is transparent to the programmer: programs are written in a sequential language such as Fortran and then "vectorized" or "parallelized" using a sophisticated compiler that produces code for parallel machines (e.g. [AK85][PW86][AJ88][BCF+88]). In other cases, the programmer is provided with an explicit framework (however modest) in which to think about problem solutions exploiting parallelism, and tools for telling the parallel computer what to do (e.g. [Sny84][ACG86][Ble87][AS88][CM88][LSF88]).

In an often-cited paper, A. Karp [Kar87] commented on the "sorry state" of parallel programming, lamenting the difficulties in programming and debugging parallel programs on existing programming models. Indeed, a variety of programming models have been proposed with not entirely satisfactory results. In this section, we introduce a new programming model for large scale parallel shared memory machines.

5.2.1 Asynchrony and programming models

As discussed in section 1.2.4, writing, debugging, analyzing, and testing programs for asynchronous machines can be very difficult due to (1) the subtleties of dealing with nondeterministic orderings among the communication events during program execution and (2) the lack of simple, global states in programs written in these models. Most existing programming models for MIMD machines (e.g. the EPEX programming model [DRGNP86]) provide little help in overcoming these difficulties. We refer to these models as **fully asynchronous programming models.**

In contrast, **synchronous programming models**, in which the processors are presented as executing in lock-step, are much easier to program, debug, and analyze than

fully asynchronous models. From the programmer's viewpoint, a program reaches a deterministic global state at each step, i.e. each processor executes its instruction i before any processor proceeds to it instruction $i+1$. Debugging is simplified since program execution is repeatable: any bug can be recreated by rerunning the program.

However, as discussed in section 1.2.4, machines that permit the processors to run asynchronously have a definite performance advantage over those that restrict the processors to lock-step execution. (The sole exception may be in application domains that lend themselves naturally to programs with frequent interprocessor communication at regular time intervals between neighboring processors, as in systolic algorithms [Kun82].) The performance advantage of asynchronous machines grows with the irregularity in communication and the number of programs running simultaneously on the machine. Thus, a synchronous programming model will be mapped, in most cases, to an asynchronous machine. As discussed in section 2.1, this mapping is inherently inefficient since the ability of the machine to run asynchronously is not fully exploited and there is (a potentially large) overhead in synchronizing the processors as part of each instruction. Compilers can be used, to some extent, to convert a synchronous program into one more suitable for an asynchronous machine, by removing unneeded synchronization (more on this below).

5.2.2 Semi-synchronous programming models

There are general trade-offs in restricting the sources and types of asynchrony in programming models. Interestingly, all existing models fall into the two extremes: synchronous or fully asynchronous. Our key observation is that there are points in between these two which may be more desirable for many application domains.

In **semi-synchronous programming models**, each processor executes its instructions independently, but all "competing" accesses to a memory location are serialized in a predetermined order. Two accesses to memory are **competing** if they are both to the same location and

- one is a read and one is a write, or
- both are writes that are attempting to write different values.

Otherwise, the two communication events are **non-competing**. The order in which competing accesses are serialized may be data-dependent, but a fixed input must always produce a fixed ordering.

If the order of competing communication events is *not* serialized in a predetermined order, then race conditions can result in which the value read or written depends on the delays encountered during program execution. It is precisely this type of nondeterminism that we seek to avoid. Since there is a predetermined ordering of all competing events in

semi-synchronous models, program computations in these models are indeed repeatable. There are no race conditions.

The idea of repeatability has a long history (e.g. [MSGR61][KM69]), but has been largely ignored by the designers of programming models. In fact, current research either assumes that programs will have nondeterminism (e.g. [MC88]) or explores the extent to which nondeterminism can be *encouraged* in programs (e.g. [CM88]).

Particular semi-synchronous models may place restrictions on the concurrent access of a memory location in order to reflect the (lack of) support for concurrent access in the target machine(s). For example, the model may place restrictions on the concurrent writing of a location if the target machine(s) can not support a large number of write operations to the same location in a short period of time. In asynchronous models, restrictions on concurrent access can be enforced by specifying that the accesses be serialized in the program (since they will likely be serialized during run time). In other words, we will extend our definition of *competing* for these more restrictive models. In a *CREW* semi-synchronous model, two accesses to the same location are competing if at least one is a write. In an *EREW* semi-synchronous model, any two accesses to the same location are competing. These models are more suited for target machines with networks that do not have combination hardware. The general definition given at the beginning of this section (with no further restrictions on concurrent access) corresponds to a *CRCW* semi-synchronous model.

On the other hand, higher level primitives may be provided for coordinated actions such as a "synchronized" concurrent write step, in which the processor with highest priority succeeds in writing the location and all others fail. When treated as a "black box", this primitive is repeatable. A primitive or library routine is a **black box** if the programmer has access only to the inputs and outputs of the routine. These black boxes are assumed to be completely debugged, and thus will work regardless of the delays that may occur on different runs or due to the presence of a debugger. These higher level primitives simplify the programming task. In order to have the programming model reflect the target machines, the model should assign a cost to the higher level primitive that reflects its expected running time on the target machines.

Semi-synchronous models permit asynchronous execution (for efficiency), and yet non-determinism is restricted to non-competing memory accesses (for ease of programming and analysis). A detailed comparison between semi-synchronous and fully asynchronous programming models will be made in section 6.2.

5.2.3 All processor synchronization vs. subset synchronization

There are tradeoffs between using a semi-synchronous programming model with only all-processor synchronization (as in the Phase LPRAM) and one with subset synchronization. Subset synchronization is likely to lead to faster programs, since processors wait for each

other only when needed. However, the model is more complicated since synchronization points do not provide simple, global states during execution.

The efficiency of the all-processor version can be improved as follows. Once the program is debugged, an optimizing compiler can be used to "relax" the synchronization barriers. Instead of having *all* the processors synchronize at each barrier instruction, a smart compiler can determine cases in which only subsets of the processors need to synchronize among themselves before continuing in the program. Assuming the compiler works correctly, this will not introduce any new bugs to the program. This optimization should be used only after the program is believed to be debugged, since it may destroy some of the desirable properties of the model (e.g. easily specified global states at each synchronization point).

An important open question is the extent to which compiler technology can transform more synchronous programs into less synchronous ones.

5.3 Hardware Support for Barrier Synchronization[1]

In order to effectively support an Asynchronous PRAM with all-processor synchronization, it is essential that a barrier synchronization primitive require minimal run time overhead. Barrier synchronizations can exact a tremendous performance penalty if not implemented properly [Axe86].

In this section, we present a simple hardware mechanism for supporting efficient synchronization barriers in certain classes of MIMD parallel computers. We generalize this mechanism to yield a family of methods for implementing barriers that differ in their cost and their ability to adapt to varying run time delays. Our focus is on shared memory machines with hundreds (or more) of interconnected processors. In fact, the performance advantage of these barrier methods over existing methods increases with the number of processors. For the purpose of this thesis, we omit some of the details and generalizations of our barrier method. These details can be found in the full paper on the method [BGSS89].

5.3.1 Introduction and terminology

The family of barrier methods described here is designed for MIMD machines working on a single problem (no multiprogramming). The barriers are all machine-wide, so a barrier instruction must be included in the local programs of all processors for every barrier. Nevertheless, special treatment is given to processors that do not truly participate in a given barrier.

As we shall see, our methods for barrier implementation are particularly attractive when used in machines employing **dancehall** multistage interconnection networks with monotonic routing. In these machines, the nodes in the network graph can be partitioned

[1]This section represents joint work with Yitzhak Birk, Jorge L.C. Sanz, and Danny Soroker.

into "stages" such that (a) all processors are in the first stage, (b) all memory banks are in the last stage, and (c) all links are between stages. Paths from processors to memories follow strictly increasing stage numbers, and paths from memories back to processors follow strictly decreasing stage numbers; the two sets of paths are disjoint. Furthermore, any processor accesses any memory location in exactly one pass through the network. Examples of such machines include the IBM RP3 [PBG+85], the NYU Ultracomputer [GGK+83], Cedar [Gaj83], and the BBN Butterfly [BBN86].

For dancehall networks with monotonic routing, our methods implement barrier synchronization in one pass through the network, using special "barrier messages". Run time synchronization overheads are minimized since the processors can pipeline barrier messages with other memory requests, and thus continue to work without waiting for the barrier to complete. The logic needed in the switches to support the method is very simple, and does not slow down normal message traffic. We know of no previous synchronization method that provides both these advantages.

For shared memory MIMD machines that do not belong to the foregoing class, our methods become more complicated and are somewhat less attractive, but are nevertheless interesting. In particular, we present implementations of our methods for various network topologies such as butterflies, hypercubes, and meshes.

We will use the following definitions. Traffic from processors to memory will be referred to as **messages**. Traffic from memory to processors will be referred to as **replies**. Both messages and replies are assumed to be autonomous and can be thought of as packets.

Since there will typically be multiple synchronization barriers in a single program, we use the term **barrier instance** to refer to each one. The actual enforcement of the required synchronization will be referred to as the **implementation** of the barrier instance. **Barrier messages** will be used to denote messages that are sent strictly for the purpose of implementing a synchronization barrier.

Data (non-barrier) messages issued by a processor prior to its reaching a given barrier instance in its program will be referred to as **pre-barrier** messages with respect to this instance; those issued after reaching a barrier instance will be referred to as **post-barrier** messages. (A message is **issued** when it is sent by its originator.)

Many barrier synchronization methods have appeared in the literature. In the next section, we survey these existing methods.

5.3.2 Existing methods for barriers

We distinguish seven types of existing methods for supporting barrier synchronization: wire, fetch-and-add at memory bank, fetch-and-add in a combining network, combining-with-holding network, fetch-and-add in holding tree, multi-pass tree, and double network. In this section, for each type, we give a brief description of the method and then evaluate

it in the areas of run time overheads, hardware costs, and generality.

There are four potential sources of run time overheads associated with synchronization barriers.

- Longer switch cycle times due to the hardware complexity of the barrier method.
- Slower global memory access times for data messages due to the increased network congestion and delays resulting from barrier messages (e.g. hot spots).
- Clock ticks wasted at processors waiting for other (slower) processors to begin their participation in the barrier.
- Clock ticks spent by a processor participating in the barrier after the above overheads are factored out.

We define the following quantitative measure of this latter type of run time overhead. Consider two sets of messages, pre-barrier and post-barrier, separated by a barrier instance. For a given barrier method, let its **barrier delay** be the minimum number of clock ticks between the issuing of a pre-barrier message and the issuing of a post-barrier message when a particular barrier method is used. This quantity is intended to reflect the best case behavior of the barrier method: it assumes that messages do not encounter any delays due to other messages (e.g. as a result of contention for a link or memory bank).

For concreteness, we assume that a processor can write (read) a word to (from) memory in an empty network in exactly d ($2d$) clock ticks. Moreover, a processor can pipeline messages to memory. In an otherwise empty network, a sequence of r read messages, issued during r consecutive clock ticks by a processor, will complete in $2d+r$ clock ticks. Likewise, a sequence of w write messages, issued during w consecutive clock ticks by a processor, will complete in $d + w$ clock ticks. Given these assumptions (the same ones as used in the Asynchronous PRAM with communication delay d), we can calculate precisely the barrier delay associated with any barrier method.

5.3.2.1 Wire

Our first existing barrier method involves implementing a barrier instance by having all the processors "pull on a wire". Initially, every processor "holds the wire down", resulting in a low voltage. As processors reach the barrier instance, they let go of the wire. When the last one lets go, the voltage goes up and all processors know that the barrier has been implemented. At that point they may begin issuing post-barrier messages, and hold the wire down until the completion of the next barrier instance.

Run time overheads. Since the wire is not part of the processor-memory network, it neither increases the switch cycle times nor slows down processor-memory traffic. The barrier delay is calculated as follows. A processor must wait for its pre-barrier messages and

replies to complete before it can release the wire. If its last pre-barrier message is issued at clock tick k, then the message completes in clock tick $k+2d-1$, if it is not delayed by other messages. (A write takes $2d$ as well since the processor must wait for an acknowledgement that the write has completed.) The processor releases the wire, and in the best case, all other processors have already released the wire. If so, then the processor can immediately issue its first post-barrier message. Thus the wire method has barrier delay $2d$ if pulling on a wire can be done in one clock tick. As the number of processors increases, however, it becomes impractical to use a single wire. Thus, asymptotically, the wire method has barrier delay $2d + O(\log p)$ with p processors.

Costs. A single wire for synchronization can be added to a machine at almost no cost. As the number of processors increases, however, implementation of the wire method becomes more complicated, as multiple wires are needed due to limited fan-in of devices.

Generality. The method works for implementing a barrier involving all the processors, however it can not do multiple barriers in parallel. The method works along side any processor-memory network, but special care must be taken when not all the processors are in working order.

5.3.2.2 Fetch-and-add at memory bank

A **fetch-and-add** operation [GLR83][GGK+83] is a synchronization primitive with the following semantics. If a processor issues a *fetch-and-add(X, c)* instruction, where X is the name of a shared memory location and c is an integer, then a message is sent to location X. Upon arrival of the message, memory location X is read, the value of X is increased by c and written back to location X, and the old value of X is returned to the processor. The hardware ensures that only one processor has access to X at a time.

Given a primitive for performing fetch-and-add at the memory banks of the machine, a barrier among p' processors can be implemented by having each such processor execute a *fetch-and-add(X,-1)* instruction, where location X is first initialized to p'. After issuing its fetch-and-add, each processor keeps reading location X to see if the barrier has completed (i.e. the value is zero). This method is used in the BBN Butterfly [BBN86].

Run time overheads. Since the hardware to support this method is at the memory banks and not in the switches, this method does not slow down the switch cycle time. However, considerable network traffic and delays can occur as a result of multiple processors polling the same location repeatedly to test whether X has become zero. These requests are serialized at the memory bank containing X. If the network is not a combining network, this creates a bottleneck ("hot spot") which potentially delays *all* network traffic, including messages to other locations and by other users [PN85]. This problem can make the method impractical for synchronizing large groups of processors.

The barrier delay for this method is $4d$, as follows. A processor must wait for its pre-

barrier messages and replies to complete before it can issue its fetch-and-add. If its last pre-barrier message is issued at clock tick k, then it completes in clock tick $k + 2d - 1$ (if the message is not delayed by other messages). The processor issues its *fetch-and-add* instruction during clock tick $k + 2d$. It begins polling to see if all processors have arrived at clock tick $k + 2d + 1$. At best, all processors have arrived in time for the polling to succeed with this first message. After receiving the successful reply to the first polling message in clock tick $k + 4d$, the processor can issue its first post-barrier message during clock tick $k + 4d + 1$. Thus the first post-barrier message is issued $4d$ clock ticks later when there is a barrier than when there is not. As indicated above, the actual delay will be much longer due to the serialization of the requests at location X.

Costs. Hardware support for a fetch-and-decrement instruction can be provided at little cost.

Generality. The method can be used to implement barriers involving arbitrary subsets of processors, including multiple subsets in parallel. It works with any network. Since no assumptions are made on the paths used to get to the memory banks, it is easy to avoid faulty components while using this method.

5.3.2.3 Fetch-and-add in a combining network

A combining network (defined in section 1.2.2) can be used to reduce the run time overheads of the previous method. In particular, a combining network provides a more efficient implementation of the concurrent accesses to memory resulting from the previous method. Typically, the paths from the processors to location X form a tree in the network, which we call the **combining tree.** Messages can be combined when they meet at switches in this tree. If messages corresponding to *fetch-and-add(X,c)* and *fetch-and-add(X,d)* meet at a 2-by-2 switch S, they are combined: the value of c is saved for the return trip, and the message sent on is *fetch-and-add(X,c+d)*. When the value of X is returned from memory, the dual switch of S returns (a) the value of X back towards the first processor and (b) the sum of c and the value of X back towards the second processor. Likewise, multiple requests to read X (to see if its value is zero) are combined and then broadcast to precisely those processors requesting X.

The advantages and disadvantages are the same as the previous method, except (i) the combining network reduces network congestion, although there is still heavy traffic due to the polling of location X, and (ii) normal message traffic is slowed down since the switches in a combining network are considerably slower [PN85]. Using q-by-r switches for $q, r > 2$ makes the switches even slower.

The barrier delay for this method is $4d$, the same as the previous method. The actual delay, however, will likely be shorter than that of the previous method since requests to location X are no longer serialized.

5.3.2.4 Combining-with-holding network

In the foregoing combining network, the combining of messages is not guaranteed. In a combining-with-holding network, messages are held at the switches to ensure that combining occurs. This type of network forms the basis for the Fluent machine [RBJ88][Ran89]. In this network, the switches are 2-by-2. Each message is assigned a priority according to its destination. During the course of the routing, each switch holds the first message in a queue until a message of lower or equal priority arrives at the head of the other queue. If the first messages of both input queues have the same priority, these messages are combined. In this way, messages travel between switches sorted by priority. Special messages, called ghost messages and end-of-stream (EOS) messages, are used to keep messages progressing through the network despite the holding policy. In particular, the EOS messages (which are simply messages of lowest priority) are used to implement a barrier after each memory request. The routing strategy, due to Ranade, is designed to support an efficient step-by-step simulation of a CRCW PRAM on a butterfly network [Ran87].

Run time overheads. Since only the first element of each queue is compared and potentially combined, the switches may be less complex than the switches in the previous method. However, the switches are still more complex than in either of the first two methods above (i.e. wire and fetch-and-add at memory bank). Furthermore, since messages are often held in the switches before proceeding, a synchronization penalty may be incurred at every switch for every message. These two factors increase the time to access memory, whether or not the processors are executing a barrier or performing a concurrent access to memory.

On a dancehall multistage network, the barrier completes in only one forward and one backward pass through the network (both passes are needed since EOS messages travel to the memories and back). There is no polling of the status of synchronization variables.

The primary advantage of this method for barriers is that the barrier delay is one. A processor can issue its last pre-barrier message, its EOS message, and then its first post-barrier message on consecutive clock ticks. The routing strategy ensures that any pre-barrier message reaches the memories before any post-barrier message to the same location.

Costs. The switches in the Fluent machine are augmented to include word-size comparators for comparing priorities, extra buffer space for storing return information, and arithmetic units for operating on the data fields of messages being combined.

Generality. The method can be used only for barriers among all the processors. Because messages are held at each 2-by-2 switch until an appropriate message arrives at the other queue, the machine deadlocks if one processor fails to issue an EOS message, or if an EOS message is lost. The Fluent machine uses time-out methods in the switches to prevent deadlock. Given an Omega network (i.e. a butterfly network with a few extra columns for fault tolerance), a faulty link into a switch can be avoided if the entire switch is avoided (so that messages arriving on a good link are not held waiting for a message to arrive on the

bad link).

The routing strategy has been generalized to meshes and other network topologies [LMR88], but not to q-by-r switches. If the strategy were implemented for q-by-r switches, two serious inefficiencies would arise. In order for messages to travel between switches sorted by priority, q priority fields must be compared in each switch cycle to select the next message to be sent. Furthermore, only one message would be forwarded each switch cycle, not r messages as desired.

As we shall see, our barrier method is similar in appearance to the EOS messages used in this routing strategy. However, there are some differences, both in the applicability and in the resulting performance, as will be discussed later.

5.3.2.5 Fetch-and-add in holding tree

This method, due to Jayasimha [Jay88], is related to the two previous methods. The synchronization barrier is implemented using a primitive, called a *distributed synchronizer*, in which "barrier" messages perform fetch-and-add operations on variables stored in the switches at each level of the network. More precisely, consider the combining tree associated with a location X for a machine-wide barrier. Let each switch in this tree contain a counter variable which is initially set to q, where q is the number of inputs to the switch. To implement a barrier, each processor issues a *fetch-and-add(X,-1)* message. Each such message is sent to the appropriate switch in the first level of the combining tree, at which point it decrements the counter stored at the switch. If the counter is not zero, the message is discarded. The last message to arrive will set the counter to zero. This message continues on to the next level of the combining tree and decrements the counter stored there. The process is repeated for each level of the tree. When the counter at the root of the tree is set to zero, all the processors have synchronized. A second phase follows in which the processors are notified that the barrier has completed. This is done by reversing the first phase: the message at the root is broadcast to the switches at the previous level of the combining tree, and so forth level-by-level, back to the leaves of the tree. At the conclusion of this second phase, all the processors have been notified that the barrier has completed.

Run time overheads. Because the intended functionality is greatly simplified, the switches will be much simpler than those in a sophisticated combining network. There is no polling/busy-waiting needed: processors are signaled when the barrier has completed. Similar to the other fetch-and-add methods, the (best case) barrier delay is $4d - 1$. The actual delay will be longer since at each level of the tree, at least q cycles are needed to decrement a switch's counter variable from q to 0.

Costs. The switches in the network must be augmented to include counters and some additional logic.

Generality. The method works best for barriers among all the processors, since in this case the counters in each switch can be reinitialized automatically by using mod-q counters. Barriers among a subset of the processors require the counters to be initialized according to the shape of the combining tree dictated by the particular set of processors. Moreover, the broadcast method of the second phase must handle selective broadcast at each switch. Faulty switches can be avoided by routing around them and then initializing the counters to be the number of *non-faulty* inputs to the switch.

5.3.2.6 Multi-pass tree

A barrier among an arbitrary set of p' processors can be implemented in $\log p'$ rounds of coordinating pairs of processors in a tree-like fashion, each round involving at least one pass through the network. At each round, each right sibling waits until its left sibling in the current level of the tree has finished the previous level, by busy-waiting on a shared global variable. All processors have synchronized when the processor assigned to the root of the tree performs its final coordination. By retracing the tree in $\log p'$ rounds starting from the root and progressing level-by-level to the leaves of the tree, the processors can be informed that the barrier has completed.

Run time overheads. The main advantage of this method is that it avoids hot spots without resorting to using complex switches, as in combining network methods. The main drawback is that it requires at least $\log p'$ passes through the network. In addition, the busy-waiting at each round can create considerable network traffic. The barrier delay for p processors is $2d(\log p + 1)$.

Costs. No additional hardware support is needed in the network. Hardware support at the processors is essential to avoid the overheads of initiating each round of the tree under software control.

Generality. The multi-pass tree method works on any network, including networks with fast, simple switches. Since no assumptions are made on the paths used to get to the memory banks, it is easy to avoid faulty components while still using this method. Unlike the fetch-and-add methods which use a single memory location, in this method $O(p)$ memory locations are used. In order to synchronize an arbitrary subset of the processors, at each round each active processor must somehow know which memory location to use to coordinate with its neighbor.

Another related method, due to Brooks [Bro86], involves coordinating the processors in a butterfly-like fashion with $\log p'$ passes through the network (as in the complete-exchange algorithm of section 4.2.3). The advantage of this method is that all processors learn when the barrier has completed without busy-waiting on a shared status variable or having a separate phase for notification.

5.3.2.7 Double network

Another method, proposed by the RP3 group [PBG+85], is to use two networks: a non-combining network for normal message traffic and a combining (without holding) network for synchronization (as well as for supporting processor scheduling and concurrent access to memory). Data message traffic is not slowed down by synchronization, since data messages use a separate network that has fast, simple switches. The "synchronization" network for the RP3 supports the fetch-and-add in a combining network method. Another possibility for a synchronization network is to use a combining-with-holding network.

The wire method described at the beginning of this section is also a double network method. The drawbacks of double network methods are (i) having two networks increases the cost of the machine, and (ii) the barrier delay must be at least $2d$ since a processor must wait for its pre-barrier messages and replies to complete in the "data" network before it can issue its barrier message in the "synchronization" network.

5.3.3 Global barriers for dancehall MINs

In this section and section 5.3.4, we present a new family of methods for implementing synchronization barriers in shared-memory machines employing dancehall interconnection networks with monotonic routing and with FIFO links and queues. In such networks, the messages and replies can be thought of as using separate networks, since they use disjoint sets of queues and links. This, along with the fact that processors only interact via the memories and our assumption that messages are not lost, permits us to focus on guaranteeing the correct arrival order of messages at memories. The replies take care of themselves.

Of the barrier methods described in the previous section, only the combining-with-holding network method achieved a barrier delay of one. All others had a barrier delay of at least $2d$. Since combining networks are slow and expensive, we propose extracting the "barrier" aspects of this method for use on simple, fast networks. We show how such barriers can be supported with very simple logic at the switches, avoiding the need for word-size comparators. Holding of messages is done on demand in our method, not for every message as in the combining-with-holding method. In addition, we can efficiently use q-by-r switches, for $q, r > 2$, unlike the previous method. Furthermore, we will generalize our method in several important ways.

- We describe techniques that permit run time adaptability of the enforcement of a barrier instance, permitting certain data messages to "pass through" a barrier that has been delayed due to a slow processor.

- We present extensions to other network topologies such as hypercubes and grids that permit more flexible routing schemes than those for which the combining-with-holding

network method applies.

- We provide a general framework and correctness tests for applying our method to any network.

We begin by describing the method for implementing a global barrier, i.e. a barrier among all the processors.

5.3.3.1 The method

We use two types of messages:

- "D" messages are data messages sent by processors. They contain an address and data.
- "X" messages are barrier messages. Their existence is the only information they carry. (They could carry additional information for other purposes, but this is not necessary for the barrier implementation.)

At programming and/or compilation time, instructions to issue X messages are inserted into the program of every processor at the points at which synchronization barriers are required. (Every processor must have such instructions for every barrier instance, since the barriers are all global.) Thus, a processor issues mostly D messages, with occasional X messages. A processor may continue to issue D messages immediately following an X message without fear of any interprocessor synchronization problems. Of course, its own program may have to wait for some pre-barrier replies to arrive before the processor can continue with its post-barrier instructions.

To explain the method, let us initially focus on a single switch. A switch has a FIFO input queue for each of its inputs, and a FIFO output queue for each of its outputs. Also, it has a path from every input queue to every output queue and some switching logic to control the paths and other aspects of switch operation.

The algorithm at a switch is as follows:

- A D message at the head of an input queue is routed to the tail of the appropriate output queue as soon as possible (i.e. once there is space available in the output queue and the data path is available). Then and only then, the next message in that input queue becomes the head-of-queue.
- An X message, in contrast, is held at the head of an input queue until the messages at the heads of the other input queues of the switch are all X messages.
- When there is an X message at the heads of all the switch's input queues:
  1. An X message is placed at the end of every one of the switch's output queues.

2. The X messages at the heads of the input queues are then (and only then) discarded, thereby permitting new messages to become heads-of-queue.

The switch can operate either synchronously or asynchronously. However, care must be taken to ensure that all intra-switch transfers have completed before step 1 is performed.

To implement a barrier instance, each processor issues an X message when it encounters the barrier instruction in its program. The messages are forwarded by the switches per the foregoing algorithm, and thus constitute a **wave** of messages that travels from the processors to the memories.

5.3.3.2 Correctness

The implementation must guarantee that there can be no deadlock and that a post-barrier message can only reach a memory bank after all pre-barrier messages have reached it. A sufficient condition for satisfying the latter requirement is that every pre-barrier message reaches its destination before the barrier does AND that every post-barrier message reaches its destination only after the barrier reaches it. (Reaching a destination means reaching the head of an input queue and being serviced. For a data message this simply means reaching the head of an input queue, but for a barrier message it also means that the heads of all other input queues of the destination switch are also occupied by barrier messages.) We refer to these two properties of a barrier implementation as **flushing** and **restraining**, respectively, and proceed to show that our methods possess them.

Lemma 21 (flushing property) *An X message traverses any given link only after all pre-barrier messages that use that link.*

Proof. By induction on k, the interconnection stage to which the link belongs. The base case ($k = 0$) holds since each processor issues all its pre-barrier messages prior to issuing the X message, and since a processor does not handle messages other than its own.

Assume that the claim is true for links in level k. Consider the level $k+1$ links outgoing from a switch S. By the induction hypothesis and the use of FIFO queues and links, when an X message reaches the head of an input queue of S, we know that there are no more pre-barrier messages that need to use the link over which it arrived. Thus when X messages reach the heads of all input queues for S, we know that all pre-barrier messages that are routed via S have already been transferred to outgoing queues. By the barrier-forwarding policy, no X messages have been transferred. Only then are X messages placed at the end of every output queue. Given the FIFO policy in the output queues and the links, the claim follows. □

Lemma 22 (restraining property) *An X message traverses any given link before all post-barrier messages that use that link.*

Proof. All post-barrier messages are issued by processors, and a post-barrier message is issued by a processor only after the X message. D messages cannot bypass an X message in a switch's queues, and the X messages are placed in all output queues. This guarantees that the X messages cover the tree of all paths from any given processor to all memory banks. □

It remains to show that there is no deadlock. To do so, we prove another property of our barrier wave.

Lemma 23 (clear route property) *The path of a pre-barrier message (with respect to the i^{th} barrier instance) is always clear of any i^{th} or higher X messages and post-i^{th}-barrier D messages.*

Proof. Follows immediately from the fact that the flushing and restraining properties were true at all network stages, the transitivity of temporal order, and the fact that replies use separate routes. □

The issuing of an X message by a processor for a given barrier instance can thus never result in blocking the path of pre-barrier messages from it or any other processor, so there is no deadlock.

5.3.3.3 Self-regulation

It is also interesting to observe that whenever a processor is behind, no other traffic interferes with its progress. The system is thus *self-regulating* in some sense (negative feedback). This property is not true of most of the other methods, e.g. any of the fetch-and-add methods. Large performance penalties can be incurred if messages from processors that are polling to see if a barrier has completed are allowed to interfere with pre-barrier messages [Jor85].

5.3.3.4 Implementation

By the flushing and restraining properties, the set of X messages used to implement a given barrier instance stay behind any X messages from earlier barriers instances. Barrier instructions thus stay in order. This obviates the need to tag X messages with a barrier instance number, and leads to the following simple hardware implementation. Each message contains a "type" bit that is set to 1 for X messages and to 0 for D messages. The test to see if there is an X message at the head of all input queues for a switch is performed by ANDing the type bits of the messages at the heads of those queues.

5.3.3.5 Comparison with existing methods

The barrier method described in this section minimizes run time overheads as follows.

- It can be implemented with minimal hardware costs: there are no comparators, combining mechanisms, counters, or fancy queues.
- For each barrier, there is only one barrier message that uses any given link or is destined for any given memory bank, so there are no hot spots resulting from the barrier.
- Furthermore, as discussed above, the method is self-regulating. Post-barrier messages are blocked behind barrier messages in queues, creating a clear route to the memory banks from processors that are still issuing pre-barrier messages. This minimizes the time spent waiting for slower processors to begin their participation in the barrier.
- Finally, the barrier delay is 1: a processor can issue its last pre-barrier message, its barrier message, and then its first post-barrier message on consecutive clock ticks.

The primary advantages of our method (as described thus far) over implementing a complete combining-with-holding network are that (i) the network can use switches that are simple, fast, and inexpensive, (ii) holding of messages is done on demand, not for every message, (iii) q-by-r switches, for $q, r > 2$, can be efficiently supported, and (iv) *adaptive* monotonic routing may be used if desired.

The disadvantages are that the simplified network does not support combining and may not have a provably good routing scheme.

Whether or not combining-with-holding networks are worth the price is an interesting open question. By separating synchronization from combining, however, we have developed a simple mechanism for barrier synchronization that can be used with existing, noncombining networks.

5.3.4 Selective barriers for dancehall MINs

We have assumed thus far that all processors in the machine were truly *interested* in the barrier, i.e. each processor needed to have (a) its pre-barrier messages reach their destinations before the barrier and (b) its post-barrier messages to reach their destinations after the barrier, in order for the program to behave correctly. In general, however, some processors may be *uninterested* in a given barrier instance and should not be delayed by it. A **selective barrier** is one in which all processors participate, but some may be interested in the instance and some may not be.

In this section, we describe a way for distinguishing between interested and uninterested processors for any given barrier instance, and illustrate how this can be used for simple runtime adjustments of the enforcement of barrier messages with respect to data messages of uninterested processors, even after the barrier messages have been issued.

To distinguish between interested and uninterested participants, we have uninterested processors issue a dummy barrier message, denoted "\". Thus, we have a total of three different types of messages:

- D - data message
- X - barrier message of an interested processor (true barrier)
- \ - barrier message of an uninterested processor (dummy barrier)

A processor which is interested in a given barrier instance issues an X message at the appropriate position in its program (same as for the global barrier). An uninterested processor issues a \ message. This may be issued any time after the latest X message issued by that processor for an earlier barrier instance and before the earliest X message issued by the processor for a later instance. An early issue helps propagate the barrier, but may unnecessarily block data messages if the barrier is not ready to propagate. The switch model is fundamentally the same as was used for the global barriers.

5.3.4.1 The family of policies

The introduction of the \ messages paves the way to a family of barrier policies. The differences between the family members are in the treatment of \ messages. For brevity, we use HOQ to denote head-of-queue.

- A **forward-porous barrier** permits D messages of uninterested processors to skip over \ messages whenever the \ is at the HOQ and yet cannot be propagated. However, once a D message is in front of a barrier, dummy or real, it never falls behind it.

  The forward-porous barrier policy helps mitigate the effect of a premature issuing of a \ message by an uninterested processor.

- A **backward-porous barrier** permits \ messages to skip over D messages of uninterested processors, but not vice-versa. This skipping occurs inside the queue when the \ message arrives, rather than when it is at the head of the queue.

- A **doubly-porous barrier** is both forward and backward porous, as follows.

  - A D message is permitted to hop over a \ message if and only if (i) there are only \ messages between the D message and the HOQ, and (ii) at least one of the HOQs at the other inputs of the switch is neither \ nor X (i.e. the dummy barrier is unable to advance).
  - A \ message is always permitted to hop over a D message in the queues.

The doubly-porous policy can be summarized as follows. We try to propagate the barriers as best we can, but stuck dummy barriers do not prevent D messages from progressing.

In all cases, a barrier is propagated when and only when all inputs of the switch agree to propagate it, and the type propagated is X unless all inputs support \.

5.3.4.2 Correctness

To accommodate the introduction of uninterested processors, we make the following revision to our definition of pre-barrier and post-barrier messages. A data message is a **pre-barrier** (**post-barrier**) message with respect to every barrier instance appearing later (earlier) in the issuing processor's program and in which that processor is interested.

Lemma 24 (correctness) *All members of the family operate correctly and retain the flushing and restraining properties of the global barrier.*

Proof. Since X and \ messages are never allowed to overtake any other X and \ messages, their order is preserved and barrier instances are thus never confused. Since the type of barrier messages placed on all output queues by a switch is "conservative", i.e. it is an X even if only one of the enabling barrier messages was an X, messages issued by an interested processor always hit a true barrier and wait behind it. □

5.3.4.3 Implementation

The doubly-porous policy lends itself to a simple implementation. We use the same basic switch structure as for the global barriers, except that each queue in the switch is now replaced with two queues. The *primary queue* contains only X and D messages, whereas the *control queue* contains only X and \ messages. An example is shown in figure 5.1.

Each input to a switch, consisting of a pair of queues, contains the following state information:

- Whether it is blocked for D messages. It is blocked if and only if HOQ(primary)=X.
- Whether it agrees to propagate a barrier and, if so, what type of barrier. Propagation of a \ barrier is supported if and only if HOQ(control)= \. Propagation of an X barrier is supported if and only if HOQ(primary)=HOQ(control)=X.

Implementation of either of the one-way-porous schemes appears to be more complicated and is not discussed here.

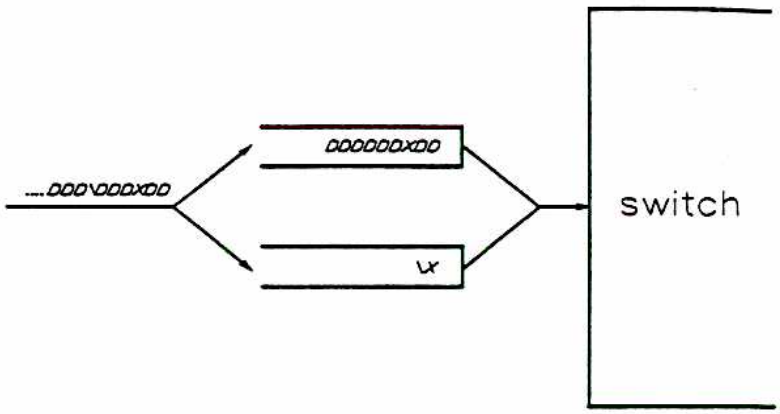


Figure 5.1: Implementation of doubly-porous selective barriers. There are two queues for each input to the switch (only one switch input is shown). D messages are placed in the top queue, \ messages in the bottom queue, and X messages in both queues.

5.3.4.4 Schemes with enhanced adaptivity

Our goal is to permit the propagation of barriers at the earliest possible time, while permitting D messages that should not be delayed by a given barrier instance (i.e. the issuing processor is uninterested in that barrier instance) to hop over it. The problem in our method is that once a \ message reaches a switch whose other input is reachable from an interested processor, the barrier forwarded will be a true (X) barrier and hence no longer porous to D messages of uninterested processors.

In this section, we present three alternative barrier methods. Each method is more adaptive than the method discussed thus far, but requires more complicated hardware.

The first alternative scheme is to use only X and D messages: every X message would be tagged with an instance number, and every D message of an uninterested processor would be tagged with the delimiters of the instance range in which the issuing processor is not interested. Skipping in both directions would be allowed within this range.

The second scheme, which does not require absolute instance numbers, is as follows. Again, we use only X and D messages, but the X messages are not tagged. Every D message carries a tag indicating the number of barriers over which it may hop and a counter. The counter is initially equal to the tag number. Whenever the D message wants to skip over an X message, this is permitted only if its counter is greater than zero. If the skip takes place, the counter is decremented. Whenever a barrier wants to skip over a message, this is permitted only if the counter is less than the tag. If the skip takes place, the counter is incremented.

In both these two schemes, barrier messages of an uninterested processor can be issued as early as possible since they will not delay any of its own D messages. It is not clear whether

either of these two schemes can be implemented easily in hardware, since each requires the capability to swap any adjacent entries in a queue based on their counter values.

The third scheme uses a more elaborate queuing and message-identification system. The idea is to issue post-barrier messages without delay, but to have them wait in a service-queue at the destination until the barrier arrives. This scheme does not unduly delay D messages and can be used with any network. However, it can create unbalanced message loads at the destinations (e.g. bursts at barrier points). Furthermore, the scheme is not self-regulating, i.e. post-barrier messages can interfere with the progress of pre-barrier messages from processors that are behind. (No deadlock will occur, but the delay may increase.) Finally, this third scheme requires rather complicated bookkeeping, especially to support multiple simultaneous barrier instances.

In summary, although more adaptive schemes are possible, our method may be the best that can be implemented at a reasonable cost.

5.3.5 Barriers for other networks

Previous sections have focused on multistage "dancehall" networks with monotonic routing. The full paper on our barrier method [BGSS89] presents a "generic" barrier method that is suitable, in principle, for any interconnection network and any set of routes. We say "in principle" since the method will be impractical for many sets of routes. The paper introduces a general framework and a set of correctness tests for applying our method to any network. Various implementation/efficiency tradeoffs are discussed as well.

In this section, we demonstrate our method for three specific network topologies: populated multistage networks, hypercubes, and meshes. We follow these with a short comparison to existing methods.

In dancehall MINs with monotonic routing, a single "wave" of barrier messages suffice to implement a barrier instance. To accommodate these other networks, however, multiple waves are needed. Each wave starts and propagates independently. A wave consists of one or more "sweeps", where a **sweep** is a single traversal of all the nodes in the network.

We will present multi-wave barrier methods that ensure the following three conditions:

- No deadlock.
- A pre-barrier message must reach its destination before the last barrier message for the given barrier instance to reach that node does so.
- A post-barrier message may reach its destination only after *all* barrier messages that should reach that node do so.

It is not required that a pre-barrier message reach its destination before *any* barrier message reaches that node.

As will become clear later, the mechanism needed for correctly implementing global barriers in non-dancehall topologies resembles the forward porous implementation for selective barriers in dancehall MINs. Here we need to introduce a third type of barrier message (which we will call "f"). An f message cannot overtake any other message, but regular messages are permitted to overtake it. We will discuss here **only global barriers.** Extensions to selective barriers are discussed in the full paper.

Each processor issues pre-barrier messages, then begins its participation in the barrier (by issuing or forwarding f or X messages). The processor continues to forward waves that arrive at its node. After forwarding the last sweep of each wave that will arrive, the processor can safely issue post-barrier messages, i.e. it is **released.**

5.3.5.1 Populated multistage networks

A **populated** multistage network is one in which there are processors and memory banks scattered throughout nodes in many columns (the common case, when there is a processor and memory bank in every node, is called **fully populated**). A multistage network has **wrap-around** if the first column is identified with the last one (i.e. the columns are ordered cyclically). We will concentrate on such networks in this section. The links of such a network are of two types: clockwise (increasing column number, cyclically) and counterclockwise.

We will consider two classes of routing algorithms - monotonic and uniformly monotonic. *Monotonic* means that each message route is either totally clockwise or totally counterclockwise. *Uniformly monotonic* means that all routes are monotonic with the same orientation (say clockwise).

Monotonic routing. The barrier implementation involves two waves, both of which start at column 0. One wave propagates clockwise and the other counter-clockwise. The number of sweeps of each wave is determined by the need to flush all pre-barrier messages, and is thus only a function of the message routes. In the butterfly network, for example (as well as various others with $\log p$ columns for p processors), the maximum (monotonic) distance between any two nodes is 1.5 rounds. However, since the message route may start in column $\log p - 1$, whereas the waves start at column 0, three sweeps are required. In the first two sweeps, f messages are used (to prevent deadlock). In the third sweep, X messages are used.

The correctness proof of this scheme can be found in the full paper.

Uniformly monotonic routing. The barrier implementation here is very similar to that of monotonic, only simpler. Instead of two waves there is only one, which must travel in the same direction as the messages. As before, the wave starts at column 0 and makes three sweeps.

It is also possible to permit non-monotonic routing. However, the number of sweeps required increases with the complexity of the routes.

5.3.5.2 Hypercube example

In a hypercube of dimension n there is a node corresponding to every binary string of length n, and there is a bidirectional link between two nodes if and only if they differ in one bit (we identify a node with its corresponding string). We define a link as *upward* if traversing it increases a bit value (from 0 to 1). Otherwise the link is *downward.* Let $\overline{0}$ and $\overline{1}$ be the node of all 0's and all 1's, respectively.

We can view the hypercube as a multistage network, wherein two nodes are in the same stage if they have the same *Hamming weight* (i.e. the same number of 1's). Using this view, there is a natural partial ordering of the nodes according to their weight. This ordering is equivalent to suspending a hypercube by node $\overline{1}$ (see figure 5.2). We will permit *one-turn* routing: each source-destination route will be either UD or DU, where UD is a route whose prefix contains only upward links and whose suffix contains only downward links. DU is symmetrical. The barrier will have two waves, each consisting of two sweeps, up and down. One wave starts at $\overline{0}$, sweeps up to $\overline{1}$, and then sweeps back down to $\overline{0}$. The other wave is symmetrical, and starts at $\overline{1}$. The first sweep of each wave uses f messages (to prevent deadlock), while the second sweep uses X messages.

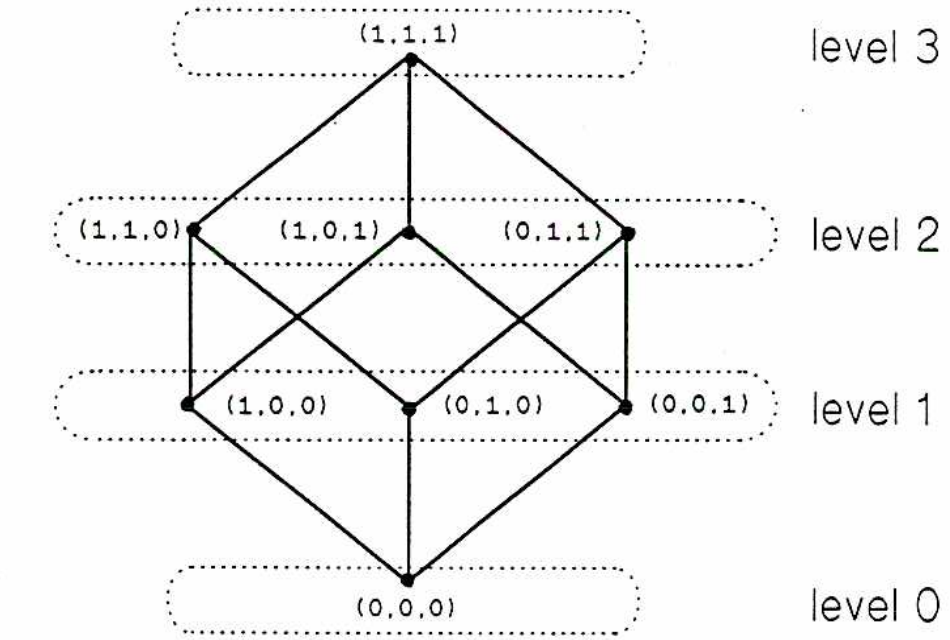
As is the case with uniformly monotonic routing in multistages, if all routes are UD then the barrier implementation requires only one wave, starting at $\overline{0}$. Interestingly, there is an asymmetry between the delays at the nodes in this case. Node $\overline{0}$ must begin its participation in the barrier before the barrier can propagate, and it is released only after all other nodes have been released. At the other extreme, node $\overline{1}$ is the last to be required to participate in the barrier and the first to be released. Thus node $\overline{1}$ is a candidate for a larger workload than other nodes, since it experiences the shortest barrier delay.

Correctness proofs of the schemes described here are given in the full paper.

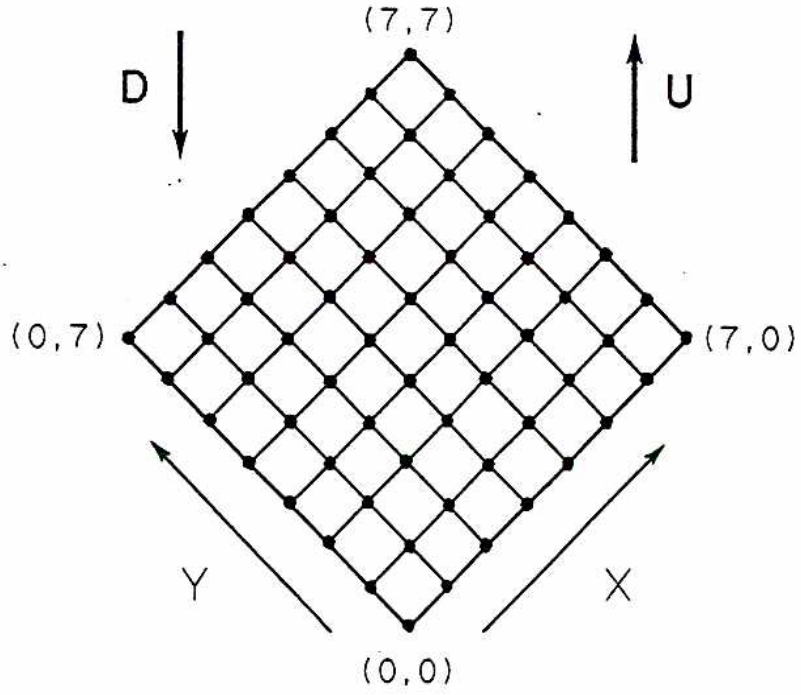
5.3.5.3 Grid example

For simplicity of exposition, we will discuss here square, two-dimensional grids. The discussion can be extended in a natural way to rectangular and higher dimensional grids. In a (square two-dimensional) grid, each node has two coordinates, (x, y) , $0 \leq x, y \leq n$. Two nodes are connected by a link if they differ in only one of their coordinates, by 1. A link can go *up, down, left* or *right* with the obvious interpretation.

First observe that the "natural" barrier scheme for grids does not work. By this we mean the scheme in which one wave starts at each side of the grid and propagates to the opposite side. For example, one wave starts at the set of nodes with x-coordinate 0 and propagates to the set of nodes with x-coordinate n. This method does not work because most message routes are not covered by a single barrier wave (which violates one of the correctness conditions given in the full paper).



(a)



(b)

Figure 5.2: (a) The 3-dimensional hypercube viewed as a multistage network. Nodes with equal Hamming weight are in the same level. (b) The 8-by-8 grid viewed as a multistage network. The Up and Down directions are marked.

However, if the grid is viewed as a multistage network, where two nodes are in the same stage if their weight is equal (where weight here means the sum of the coordinates), then the scheme used for hypercubes will work for grids. The ordering on the nodes is equivalent to suspending the grid by node (n, n) (see figure 5.2). A UD route in this case would be a route whose prefix contains only "upward" links (i.e. only up links and right links in the original grid), and whose suffix contains only "downward" links (i.e. only left links and down links in the original grid). The barrier scheme for UD routes would have one wave, starting at $(0, 0)$, sweeping up to (n, n), and then sweeping back down to $(0, 0)$. As before, nodes are released after forwarding the second sweep of the wave. Thus, similar to the hypercube with UD routes, node $(0, 0)$ has the longest barrier delay while node (n, n) has the shortest.

If more flexibility in routing (beyond UD routes) is desired, then we can define DU, LR, and RL routes and use corresponding waves. For example, an RL route is one whose prefix contains only right links and down links in the original grid, and whose suffix contains only left links and up links. The corresponding wave starts at $(0, n)$, sweeps across to $(n, 0)$, and then sweeps back to $(0, n)$.

5.3.5.4 Comparison with existing methods

In contrast to the case of dancehall MINs, the barrier delay may be greater than one for our method on other topologies. Because waves are initiated independently of one another, the barrier delay depends only on the maximum number of sweeps in a wave (and not the number of waves). As in section 5.3.2, let $2d$ be the number of clock ticks required for a processor to retrieve a word from a memory on the opposite side of the network. Furthermore, assume that a single sweep across the network takes d ticks. If k is the maximum number of sweeps in a wave, then the barrier delay for our method is kd.

The barrier delay of our method compares favorably to the wire method ($2d + O(\log p)$ delay) when $k \leq 2$ and to the various fetch-and-add methods ($\geq 4d - 1$ delay) when $k < 4$. Moreover, in our method, most nodes in the network experience a barrier delay of less than kd. For example, the barrier delay at nodes in the hypercube or grid with UD routing ($k = 2$) ranges from 1 to $2d$.

We conclude this section with a comparison of our method with the combining-with-holding network method for networks other than dancehall MINs. Ranade's routing technique can be applied to networks other than dancehall MINs. Indeed, it was originally designed for a fully populated butterfly network. Leighton, Maggs, and Rao [LMR88] extended the routing technique to other topologies such as hypercubes and grids. Our approach to imposing a multistage framework on hypercubes and grids is reminiscent of the Leighton, Maggs, and Rao technique of using "leveled" paths for routing in hypercubes and grids. The same arguments for separating synchronization from combining that applied in the case of dancehall MINs (section 5.3.3) apply here, namely (i) the network can use simple

switches, (ii) holding of messages is done only on demand, (iii) switches with more than two inputs and outputs can be efficiently supported, and (iv) adaptive routing may be used.

We elaborate on the fourth advantage. Our method works, in principle, with any routing scheme. Moreover, our method provides *efficient* barrier implementations for many routing schemes forbidden in the Leighton, Maggs, and Rao framework. For example, their technique is in the context of oblivious routes (only the delays at a switch are adaptive: a message is held until a message of lower or equal priority arrives at the head of the other queue). In contrast, even our one-wave-barrier implementation for the hypercube or grid can be used in conjunction with adaptive routing, providing many alternative paths. In their scheme, packets typically do not take the shortest source-destination routes. For example, their routes for high dimensional grids and butterflies must first travel to a random intermediate destination and then on to the true destination. Our routes can use the shortest path from source to destination if desired.

5.3.6 Discussion

We have presented a family of efficient schemes for implementing synchronization barriers in shared-memory MIMD machines employing a variety of interconnection topologies. These barriers permit processors to begin issuing post-barrier messages before all pre-barrier messages reach their destinations, yet guarantee that no post-barrier message reaches any given destination before all pre-barrier messages have reached it. This is very useful in conjunction with the Asynchronous PRAM and the semi-synchronous programming model, since it permits consecutive computation phases to be partly overlapped.

We were able to extend the methods to support "selective" barriers. These handle messages of processors that are not interested in a given barrier instance in a flexible manner, by permitting them to skip back and forth over the irrelevant barrier instance to prevent unnecessary waiting. Nevertheless, one would like to support synchronization barriers among multiple sets of processors in parallel. For example, in multi-user and multi-programmed environments, a barrier in one multiprocessor program should not unduly slow down other multiprocessor programs running at the same time. This does not seem possible with our method, unless perhaps each program is assigned its own contiguous block of processors in the network. On the other hand, barriers among all the processors may be sufficient for *time-sharing* multiple users, where each user is given the entire machine for a short duration ("time slice").

We have shown that the run time overheads in our method are smallest for dancehall MINs with monotonic routing. Nevertheless, our methods can be implemented with any network topology, including non-multistage ones. A given network and barrier scheme determine the set of allowable message-routes. However, apart from the constraints on route lengths and monotonicity, the message-routing policy is independent of our scheme,

and can be quite flexible, even permitting adaptive routing. Another interesting property of our methods is self-regulation: a processor that falls behind finds its paths to memory clear of post-barrier messages of other processors.

The cost of implementation and the performance benefit vary with the topology, the set of permissible routes, and the choice of barrier waves. Hypercubes and grids, for example, permit very efficient implementations with the appropriate routing restrictions.

The barrier messages offer a powerful mechanism for gathering and disseminating information throughout the machine. Possible uses of this mechanism warrant further exploration, especially in view of the fact that they are essentially free of charge since the barrier messages contain very few bits of information for barrier-implementation purposes, and the remaining bits are available for other purposes.

5.4 Hardware Support for Pairwise Synchronization

In this section, we discuss a method for efficiently supporting the pairwise synchronization associated with an *Asynchronous PRAM with subset synchronization*. We first present a cache protocol that supports the pipelined reads and writes of an Asynchronous PRAM. This protocol maintains memory consistency in the presence of multiple outstanding shared memory references for each processor. Then we show how a mechanism for pairwise synchronization can be encorporated into this cache protocol.

A **cache memory** for a set of main memory locations S is an auxiliary memory that temporarily holds the values of a subset of the locations in S. A cache memory local to a processor is smaller and physically closer to the processor than main memory. Thus the access time to cache memory is typically only a few processor cycles, far less than the access time to main memory. Associated with the cache is a many-to-one hash function f that maps addresses in S to addresses (lines) in the cache. This function defines which cache line is used to hold the value of a particular location.

We describe a simple cache policy/protocol suitable for Asynchronous PRAM programs running on a parallel computer with many processors, large memories, and a high bandwidth network. Our policy is related to the many existing policies; it is derived from a combination of known ideas. Our policy differs from existing policies in its support of a semi-synchronous programming model (as opposed to a fully asynchronous model) and its emphasis on pipelining memory requests while maintaining a single instruction stream at a processor. Supporting a semi-synchronous model, instead of a fully asynchronous model, greatly simplifies the cache policy. The purpose of this section is to provide evidence of the practicality of the Asynchronous PRAM by showing how simple an effective cache policy for the Asynchronous PRAM can be.

There are four types of memory operations in the Asynchronous PRAM, as follows.

Consider a local operation on the Asynchronous PRAM, i.e. a RAM operation where the operands are in private memory and the result is stored in private memory. For convenience, we will view a local operation as simply a request to copy the value of each operand from its private memory location into a register and then copy the value in a register into the target private memory location. These two copy operations are denoted **private read** and **private write** operations, respectively. (This corresponds to load-store uniprocessor architectures [Pat85] where the two operations are called *load* and *store*, respectively.)

The other two operations are **global read** and **global write**. On a global read, a *copy* of the shared location is stored in a private location. The processor has exclusive use of the value, which simplifies the cache policy.

Throughout this section, let s be a shared memory location, p a private memory location, and r a private register. The four types of operations to support are global read ($p := s$), global write ($s := p$), private read ($r := p$), and private write ($p := r$). Note that each operation involves both a read and a write. Later, we will add a fifth type of operation, a *synchronized read* ($p := \$s$), for the purposes of implementing synchronization.

Loading a shared memory location into a register is performed using two instructions: a global read followed by a private read. Dividing the shared memory read operation into two parts permits a processor to continue with other work while the contents of the shared location is brought local to the processor. This division is similar to the split-phase reads of a dataflow machine [NA88] and the prefetch operation of the Stanford DASH machine [Gup89]. By pipelining its memory requests in this fashion, a processor can mitigate the performance penalty associated with accessing shared memory.

Our cache policy was designed with the following priorities in mind. We have aimed for a solution with the following properties.

- *Lockup-free* caching for all writes, for global reads, and for synchronization, i.e. neither the processor nor the cache controller should block on a cache miss. This goal minimizes run time delays.

- Aggressive caching. Any read or write to a main memory location should cause the location to be cached (if it is not already cached). If the cache line for the location is full, i.e. a **cache collision** occurs, the current value is discarded. However, the discarded value should be stored in local memory so that it does not need to be refetched from a remote node. This simple greedy approach seeks to minimize the number of main memory accesses needed by active processors.

- Single instruction stream per processor. Each processor has a sequence of instructions that it executes in order. It has a single instruction counter. There is no run time rearranging of instructions based on the arrival times of memory references or synchronization events, as there is, for example, in the HEP machine [Jor85] or in a

dataflow machine [NA88]. We believe that such run time rearranging is not worth the cost to support it. Delays to the processor can be minimized in our scheme by judicious prefetching.

- Minimal state information in cache lines. There should be only a moderate amount of state information in a cache line. There should not be extra address or value fields added to a cache line. This minimizes the physical area needed to hold a cache line.

- Simple memory bank interface. The complexity of the protocol is at the cache interface, not the memory bank interface. The mechanism should not require special hardware at the memory bank itself, e.g. for fetch-and-add [PBG+85], and should not add state information to memory words, e.g. for synchronization [GVW89]. This goal simplifies main memory access, and hence increases memory throughput, and it does not increase the word length of memory.

Our policy uses the configuration of memory banks, caches, and queues depicted in figure 5.3. Each processor has a local memory bank which is divided between private and shared memory locations. There is a cache memory for the private memory locations associated with a processor. There is a cache memory for the shared memory locations associated with a shared memory bank. First-in-first-out queues reside between the cache, the local memory, and the network to help accommodate the varying consumption and production rates of these components. FIFO links connect nodes in the network, and thus the series of messages traveling from a node u to a node v are guaranteed to arrive at v in the order they are dispatched from u.

Each memory location is cached only in the cache *local* to the memory bank in which the location resides. Since a global read is an instruction to copy the contents of a location into private memory, the data returned by a global read of s is stored in a cache line for the private location (in the private cache), not in a cache line for s, as in common cache schemes [DSB88][SD88] which support fully asynchronous programming models. In both shared and private caches, we use a *write-through* cache policy, i.e. a write operation updates both the cache line and the local memory location.

Our cache policy provides an effective way to decrease program running times by the following four means.

- Provide fast references of recently used private memory locations.

- Permit accesses to recently used shared memory locations that avoid accessing the memory bank in which the location resides (although accesses to locations in distant memory banks will still be slow).

- Enable the processors to have multiple private and shared accesses in transit at a time

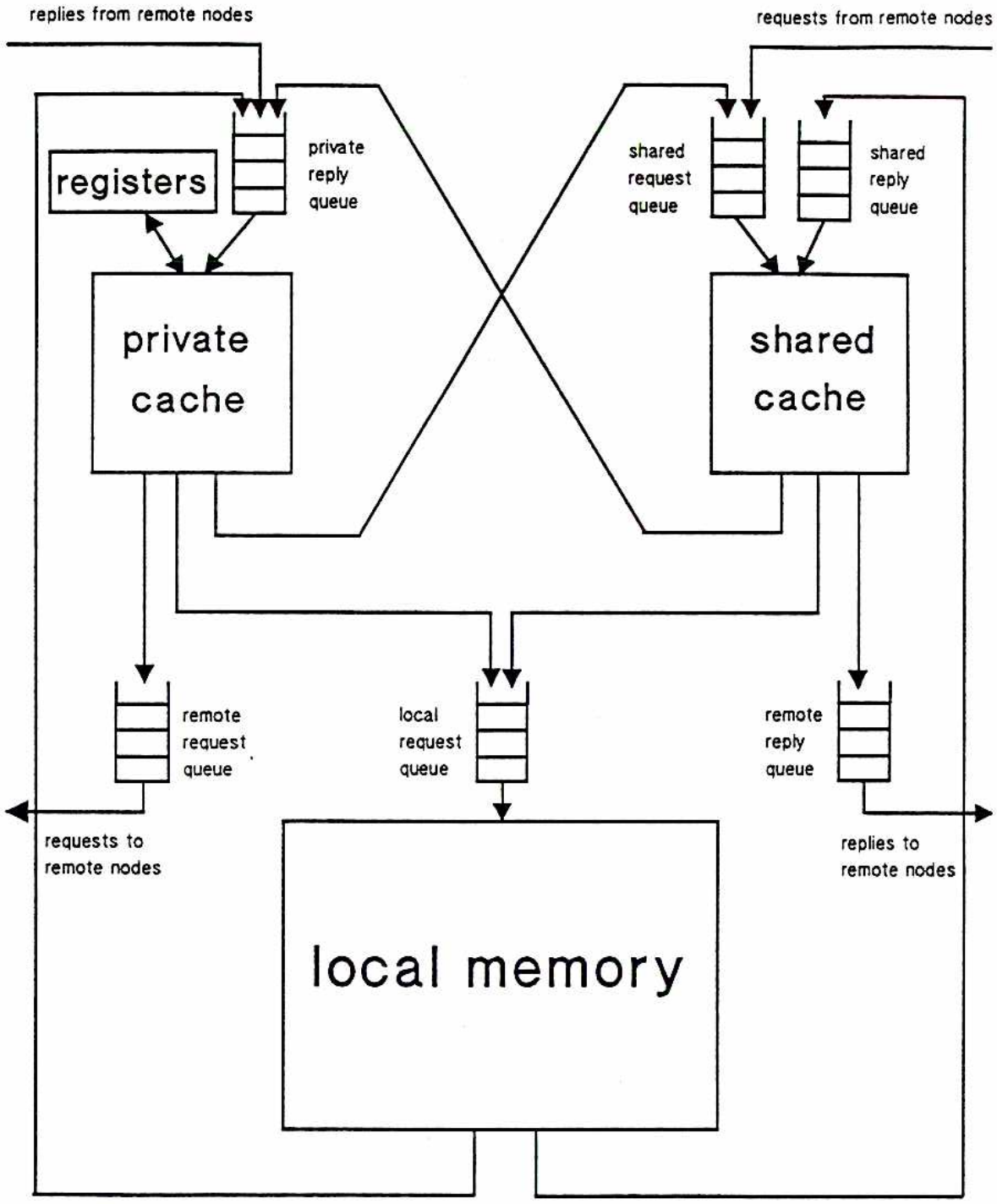


Figure 5.3: The two caches for a node in the network. Each node with both a processor and a shared memory bank has two caches, one for private data and one for shared data. Shown here are the data paths that connect the various memory components at a node.

by keeping track of outstanding memory requests and enforcing the necessary order of accesses to a location.

- Support efficient pairwise synchronization.

5.4.1 A cache protocol for the model

In this section, we will assume throughout that each shared location is accessed in a non-competing manner. In other words, each shared location is either accessed only for reading, only for writing some common value, or only by one processor. Enhancements to the mechanism to handle competing accesses and the synchronization steps between them will be described in the next section.

5.4.1.1 The basic policy

We divide the description of the policy into two parts. First we discuss the policy with regards to private locations, then we discuss the policy with regards to shared locations. As in [SD88], we consider the case where each cache line contains only one word.

Private. We show the policy for each of the four instruction types. In this part, we will view shared memory as a black box as follows. A request to read or write a shared memory location is placed in either the shared request queue (if the shared location is local) or the remote request queue (if the shared location is not local). For a read request, a reply later returns with the correct value to the reply queue. For a write request, the shared location is updated later.

- **Private read** ($r := p$). Wait until cache line $f(p)$ is not blocked.

  If line $f(p)$ contains the value, v, of private location p, then copy v into register r, and continue to the next instruction.

  Otherwise, add a request to read location p to (the tail of) the local request queue. When the local reply returns, place the value returned, v, into line $f(p)$. Copy v into register r, and continue to the next instruction.

- **Private write** ($p := r$). We will write the value to both the cache line and the memory location.

  Wait until cache line $f(p)$ is not blocked. Place the contents of register r into line $f(p)$. Add a request to write the contents of register r to private location p to the local request queue, and then continue to the next instruction. (The private memory location p gets updated a short time later.)

- **Global read** ($p := s$). We will issue a request for the shared memory location, get the reply, and place the value in both the cache line and the memory location.

Wait until cache line $f(p)$ is not blocked. Mark the line as blocked for reading s into p. Add a request to read shared location s to the shared or remote request queue, and then continue to the next instruction.

(When the global reply returns, place the value returned, v, into line $f(p)$. Add a request to write v to location p to the local request queue and unblock the line. The private memory location p gets updated a short time later.)

- **Global write ($s := p$).** We will get the value of p and then issue a write request to the shared location.

  Wait until cache line $f(p)$ is not blocked.

  If line $f(p)$ contains the value, v, of private location p, then add a request to write v to shared location s to the shared or remote request queue, and then continue to the next instruction. (The shared memory location s gets updated later.)

  Otherwise, add a request to read location p to the local request queue. When the local reply returns, place the value returned, v, into line $f(p)$. Add the request to write v to location s to the shared or remote request queue, and then continue to the next instruction. (The shared memory location s gets updated later.)

Shared. In this part, we focus on processing requests in the shared request queue. There are two types of requests. For a read request, the value of the requested location is placed in either the private reply queue (if the return destination is in the same node) or the remote reply queue (if the return destination is in a remote node). For a write request, the requested memory location is updated.

- **read** s. Wait until cache line $f(s)$ is not blocked.

  If line $f(s)$ contains the value, v, of memory location s, then add a reply containing v to (the tail of) the remote or private reply queue, and continue to the next request.

  Otherwise add a request to read memory location s to the local request queue, mark line $f(s)$ as blocked for s, and continue to the next request. (When the local reply returns, place the value returned, v, into line $f(s)$. Add a reply containing v to the remote or private reply queue and unblock the line.)

- **write** s. Wait until cache line $f(s)$ is not blocked. Write the value into line $f(s)$. Add a request to write the value to memory location s to the local request queue, and continue to the next request. (The shared memory location s gets updated a short time later.)

The above policy satisfies the goals outlined in the previous section, namely lockup-free caching for all writes and global reads, aggressive caching, single instruction stream per processor, minimal state information in cache lines, and simple memory bank interface.

Lockup-free caching is used in all cases except for a read of a private location. This occurs in both private read and global write operations. The processor is stalled on a cache miss for a private read in order to satisfy our goal of having a single instruction stream for the processor. The processor is stalled on a cache miss for a global write in order to ensure that two writes by a processor to the same memory location occur in program order. Consider two global write operations, $s := p$ and $s := p'$, where p is not in the cache and p' is in the cache. If the processor is not stalled, then the second write can be added to the remote request queue while p is being fetched from memory, and hence the global writes will occur out of order. Using lockup caching for reads of a private location delays only the processor itself; it does not slow down any other processor. Even the cache controller for the private cache is not delayed: it can process replies in its private reply queue in the meantime.

The only other source of delay is a blocked cache line. This delay occurs only on a cache collision involving two requests *in progress* (if the earlier request has completed, the line is not blocked and can be discarded). We expect this to be a rare occurrence. Otherwise, a processor or cache controller is delayed only as long as it takes to complete the tasks described in the policy.

5.4.1.2 A correctness proof for the policy

In this section, we prove the correctness of our cache protocol by showing that (a) memory consistency is maintained despite the prefetching and lockup-free caching and (b) there can be no deadlock as a result of the protocol. Our task is greatly simplified by the fact that memory is accessed in a non-competing manner. Thus the only possible sources of inconsistency are in locations that are accessed by a single processor. For such locations, we need to show that the value returned on a *load* instruction is always the value given by the latest *store* instruction with the same address [CF78].

We divide the proof into three parts. First we consider the case where there are no cache collisions and show that consistency is maintained for private locations. Next we show that consistency is maintained for shared locations. Finally, using these two results we show that consistency is maintained even in the presence of collisions. In addition, we prove that there can be no deadlock as a result of the policy.

Lemma 25 *Suppose the function f that maps private memory locations to private cache lines is one-to-one (so that no collisions occur). Consider the following initial setup: (a) each private cache line is empty, (b) the most recent store for each private memory location has already been added to the local request queue, and (c) the most recent load or store for each shared memory location has already been added to either the shared request queue (if local) or the remote request queue (if non-local). Then the above policy ensures the*

consistency of registers, private cache lines, and private memory locations with respect to a sequence of program instructions. In addition, the policy ensures that shared memory requests are placed in the (shared or remote) request queue for a processor in program order.

Proof. A register is accessed in program order since the processor is blocked until a private read completes and until a private write has accessed the register.

Now consider private memory. Note that each of the four operations involves a private memory load or store.

The first load or store to a location will bring the location into the cache. A cache line for a private memory location receives the most recent value of the location since (a) on a store, the value is placed directly in the cache and (b) on a load, the request to read the location is added to the local request queue after any previous store, and the processor is blocked until the value is placed in the line.

In the absence of collisions, all subsequent private memory loads will use the value in the cache and all subsequent private memory stores will update both the cache and the memory. A cache line for a private memory location is accessed in program order since (a) a global read instruction blocks its cache line until the returned value is placed in the cache, thereby preventing any other instructions involving the same location to proceed, and (b) any other load or store is completed while the processor is blocked.

Stores to a private memory location occur in program order since (a) the local request queue is a FIFO queue, (b) a global read instruction blocks its cache line until the returned value is placed in the local request queue, thereby preventing any other instructions involving the same location to proceed, and (c) the processor is blocked on a private write until the write request is added to the local request queue.

Requests to read or write a shared memory location at a local (non-local) node are added to the shared (remote) request queue in program order since the processor is blocked until the request is added to the queue.

For each private memory location and its cache line, there is at most one outstanding read request on its behalf. Thus no matter what order the replies return from the shared memory, there will be no confusion since each reply is labeled with its target location. □

Lemma 26 *Suppose the function f that maps shared memory locations to shared cache lines is one-to-one (so that no collisions occur). Consider the following initial setup: (a) each shared cache line is empty, (b) the most recent store for each shared memory location has already been added to the local request queue, and (c) the most recent read reply from each shared memory location has already been added to either the private reply queue (if local) or the remote reply queue (if non-local). Then the policy ensures the consistency of shared cache lines and shared memory locations with respect to a sequence of requests in the*

shared request queue. In addition, the policy ensures that the correct values are returned in response to read requests.

Proof. The first read or write to a location will bring the location into the cache. A cache line for a shared memory location receives the most recent value of the location since (a) on a write, the value is placed directly in the cache and (b) on a read, the request to read the location is added to the local request queue after any previous store, and the line is blocked until the returned value is placed in the line.

In the absence of collisions, all subsequent reads will use the value in the cache and all subsequent stores will update both the cache and the memory. A cache line for a shared memory location is accessed in request order since the cache controller is blocked until the write request has updated the cache or the read request has accessed the cache line (since there are no cache misses). Write requests to a shared memory location occur in request order since (a) the local request queue is a FIFO queue and (b) the cache controller is blocked until the request to update the location has been added to the local request queue.

Since the correct values are added to the remote or private reply queue while the controller is still blocked, the correct values will be returned in response to read requests. Note that we do not need to guarantee the order of replies. □

Theorem 15 *Given initially empty caches, the above policy ensures memory consistency, even in the presence of collisions. In addition, there can be no deadlock as a result of this scheme (assuming no lost messages or other faults).*

Proof. There can be no deadlock as a result of this scheme (assuming no lost messages or other faults). Requests on the local request queue will always complete, so a read s request that blocks a cache line will complete. Thus any line in the shared cache will eventually become unblocked, permitting any new request to complete. A global read request that blocks a cache line waits on a read s request, which will complete. Thus any line in the private cache will eventually become unblocked, permitting any new operation to complete. Since there can not be a cycle of blocking, there can be no deadlock as a result of this scheme.

Now we show that the policy ensures memory consistency even in the presence of collisions.

Consider a private memory location p. The sequence of private memory loads and stores during the program can be partitioned into subsequences as follows. Each subsequence starts with cache line $f(p)$ *not* containing the value of p, and consists of (a) an initial load of the cache line for p in response to a load or store request for memory location p, and (b) zero or more requests to read or write p while the cache contains the value of p. Note that this is precisely the setup for lemma 25 and that no collisions occur within a

subsequence. It follows by an easy induction proof using lemma 25 that the policy ensures memory consistency.

We now consider a shared memory location s. Requests to read or write s, where s is local (non-local), are added to the shared (remote) request queue in program order since the processor is blocked until the request is added to the queue. Since the links in the network are FIFO, these requests are added to the local shared request queue in program order. Similar to the private case, an easy induction proof using lemma 26 shows that the policy ensures memory consistency and that replies are returned in response to read requests. Putting together the two cases yields the theorem. □

5.4.1.3 Cache line format

The above policy can be implemented using a cache line with the following fields.

- **Status.** There are three possible values.
  1. **Invalid.** The cache line can not be relied upon to contain useful information. The line is valid if the status is not "invalid".
  2. **Ready.** The cache line contains the value of the memory location indicated by the address field.
  3. **Blocked.** There are two cases. (a) For a line in a private cache, the cache line has been preallocated for a (local) private memory location, awaiting a value that is on its way in response to a read of a shared memory location. (b) For a line in a shared cache, the cache line has been preallocated for a (local) shared memory location, awaiting a value that is on its way in response to a request to read the local location.
- **Address.** For a valid line, this field contains the memory location to which this line corresponds.
- **Value or requested address.** If the status is *ready*, this field contains the value of the cached location. If the status is *blocked*, this field contains the address requested.

5.4.1.4 An example

We now turn to a short example. Consider the problem of adding two vectors. A program for adding two vectors stored in shared memory is given below. We use two new instructions: **issue** and **complete**. These two primitives enclose a set of instructions that comprise a *pipelining block*. The instructions in a pipelining block will be issued one after another without waiting for any global read or write instructions in the block to complete. Thus no instruction within a block may use the value returned by any read instruction in the same block. These values may be safely used only by instructions that follow the block.

Vector Sum program:

/*
inputs: Two vectors, $A_0, A_1, \ldots, A_{n-1}$ and $B_0, B_1, \ldots, B_{n-1}$, are stored in the shared memory.
outputs: The output vector $C_0, C_1, \ldots, C_{n-1}$ is stored in the shared memory, where, for all i, $0 \leq i < n$, $C_i = A_i + B_i$.
description: Each processor reads in two elements, adds them together, and writes the result. The variables x, y, and z are private variables.
*/

for all processors i **in parallel do** ⟨

    issue
        $x := A_i$;
        $y := B_i$;
    complete
    $z := x + y$;
    $C_i := z$;

⟩

Consider processor i and suppose that A_i has value a, B_i has value b, both A_i and B_i are in their respective caches, and that $c = a+b$. Figure 5.4 shows the instructions executed by processor i, the actions taken, and the state of its private cache after these actions have been performed.

5.4.2 Cache support for synchronization

Section 5.4.1 presented a protocol for handling non-competing memory accesses. In this section, we show how to enhance the protocol to handle competing accesses and the synchronization steps between them.

We add a fifth type of operation, **synchronized read** ($p := \$s$), which reads a value into p when a successful pairwise synchronization has been made at s. (As we shall see, the particular value placed in p is not important.) This instruction invokes a new shared memory request, **synchronize** s. As in section 4.2, synchronization is between two "partners" that wait for each other at a common location s and return once both have arrived at s.

Synchronization is supported at the cache level: the memory location s is neither read nor written to. The first of the partners to arrive places its return address in the cache line. When the second one arrives, a reply is sent to each partner. If a cache collision occurs prior to the arrival of the second partner, the first partner is dropped from the cache. The return address in the line is used to notify the first partner that it must reissue its request to synchronize at s. This contrasts with synchronization methods that include synchronization bits with each location in memory (e.g. [GVW89]).

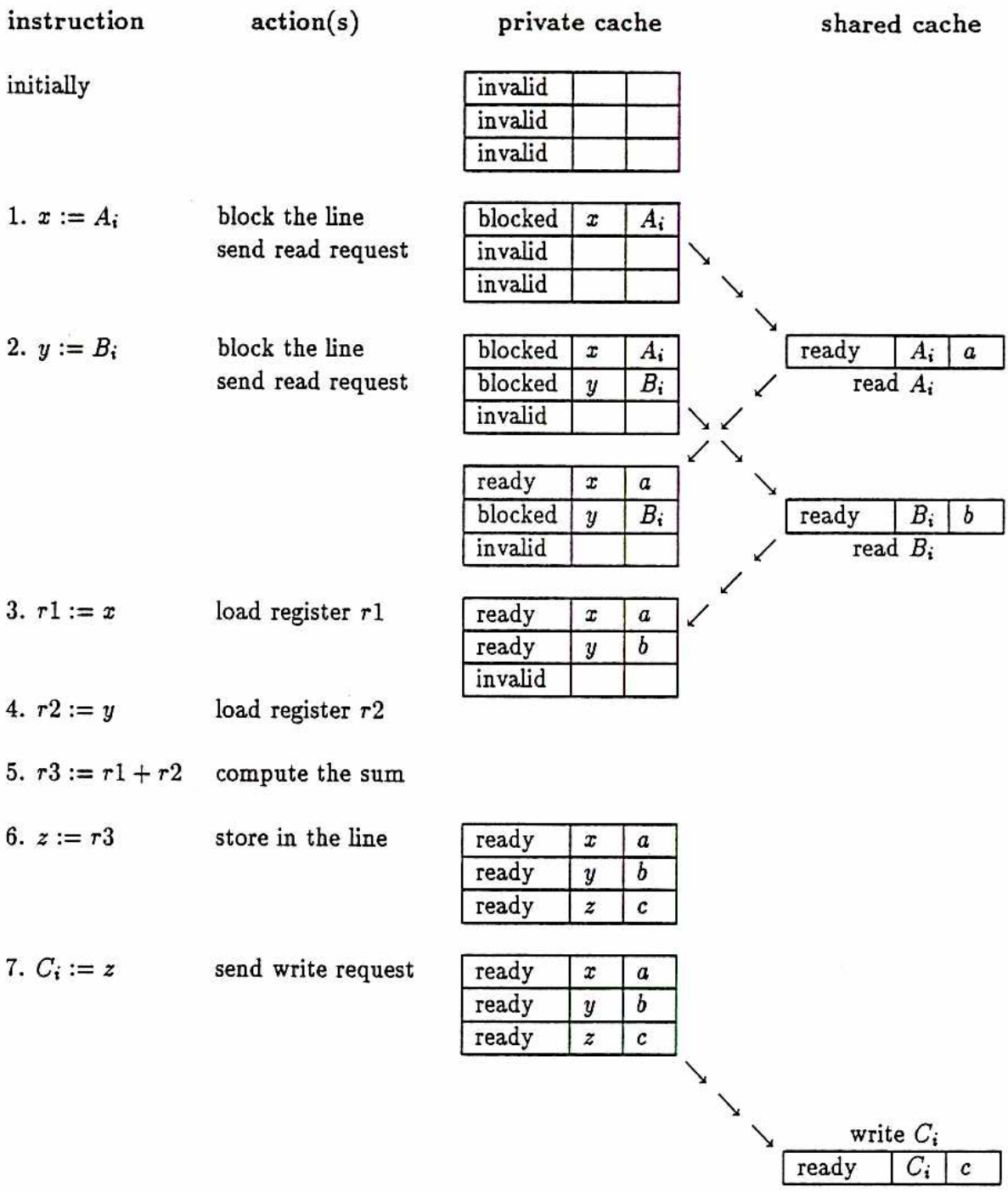


Figure 5.4: An example of the cache policy is shown for the Vector Sum program given in the text. The first column shows the instructions executed by processor i, where $r1$, $r2$, and $r3$ are private registers. For each instruction, the second column lists the actions taken by the cache controller, and the third column shows the state of the private cache after these actions have been performed. The last column depicts the arrival of the shared memory requests at the shared cache and the subsequent departure of the shared memory replies back towards the private cache.

5.4.2.1 The full policy

As before, we divide the description of the policy into a discussion on private locations and a discussion on shared locations.

Private. The policy for the original four operations is unchanged.

- **Synchronized read (p := $s).** Wait until cache line $f(p)$ is not blocked. Mark the line as blocked for synchronizing at s and returning the status into p. Add the request to synchronize s to the end of the shared or remote request queue, and then continue to the next instruction.

  (When the global reply successfully returns, add the request to write to p to the end of the local request queue, and unblock line $f(p)$.)

  (If a reissue reply returns, add the request to synchronize s to the end of the shared or remote request queue.)

Note that the policy for global reads can be used for synchronized reads assuming that reissues are handled automatically and that cache lines being used for synchronization can be specially marked (if such marking is desired).

Shared. The policy for shared requests is as follows.

- **synchronize** s. Let p be the return address. Wait until line $f(s)$ is not blocked.

  If line $f(s)$ is marked as waiting for s, then the partner has already arrived and the cache line contains the return address p_1 of the partner. Add a reply destined for p_1 to the end of the (remote or private) reply queue. Add a reply destined for p to the end of the (remote or private) reply queue. Invalidate the cache line and continue to the next request. (The two replies are sent out a short time later.)

  Otherwise, the partner will arrive later. If the cache line is marked as waiting (for some other location s' with return address p'), add a "reissue" reply destined for p' to the end of the (remote or private) reply queue. This reply message indicates that the synchronize s' request on behalf of p' should be reissued. Else the cache line is simply discarded. In either case, place the return address p in the cache line and mark the line as waiting for s. Continue to the next request. (The reply, if any, is sent out a short time later.)

- **read** s. Wait until cache line $f(s)$ is not blocked.

  If cache line $f(s)$ contains the value of s, then add the reply containing the value of s to the remote or private reply queue, and continue to the next request.

  Otherwise the cache line does not contain s. If the cache line is marked as waiting (for some location s' with return address p'), add a reissue reply destined for p' to the

end of the (remote or private) reply queue. Else the cache line is simply discarded. In either case, mark the line as blocked for location s. Add the request to read s to the end of the local request queue, mark line $f(s)$ as blocked for s, and continue to the next request. (When the local reply returns, place the value returned into $f(s)$, add the reply containing the value to the remote or private reply queue, and unblock the line.)

- **write** s. Wait until cache line $f(s)$ is not blocked. If the cache line is marked as waiting (for some location s' with return address p'), add a reissue reply destined for p' in the cache line to the end of the (remote or private) reply queue. Else the cache line is simply discarded.

  In either case, place the value in line $f(s)$. Add the request to write the value to s to the end of the local request queue, and continue to the next request. (The shared memory location gets updated a short time later.)

5.4.2.2 Discussion

To accommodate these extensions to the policy in the cache, we need a fourth status type, **waiting**, for a shared cache line that is holding one of the two partners of a synchronization. In addition, it may be convenient to tag private cache lines that are being used for synchronization.

Given this support for pairwise synchronization, we can efficiently support the Asynchronous PRAM. Consider a sequence S of (normal) memory requests, followed by a synchronization step, followed by a sequence S' of (normal) memory requests. The processor can issue a synchronized read instruction, $p := \$s$, as soon as the only outstanding requests are to the same memory bank as s. While waiting for the synchronization to complete, the processor can perform local operations using the values prefetched prior to the synchronization and/or issue other synchronized read instructions.

We have not discussed acknowledgements of write operations. Acknowledgements can be used, for example, to inform the processor that all its memory requests have completed and thus it can issue a synchronization step. To accommodate acknowledgements, an extra field is needed in a write request to hold the return address. The acknowledgement can be sent as soon as the write request arrives at the shared request queue for its destination node. If a processor is running only a single program, it can maintain a count of the outstanding global read and write requests. The count is incremented whenever a shared request is made and decremented whenever a read reply or write acknowledgement is received. Alternatively, a test to see if all outstanding reads have completed can be made by performing an "or" of the appropriate status bit in all the cache lines.

Consider a request to synchronize s_n that arrives at the head of a shared request queue

only to find that cache line $f(s_n)$ is waiting for a second request to synchronize s_o, where $s_n \neq s_o$. In our policy, we place the new request s_n in the cache line and force the old request s_o to reissue. Note that this forced reissuing will not upset the assumed ordering of operations: a processor already must wait until a synchronization step completes before continuing with instructions that must follow the synchronization step.

We now compare this approach to collisions with three alternative approaches.

1. Place the new request s_n in the cache line and store the old request in memory location s_o. The advantages of this alternative are that a synchronization can complete as soon as the second partner arrives and there is no need to reissue requests on collisions.

   On the other hand, each request to synchronize s must check memory location s to see if its partner is in memory. This delays the mechanism even when there are no collisions. Each memory location used for synchronization must be tagged to distinguish a cell containing normal data from one containing the return address of a partner. This tagging may increase the memory line size if an address uses all the bits in a memory word.

2. Keep the old request s_o in the cache line and force the new request s_n to reissue. This eliminates the thrashing that can occur in our approach when requests arrive in alternating order, e.g. s_o, s_n, s_o, s_n. On the other hand, the new request s_n can be delayed for a long time if the partner of s_o is slow to arrive. In the worst case, a bug in another program running on the machine can tie up the cache line indefinitely, preventing the new request from ever succeeding.

3. Keep the old request s_o in the cache line and store the new request in memory location s_n. This has the drawbacks of both of the previous two approaches. Before the new request can be stored in s_n, memory location s_n must be fetched to see if the partner has already arrived. If the partner of s_o is slow to arrive, a cache line can be unavailable to other requests for a long time. This alternative is the most complicated of the three.

As indicated above, thrashing can occur in our approach when requests that map to the same cache line arrive in alternating order. We argue that this thrashing is unlikely to continue for long due to the variations in the round trip time for a reissue and the fact that a reissue does not begin until a new arrival occurs. This assumes that all four requests (two s_o and two s_n) are not issued by the same processor. Indeed, there is no need for a processor to synchronize with itself.

A possible optimization is to have a separate queue for synchronize requests that are bumped from the cache. This can save the overhead of the round trip for a reissue. To avoid the problem of continued thrashing discussed in the previous paragraph, the queue should not be used for the same request twice.

Our synchronization exchanges carry no data. This is because the first partner is storeed in the cache and we did not want to add an extra field to the cache line to hold any additional data. The "value" field in the cache holds the return address for the partner. We can, however, return to a processor the return address of its partner. In this way, each processor can learn its partner's id.

Throughout our discussion of the protocol, we have assumed no faults occur in the network. In practice, there will be a timeout mechanism for reissuing a global read request. The "requested address" field in the private cache line can be used to reissue the request.

We now turn to an example. Consider a processor wishing to synchronize both at shared memory location A and shared memory location B prior to reading shared location C. This can be accomplished by the following (compiler-generated) instructions: (a) $p := \$A$, (b) $q := \$B$, (c) $r0 := p$, (d) $r0 := q$, and (e) $x := C$, where p, q, and x are private locations and $r0$ is a register. First, a request will be issued to synchronize at A. While this is in progress, a request will be issued to synchronize at B. Then a private read of p will be issued. This instruction will not complete until the synchronization at A has successfully completed (since prior to that the line will be blocked). Finally a private read of q will be issued. Likewise, this will not complete until the synchronization at B has successfully completed. Thus both synchronizations will have completed by the time that instruction (e) is issued. Note that we did not have to test the value of p or q in order to know that the synchronization had completed (since if the line is unblocked, then the synchronization has completed). This assumes that pending synchronization reads map to unique cache lines. As discussed in the next paragraph, cache collisions can lead to "hidden" deadlock in certain anomalous cases.

We can support the full generality of synchronization exchanges discussed in section 4.2, including nonblocking exchanges as in the above example. However, special care must be taken so that cache collisions do not induce deadlock in a program. With nonblocking exchanges, the relative order of the synchronizations at A and B is unimportant. One processor may issue the synchronization at A then B, while another may issue the synchronization at B then A. The order is unimportant since both are issued before either must complete, and hence both can complete. However, if, for example, p and q above map to the same cache line, then the synchronization at A will block the cache line. This prevents the processor from issuing the synchronization at B until the synchronization at A completes. Thus we have an additional dependency between A and B that is not present if two different cache lines are used. With this added dependency, the relative order of the synchronizations at A and B is now important. Deadlock can arise in this scenario if one processor issues A first while the other first issues B. However, since p and q are temporary variables used only for the purpose of synchronization, they can readily be selected so as to map to distinct cache lines. Alternatively, the relative order of the two synchronizations can be selected so

as to avoid deadlock even in the presence of cache collisions, e.g. by having both processors issue the synchronization at A before the synchronization at B. Thus we suspect that this type of "hidden" deadlock due to cache collisions is not a problem in practice.

Chapter 6

Discussion and Related Work

6.1 Introduction

In the previous chapter, we claimed that the Asynchronous PRAM was a practical model for the following reasons: it supports an effective programming model for many application domains, it serves as a good basis for studying algorithms and complexity issues, and it can be implemented efficiently in hardware. In this chapter, we evaluate these claims and compare the Asynchronous PRAM with related models.

We begin with a discussion of the machines, programming audience, and application domains for which the model is well-suited. Section 6.2 discusses the advantages and disadvantages of semi-synchronous models. Section 6.3 evaluates other features of the Asynchronous PRAM model. We will limit our discussion to shared memory models and shared memory parallel computers. It is an important open question whether message passing or shared memory is the "better" model of processor communication for large-scale parallel computers. In section 6.4, we enumerate some of the differences between message passing and shared memory in terms of their effect on programming models, routing strategies, and architectural features.

6.1.1 Target machines

In section 1.2, we discussed four bottlenecks in large-scale parallel computing, namely, latency to global memory, contention in the network and the memory banks, synchronization overheads, and asynchronous communication. Based on the arguments presented there, we believe that the fastest general purpose shared memory parallel machines of the near future will be of the following type.

There are hundreds or more processors which communicate by reading and writing data words to a shared memory. The processors are (nearly) identical. The machine is asynchronous. Each processor has some fast, relatively small, private memory (e.g. registers,

cache). There is a large shared memory composed of a series of memory banks that are accessible by all processors via an interconnection network. Each processor has one such memory bank which is *local* to it, i.e. it can access the bank without going through the interconnection network. The network is (nearly) regular, i.e. it is (mostly) symmetrical with respect to the processors and memory banks. Special coprocessors take care of routing the memory requests. In order to overcome the high latency of accessing the shared memory, the parallel computer supports the pipelining of global memory accesses through the network. This implies that the interconnection network has sufficient bandwidth to handle the multiple requests per processor. Finally, the network will *not* contain features for combining messages (to avoid unneeded synchronization and/or complicated switches), but may provide support for synchronization.

6.1.2 Target programming audience

At the risk of oversimplification, programmers can be divided into three groups based on the effort they are willing to expend in order to achieve the fastest programs for their problems. The **casual programmer** wants to write programs that work with minimal programming effort. Programming is done at a high level using software packages (e.g. Mathematica [Wol88]) and/or library routines (e.g. Linpack [DBMS79]). The **informed programmer** is willing to spend considerably more time writing programs in order to improve their performance. When programming uniprocessor computers, for example, the informed programmer writes code in programming languages such as *C*, *Ada*, or *Pascal*. Finally, the **sophisticated programmer** is especially concerned with program performance and is willing to program at a very low level in order to achieve such performance. When programming uniprocessor computers, for example, the sophisticated programmer writes code tailored to the target computer, perhaps using the machine's assembly language.

The target audience for programming models based on the Asynchronous PRAM are the informed programmers. The goal is to provide the proper level of abstraction and cost measures to balance the informed programmer's desired programming effort with his or her performance expectations.

6.1.3 Target application domains

The model is targeted towards applications where nondeterminism is not needed. For example, any application suitable for a synchronous model is a good candidate for the Asynchronous PRAM. Problems from numerical computation and graph theory, for which PRAM algorithms exist, are well-suited to the model. In contrast, the Asynchronous PRAM is not suited for developing operating system routines where race conditions, arbitration, and other forms of nondeterminism are required.

The Asynchronous PRAM is well-suited to applications whose parallel programs exhibit frequent, data-dependent interprocessor communication, particularly when sufficient parallelism exists to run the program on a large scale parallel machine. Many applications fit in this framework.

6.2 The Case for Repeatable Programs

Given these assumptions about the target machines, programming audience, and application domains, we can proceed to evaluate the Asynchronous PRAM model.

This section discusses the advantages and disadvantages of the semi-synchronous approach to parallel programming, namely the restriction that programs be repeatable.

6.2.1 Reasons for more structured models

There is a natural tendency to dismiss any idea that places restrictions on programmers and/or appears to degrade the potential performance of programs. High level programming languages (even FORTRAN), structured programming, virtual memory, and so on, were each resisted for years. Eventually, however, the less structured approach is sometimes abandoned in favor of the more structured approach, typically for one or more of the following reasons.

- **Too difficult to use.** The less structured approach requires considerable programming effort to avoid making errors. For example, programmers tend to make many errors when writing programs in assembly code and/or with many "goto" statements. Another example is the popularity of structured communication paradigms such as *remote procedure call* [BN84][Gib87] as a less error-prone programming interface for communication between machines over a network. (In the remote procedure call paradigm, the communication between machines is accomplished by having a program on one machine call a procedure on another and wait for the return values.)

- **Too costly to support.** Supporting the more flexible approach in hardware and software can be costly. Reducing the number of options can reduce the cost of supporting the programming model. In the remote procedure call example, communication protocols can be simplified when intermachine communication is restricted to the single paradigm.

- **Too complex for optimizing software.** The less structured approach can thwart the efforts of optimizing compilers and other software. Existing optimizing compilers take advantage of the structured control flow of programs that are written using structured programming languages. In particular, structured control flow simplifies

the task of global data flow analysis. The performance and time saving benefits of optimizing software is lost when structured programming is not used.

- **Too little return for the effort.** The performance advantage of the less structured approach can be too small to justify the extra effort needed. For example, in most cases, programmer-controlled memory management yields minimal performance gain compared to operating systems-supported virtual memory.

The natural question that arises with regard to imposing more structure is why not give the programmer the choice of whether or not to be restricted? On the one hand, providing only the more structured model may simplify the hardware and software (see above). On the other hand, the necessary hardware may already exist in the machine and the less structured model may be needed for a few select parts of a program for which the performance is crucial. For example, a performance-critical section of a program otherwise written in a high level language may be written in assembly code.

6.2.2 A four point evaluation

We now evaluate repeatable and nonrepeatable programming in the context of the four criteria given above. As we shall see, many of the important questions remain unresolved.

Difficulty of use. As argued in detail in section 1.2.4, programs that exploit nonrepeatability are difficult to write, debug, analyze, and test. Errors can be extremely subtle, and not detectable by repeating the program. Moreover, debuggers can interfere with the timing programs. A debugger may mask certain errors by its very presence as an active agent in the computation, preventing certain timings that reveal a bug in the program. Parallel programming is already more difficult than sequential programming, so extra difficulties are not needed.

Semi-synchronous programming models, in contrast, eliminate what we claim is the most difficult aspect of parallel programming. Repeatability means the program can be viewed as having a single execution path for a given input. Hence, conceptually, the program is more straightforward and thus the programmer is less likely to make errors. Any bug can be recreated by rerunning the program. The presence or absence of the debugger does not affect the computation. However, extra programming effort may be required initially in order to create a program that is repeatable, just as extra programming effort is sometimes required to avoid using goto statements. For the problems in chapter 3, certain techniques were developed for writing repeatable programs that can be used for many other problems.

Cost to support. Programming models that permit nonrepeatability must support synchronization primitives that arbitrate between competing processors, and perhaps provide large queues to hold all the "losers" waiting for their next opportunity. Hardware support such as cache coherence mechanisms must be provided to keep the memory consis-

tent [DSB88]. Since arbitration typically involves concurrent access, expensive combining networks may be needed to achieve acceptable performance.

Semi-synchronous programming models, in contrast, can be supported by simpler hardware. Synchronization variables are used for ordering, not arbitration (since the order of competing accesses has been predetermined). Simple pairwise synchronization methods, as discussed in section 5.4, suffice since no hardware arbitration is needed. EREW semi-synchronous models can be efficiently supported on machines without combination hardware. Methods for synchronization barriers that do not need combination hardware, such as described in section 5.3, are also appropriate in this case. Since synchronization steps separate a sequence of competing accesses, each memory location has a single owner, or a set of nonconflicting owners, in the interval between two consecutive synchronization steps. This provides a simple means for cache coherence, as discussed in section 5.4. The programs in chapters 3 and 4, for example, can be efficiently supported by the scheme described in section 5.4. Sophisticated hardware coherence mechanisms are not needed since there is no arbitration of ownership to be resolved at run time.

On the other hand, hardware message-combining and cache coherence mechanisms may already exist in the machine and/or may be needed for programs from application domains that can not use the semi-synchronous model.

Complexity for software. Since programs with nondeterminism may have an exponential number of execution paths, software components may be unable to produce quality optimized code within an acceptable amount of time. Program testers must check all possible execution paths. Because programs are nonrepeatable and debuggers can affect the timing of events, traditional debugging techniques do not suffice. Existing debuggers for MIMD machines (e.g. [FLMC88][MC88]) provide an event logging mechanism that must be left on at all times, even after the program is considered to be debugged. The mechanism slows down all programs and can require a large amount of space for the log files.

In contrast, since a semi-synchronous program can be viewed as having a single execution path for a given input, the task of the optimizing compiler is greatly simplified. Instruction schedulers, for example, may be able to take advantage of the simplified control flow. (An *instruction scheduler* performs compile time reordering of machine level instructions for improved pipelining (e.g. [GM86]). Traditional debugging techniques can be used, and debuggers do not have to be left on at all times. However, a new tool is needed to detect competing accesses that have not been properly serialized, i.e. race conditions. These types of bugs must be eliminated first in order to use compiler optimization and debugger techniques that rely on repeatability. This paper does not address the design of such a tool.

More research needs to be done to study the extent to which compilers, debuggers, and cache mechanisms can exploit programs written in a semi-synchronous model.

Marginal return. This is the key open question: whether or not the performance

benefits of nonrepeatability warrant the effort. Considerable experimental research needs to be performed to test the performance implications of the semi-synchronous model. This paper provides no such data.

The performance of repeatable programs hinges on the ability of the programmer to select, in advance, a good ordering for competing communication events. For the problems studied in chapters 3 and 4, there were "natural" orderings that appear to be reasonable. Nevertheless, a nonrepeatable program has more freedom to adapt to the delays that occur during a particular run of the program.

6.2.3 Floating point computations and randomized programs

Difficulties with nonrepeatability are particularly acute for programs with floating point arithmetic. Different round-off effects can appear on two runs with the same input as a result of variations in the order of communication events. With repeatable programs, the final results (and all intermediate results) are the same whenever the input is the same.

On most machines, even programs that use randomization are repeatable in the semi-synchronous model. Randomization, on most machines, is generated using a pseudorandom number generator that produces a fixed sequence of "random" bits from an initial "seed". Each processor generates an independent sequence from its own particular seed (as opposed to a single sequence that all processors access for their random bits). Thus a semi-synchronous program is repeatable if each processor starts with the same seed as it used in the previous run. Of course, debugging a program with a fixed set of seeds does not yield a correct program, but at least a particular run that generated an error can be repeated.

6.3 Practical Evaluation of the Model

Among the variants of the Asynchronous PRAM model, we consider the variant with subset synchronization, non-unit time communication delay, and exclusive reading and writing to be the most practical for analyzing algorithms and programs, given our target machines, programming audience, and application domains. A programming model based on this variant of the Asynchronous PRAM includes "black box" primitives for the concurrent access of a memory location, in order to support a CRCW semi-synchronous model (defined in section 5.2.2). In addition, black box primitives are provided for parallel prefix, list ranking, and synchronization barriers.

In this section, we discuss design choices for programming models and models of computation, comparing the Asynchronous PRAM with related models.

6.3.1 Explicit parallelism

In the Asynchronous PRAM, parallelism is expressed explicitly by the programmer or algorithm designer. This contrasts with models in which programs are written in a sequential language and then compiled into code for a parallel machine (e.g. [AJ88][BCF+88]). These latter models permit programmers to avoid the complexities of parallel programming. Using these parallelizing compilers, existing uniprocessor software can be readily adapted to run on parallel machines.

Unfortunately, the state of the art in parallelizing compilers fails to produce high quality parallel code for many problems [AK85][AJ88]. Good parallel algorithms for a problem are often very different from known sequential algorithms. For example, the preorder numbering of trees is typically computed on a uniprocessor using depth first search [AHU83], but the fastest parallel algorithm for this problem uses list ranking and other techniques not used in the sequential algorithm [KR88]. The task of converting from a sequential algorithm to an entirely different parallel algorithm is too difficult for compilers, so explicit parallelism is needed to achieve better performance.

6.3.2 Word-level programming

In the Asynchronous PRAM, programs manipulate data words using unary or binary operations. Operations include reading a word from memory, writing a word to memory, and adding two words, where a *word* is a value that can fit in a single memory cell. A typical word size in existing computers is 32 bits. For theoretical models, a reasonable assumption is that a machine of p processors has a word size of $O(\log p)$ bits.

In contrast, in other models such as SISAL [LSF88] and the SCAN model [Ble87], programs manipulate sequences of values using functions that operate on sequences. These higher level models simplify the programming task, typically with a loss of performance.

6.3.3 Shared memory

The Asynchronous PRAM provides a shared memory model of processor communication instead of a message-passing model. Comparisons between the two will be discussed in detail in section 6.4.

6.3.4 Semi-synchronous

The issue of repeatability was discussed in section 6.2.

The semi-synchronous model also assumes that arbitrary delays are possible. For correctness in the model, the program must accommodate arbitrary delays in the completion of instructions. In practice, however, delays may be bounded by some fixed amount and/or distributed according to some known probability distribution. Models and algorithms can

be designed that take advantage of known properties of the delays. Some initial work along these lines is being done by Cole and Zajicek [CZ89].

6.3.5 Two-level memory

A **uniform shared memory model** presents a uniform programmer view of the shared memory, in which the access time to a shared memory location is (viewed as) independent of the processor making the request. A useful refinement is to distinguish between a processor accessing a local memory bank and a non-local bank, with the access time to the local bank being much smaller. We refer to this as a **two-level** shared memory model. In both models, the topology of the interconnection network is hidden from the programmer, as is the assignment of non-local data to memory banks. A **nonuniform shared memory model**, in contrast, exposes the fact that the access time to a shared memory location depends on the distance of the location from the requesting processor. Programmers are provided with an interconnection network graph, either the graph for the target machine or a graph that maps efficiently to the target machine [HRS88][KLM+89]. Moreover, programmers know and can control the mapping of the shared address space to memory banks. With this information, the programmer can compute the cost of any particular communication step. Communication primitives may be based on relative locality, e.g. each processor reads a location from its neighbor to the right in the mesh.

Certain problems, especially in numerical computation, lend themselves naturally to programs with interprocessor communication patterns that are highly structured, obey a strong locality, and can be determined in advance. The communication structure in such programs can often be mapped efficiently onto the interconnection graph of the target machine. In this case, the nonuniform shared memory model is well-suited for writing and analyzing programs that exploit locality.

In general, however, the "natural" communication patterns for a problem will not be so highly-structured. Thus, uniform or two-level shared memory programming models are typically easier to program than nonuniform models. Programmers prefer a uniform view of memory; they do not want the responsibility of keeping data values properly distributed among the memory banks throughout the course of a program (in order to minimize network and memory bank contention). If the natural communication patterns for a problem are highly irregular, it can be difficult for a programmer to exploit nonuniform access times. The model is too complicated and each access time is valid only if contention can be minimized. Another advantage of the uniform or two-level shared memory abstractions are that they are more universal since they hide the particular topology of the interconnection network. In addition, there is more flexibility in assigning processors to programs in the presence of faulty components and multiple users, since the correctness of a program is not based on a particular processor assignment, even after the computation has begun.

Uniform or two-level shared memory models can be effective in practice since

- contention at the memory bank level can be minimized by either interleaving or randomly hashing the address space,
- inefficiencies in communication can often be overcome by pipelining global memory accesses, and
- in many machines, the access time is rather uniform for all (processor, memory location) pairs.

This latter fact holds since the access time to a memory location is often dominated by software overheads and the x/k cycles needed to send an x-bit message across a k-bit wide link [Sei85].

The PRAM model provides a uniform shared memory model in which access to the shared memory takes unit time, the same as a local operation. Because of this, algorithms designed for the PRAM tend to be too fine-grained to run efficiently on real machines. This disadvantage of the PRAM can be partially overcome if the number of (virtual) processors in the program exceeds the number of machine processors on the target computer. Valiant [Val89] refers to this excess as the **parallel slackness** of an algorithm on a machine. If there are k PRAM processors per machine processor, then each processor can issue the k memory requests in pipelined fashion and thus proceed in a less fine-grained manner. The advantage to be gained through parallel slackness depends upon the rate at which memory accesses can be pipelined. For example, consider a machine in which a processor can complete a batch of k memory requests in $2l + f(k)$ steps where l is the (one-way, average) latency to non-local memory. If $f(k) = ck$ for some constant c, then the difference between a set of local references and a set of non-local references is at least a factor of c slower for the non-local case. In existing machines, c is quite large (e.g. 22 on the Stanford DASH machine [Gup89]), or worse still, $f(k)$ is not a linear function of k. Moreover, certain applications may have insufficient parallel slackness. For these reasons, the distinction between a local reference and a non-local reference is an important one.

Other models that do not distinguish between local and non-local references include the SCAN model [Ble87] and the Bulk-Synchrony model [Val89]. Models that present a two-level view of memory, besides the Asynchronous PRAM, include the LPRAM [AC88] and BPRAM [ACS89] models.

An important open question is whether hybrid uniform-nonuniform models can be effective. In these, the programmer can *selectively* control the mapping of the shared address space to memory banks for sections of the program that can exploit locality.

6.3.6 Explicit processor scheduling

Programming models can be classified according to the way in which processors are assigned work. In typical SIMD models, the single instruction dictates the assignment of processors to data items. In MIMD models, processors are either assigned work explicitly by the programmer or implicitly. In **explicit scheduling** models, such as the PRAM and the Asynchronous PRAM, the programmer explicitly schedules the work among a collection of virtual processors. In **implicit scheduling** models, such as the EPEX model [Dar87][DRGNP86], the programmer does not schedule the processors. Instead, the programmer labels instructions in the code that can be done in parallel, and relies on the compiler to add code for dynamically scheduling the processors during program execution. For programs with DO loops, the assignment of loop iterations to processors can either be done by the programmer or compiler (a **pre-scheduled loop**) or at run time (a **self-scheduled loop**).

Explicit scheduling models permit the programmer to be clever about how processors are scheduled. Moreover, the large overheads of run time scheduling are avoided. Implicit scheduling models, on the other hand, free the programmer from scheduling concerns and provide an easy way to write programs that automatically adjust to the number and relative speeds of the processors. Run time scheduling of processors is supported without operating system intervention through the use of global work queues or shared status variables. Upon completing its previous task, each processor accesses the appropriate work queue or status variable to get the next available task to be completed. For example, the status variable for a self-scheduled loop would indicate the first iteration of the loop not yet assigned to a processor. A processor that has finished its assigned work atomically updates this variable using a fetch-and-add primitive so as to obtain a unique iteration (or block of iterations) on which to work. However, considerable network traffic and contention ("hot spots") arise when many processors access the same work queue or status variable in order to be scheduled. A sophisticated combining network, with fetch-and-add hardware at each switch, is needed to avoid serious contention bottlenecks in programs with frequent run time scheduling [PN85].

Explicit scheduling models can do run time scheduling if the programmer puts instructions for load balancing into the program. Many PRAM algorithms, for example, use parallel prefix operations to assign consecutive numbers to the tasks remaining, so that the remaining tasks can then be evenly distributed among the processors (see [KR88]).

Since the target machines have no combining hardware and our goal is to have repeatable and easily analyzed programs, the Asynchronous PRAM provides explicit processor scheduling. The model provides a run time parameter, p, that indicates the number of processors assigned to the program. This parameter permits programs to be tailored to the number of machine processors assigned. For example, the best parallel algorithm to use for sorting n integers depends on the ratio of n to p [CS88]. Most of the algorithms in chapter

3, as well, consist of two or three stages in which the number of machine processors dictates the proper time to move from one stage to the next.

6.3.7 Arbitrary pipelining

The Asynchronous PRAM assumes that a batch of k requests to read any k shared memory locations completes in $2d+k-1$ time steps. In section 6.3.5, we argued that such a pipelining rate was unrealistic for existing machines. Indeed it is important to distinguish between local and non-local shared memory references for this very reason. On the other hand, adding another parameter, corresponding to the pipelining rate of the machine, would complicate the model considerably. Thus the compromise taken in the Asynchronous PRAM model is to provide a two-level view of memory and make an overly optimistic assumption on the pipelining rate, in order to avoid adding an additional parameter.

The LPRAM model [AC88] presents a two-level view of memory but does not permit pipelining. The follow-on model, the BPRAM [ACS89], permits pipelining, but only of contiguous blocks of memory. This matches the realities of certain machines in which an entire cache line or memory page can be transferred to cache or local memory as a unit. In addition, the model is well-suited for message-passing machines that use long messages.

Arbitrary pipelining is clearly more flexible than block pipelining. It is also easier to use, since the programmer does not have to ensure that values of interest are collected into contiguous locations. If arbitrary pipelining were permitted in the BPRAM, then a transpose of a $\sqrt{n}$ by $\sqrt{n}$ matrix, or any other permutation of n data items, could be done in $O(n/p+d)$ time on a model with p processors and communication delay d. This beats the BPRAM lower bound for this problem given in [ACS89]. Arbitrary pipelining is better suited to the task of simulating a program with parallel slackness, since for each machine processor, one step in the program can correspond to a batch of memory requests that are scattered throughout memory.

Block pipelining, however, has the practical advantage that a batch of values are guaranteed to return in order. This simplifies the bookkeeping and buffering tasks at the processor, and facilitates the use of vector coprocessors that may be residing in the same node as the processor. Moreover, only one instruction to access memory is outstanding at a time. Existing uniprocessors, which typically permit only a few read instructions to be outstanding at a time, can then be used as building blocks for parallel computers (e.g. in the RP3 [PBG+85]). Finally, block pipelining can reduce the size of programs, i.e. save code space, since a single instruction reads an entire block of memory.

A possible hybrid model, not studied to date, is to have block pipelining for read steps and arbitrary pipelining for write steps. Having block reads means that the locations read arrive in a fixed order and that existing uniprocessors can be used, while having arbitrary pipelining of writes means that values from a processor can be placed in different blocks of

memory, in anticipation of the next block read instructions. This simplifies programming, and permits any permutation of n data items to be performed in $O(n/p + d)$ time.

6.3.8 Limited concurrent access

Since the target machine does not have a combining network, simultaneous access to a memory location by many processors must be avoided. Thus the EREW Asynchronous PRAM is the most practical. However, concurrent reading and writing is convenient for programming, so the compromise is to provide library routines (i.e. black box primitives) with streamlined software emulation of these two primitives. A concurrent read or write is converted by the compiler or run time support to instructions to fanin and/or fanout to the desired location in a tree-like fashion.

6.3.9 Explicit cost measures

Throughout this paper, we have considered programming models and models of computation to be closely related. We believe that this view is appropriate for our target programming audience. A programming model for informed programmers must include a few select cost measures, so that algorithms can be analyzed and compared, and programmers can be guided to write faster programs without having to run the program first. These cost measures should accurately estimate the true performance, encouraging good programming practice for the parallel machine. Incorporating too many cost measures into a model results in a model that is complicated for programming and analysis, and either possesses too many parameters or is too machine specific and technology dependent.

Existing programming models such as the EPEX model have no explicit cost measures. Instead, programmers rely on run time performance monitors to get feedback on their program. Performance monitors are useful as a means for fine tuning a program, but have not yet proven to be as effective as programming using a model with explicit cost measures. In the sequential world, for example, a programmer must know the difference between an $O(n)$ and an $O(n^2)$ algorithm or suffer serious performance penalties. By the time the $O(n^2)$ subroutine in a very large software system is detected as a performance bottleneck by run time monitors, most if not all of the code likely has been written. By that time, it can be quite costly to substitute in the linear time algorithm, as the required changes may involve modifying widely used data structures.

On the other hand, the effectiveness of a model with a few simple cost measures is considerably reduced in the parallel computing world, due to the complexity of the computation itself and the degree of interference by other programs running on the same machine.

6.3.10 Synchronous cost measures

In the Asynchronous PRAM model, "unit time" is the same for all processors. If each processor executes a set of instructions that cost k and then participates in a synchronization barrier, the model charges k plus the cost for the barrier. While for *correctness*, the program must accommodate arbitrary delays in the completion of instructions, for *analysis*, the processors are viewed as executing in lock-step. Synchronous cost measures, as used in the Asynchronous PRAM, are best-suited to approximating machines where arbitrary delays occur, but (a) long delays are infrequent, and (b) they tend to be evenly distributed among the processors, especially when the processors all have similar programs. We expect this to be a reasonable approximation of the behavior of a machine with a symmetrical network and identical, tightly-coupled processors.

Other cost measures are possible. In the area of distributed systems, there is typically no notion of time. Instead the important parameter is the number of messages sent. Consider a chain of k messages, in which processor 1 sends a message to processor 2, which then sends a message to processor 3, and so forth, until finally processor k sends a message to processor $k+1$. In these models, a chain of k messages is not distinguished from k messages that can be sent in parallel. Thus the measure is not suitable for parallel computers. Recently, Awerbuch has introduced a model for distributed systems that distinguishes between chained messages and parallel messages [Awe87].

Another accounting scheme for asynchronous computation is due to Lynch and Fischer [LF81]. In this scheme, in one time unit or "round", each processor executes at least one instruction. As mentioned in section 2.3, this scheme forms the basis of the APRAM model [CZ89]. Round-based cost measures are suitable for nonrepeatable programs, even programs without explicit synchronization. However, they are more difficult to use than synchronous cost measures, and they do not account for the non-unit time latency to global memory.

6.4 Shared Memory vs. Message Passing

Thus far, we have limited our discussion to shared memory models, such as the Asynchronous PRAM, and shared memory parallel computers. In this section, we characterize the differences between the shared memory and the message passing models of processor communication. It has been observed by many (e.g. [AS88]) that each of these models can accommodate the other. However the emulations between the two are not practical: a machine streamlined for one model will not efficiently support the other model. Moreover, the emulations obscure the differences in the types of programming paradigms induced by each model. In what follows, we highlight seven important distinctions between shared memory and message passing.

Message semantics. In the shared memory model, messages have very simple semantics: a message is always directed to a shared memory bank, and can mean one of two things - either read (and return) the value of some memory cell or write a given value into some memory cell. In a general message passing model, messages can have arbitrarily complex semantics, and can directly invoke actions from various system components (not only memory banks). Special hardware might be required to handle (and interpret) messages efficiently. There is a clear relationship between hardware complexity at the receiving node and the complexity of the message semantics. The hardware mechanism for implementing only shared memory access can be quite simple.

Active vs. passive communication. In the shared memory model, the only way for a processor to obtain information on the actions or state of other processors is by reading a shared memory cell, i.e. it has to perform a global action. Thus when two processors want to communicate it requires two global actions and careful timing (write before read). Message passing permits a "passive" mode of communication - a processor can wait for other processors to send it information. This mode of communication enables, for example, implementing certain efficient synchronization schemes that cannot be performed on pure shared memory models, such as methods based on interrupts rather than polling.

Possibility of combining messages. Since shared memory messages are so limited, it is feasible to use a combining network to reduce network traffic and avoid certain serial bottlenecks. For arbitrary messages, however, combination seems much harder to implement. Furthermore, messages to the same processor in the message passing model are often unrelated and cannot be effectively combined.

Buffering requirements and switching strategies. Shared memory has short messages. General messages could be very long and require either large buffers or certain switching strategies such as circuit switching or wormhole routing ([AS88]). However, we should point out that some shared memory models, e.g. the BPRAM discussed in section 6.3.7, allow transfers of whole blocks of memory. Permitting block transfers as an atomic operation has strong implications on the machine architecture for issues such as combining and routing schemes.

Memory bottleneck. In the message passing model processor communication is separated from the the memory hierarchy, which is the usual bottleneck in traditional uniprocessor computers. Furthermore, less address bits need to be sent than in shared memory, since typically the number of processors is much smaller than the number of words in global memory. However, in message passing the receiving processor usually needs to get involved in storing the message (see next paragraph).

Processor participation in handling messages. A processor can be involved in several levels of parallel communication. The trend in current systems has been to relieve the processors from playing a role in the communication network (such as routing), while keeping

them involved in issuing and interpreting messages. In general, processor participation would be higher in message passing models than in shared memory ones. This is because processors need to respond to potentially complicated messages, and also might need to help in memory management (of their mailboxes).

Programming models supported. The two communication models induce different ways of thinking about solving problems, and therefore promote different programming models. For example, message passing can better support object-oriented models and paradigms in which processors distribute work to other processors. Shared memory, on the other hand, is more suitable for the PRAM model and its extensions. In shared memory the sender and receiver are decoupled in identity and timing, and communication is in the style of a bulletin board rather than personal mail.

In summary, even though shared memory and message passing can be viewed as computationally equivalent (in the sense of mutual emulation), they deserve distinction because of their different implications on programming models on one hand and machine architecture on the other.

Chapter 7

Conclusions

Parallel machines with hundreds to thousands to millions of processors are being proposed and developed as a vehicle for high performance computing. Four primary bottlenecks to effective use of large scale parallel computing are the latency to global memory, contention in the network and at the memory banks, synchronization overheads, and asynchronous communication. Based on these four, we believe the fastest (general purpose) shared memory parallel computers of the near future will be tightly-coupled, asynchronous machines with (nearly) identical processors and symmetrical networks. Each such machine will have a large shared memory comprised of a collection of memory banks that are accessible to all processors via the network. This network will provide sufficient bandwidth to support a high degree of pipelining of memory requests.

We have introduced the Asynchronous PRAM model of computation for the design and analysis of algorithms for such machines. The Asynchronous PRAM differs from the well-studied PRAM model in two important respects. First, Asynchronous PRAM processors run asynchronously and there is an explicit charge for synchronization. Second, there is a non-unit time cost to access the shared memory. The Asynchronous PRAM is one of the first attempts to design an asynchronous model suitable for parallel computers and study it in detail.

The Asynchronous PRAM model is based on two premises. For correctness, the program must accommodate arbitrary delays in the completion of instructions. For analysis, however, the processors approximate lock-step execution. We believe this is a reasonable model for designing algorithms for machines where arbitrary delays occur, but (a) long delays are infrequent, and (b) they tend to be evenly distributed among the processors, especially when the processors all have the same or similar programs. We expect this to be a reasonable approximation of the behavior of the machines described above.

An Asynchronous PRAM program is correct only if it works regardless of any run time delays that may occur. These delays are often too difficult to analyze by the programmer or compiler, and are best viewed as a form of indeterminacy in program execution. Sources

of delays in real machines include network congestion, memory bank contention, operating system interference, cache misses, and page faults. Because of the difficulties of writing correct programs in the presence of this indeterminacy, we impose a structured programming paradigm in which race conditions are eliminated. Any two competing references are serialized in a predetermined order (the order may, however, be data-dependent). This simplifies the use of our model, since each processor sees a deterministic view of the computation.

We have presented numerous algorithms, simulation results, and lower bounds for the Asynchronous PRAM model. The post office gossip game was introduced to study the relative power of various pairwise synchronization primitives. New techniques, not needed for synchronous models such as the PRAM, were developed for producing faster Asynchronous PRAM algorithms and reducing the number of processors needed. We believe the resulting algorithms are far more practical than algorithms developed for the PRAM.

We have provided evidence to support the practicality of the Asynchronous PRAM model. We introduced the notion of a semi-synchronous programming model, a model for repeatable asynchronous programs. Repeatable programs, in which the output and all intermediate results are the same every time the program is run on a particular input, greatly simplify the tasks of writing, debugging, analyzing, and testing programs. In addition, we presented methods for supporting the Asynchronous PRAM efficiently in hardware, including a cache protocol for the Asynchronous PRAM and a new technique for barrier synchronization.

This thesis has introduced a number of new models and measures for studying complexity issues in asynchronous parallel computation. The results presented here represent only a few of the many interesting questions that can be studied. Many open problems exist that may be easy to solve. Others will likely be more elusive. For example, algorithms can be designed for problems not addressed in this thesis (e.g. dynamic tree contraction [MR85]). It would be interesting to develop a fast sorting algorithm for the Asynchronous PRAM that is more practical than the one given in this thesis (perhaps based on the Cole parallel merge sort algorithm [Col88]). Many lower bounds remain open questions. For example, there is a gap between the upper and lower bounds for list ranking in many variants of the Asynchronous PRAM. Also, there is a gap for simulating the synchronous-LPRAM on the Asynchronous PRAM.

Many problems on the gossip game model are still to be resolved. It would be interesting to have stronger lower bounds for many of the gossip problems studied. Gossip game models can be used to study the relative power of various synchronization assumptions and primitives not addressed in this thesis, e.g. the relative power of one-way versus two-way synchronization. Whereas in *two-way* synchronization, each processor waits for the other to arrive at a synchronization point, in *one-way* synchronization, a "receiver" processor waits for a "sender" processor but not vice-versa.

We believe this work and other recent work on asynchronous models represent important first steps towards a richer understanding of asynchronous parallel computation. Understanding the computational power of various synchronization assumptions and primitives leads to an understanding of an important tradeoff in the design parallel machines: namely, how important are particular synchronization assumptions versus how much it costs (e.g. in dollars, memory access time, and machine cycle time) for the hardware to support these assumptions.

This work does not answer the question of whether or not a semi-synchronous model is practical. We have discussed its advantages, but we have no experimental data to reveal how it compares in practice. This work is entirely a paper design. Few existing machines, if any, match the description of our target machine. The hardware mechanisms described have not been built. No compilers have been written.

More work is needed in the areas discussed in sections 6.2 and 6.3. We highlight two of these. First, work is needed to develop an appropriate means of enforcing the semi-synchronous model in programs and testing for violations of the model. This enforcement and/or testing may be accomplished through some combination of programming language constructs that forbid violations, compilers that detect potential violations, and run time mechanisms that signal when a violation has occurred. Second, more work is needed to study the extent to which compilers, debuggers, and cache mechanisms can exploit programs written in a semi-synchronous programming model.

There are many difficulties to overcome before large scale parallel computers can be used effectively for high performance computing. In these early stages of the field, with relatively few machines built and/or programs written, many questions are not easily resolved. We believe that research that addresses algorithms, software, and hardware together, as has been attempted in this thesis, can best help accomplish this goal.

Bibliography

[AC88] A. Aggarwal and A. K. Chandra. Communication complexity of PRAMs. In T. Lepistö and A. Salomaa, editors, *Automata, Languages and Programing, Lecture Notes in Computer Science, Vol. 317*, pages 1–18. Springer-Verlag, Berlin, July 1988. Proc. 15th ICALP.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.

[ACS89] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. In *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 11–21, Santa Fe, New Mexico, June 1989.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[AI88] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. In R. Dierstein, D. Müller-Wichards, and H.-M. Wacker, editors, *Parallel Computing in Science and Engineering, Lecture Notes in Computer Science, Vol. 295*, pages 61–88. Springer-Verlag, Berlin, June 1988. Proc. 4th International DFVLR Seminar on Foundations of Engineering Sciences.

[AJ88] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proc. 1988 ACM Conf. on Programming Language Design and Implementation*, pages 241–249, Atlanta, Georgia, June 1988.

[AK85] J. R. Allen and K. Kennedy. A parallel programming environment. *IEEE Software*, 2(4):21–29, 1985.

[AKS83] M. Ajtai, J. Komlos, and E. Szemeredi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[AM88] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. In J. H. Reif, editor, *VLSI Algorithms and Architectures, Lecture Notes in Computer*

Science, Vol. 319, pages 81–90. Springer-Verlag, Berlin, June 1988. Proc. 3rd Aegean Workshop on Computing (AWOC).

[AS88] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–24, 1988.

[Awe87] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting leader election and related problems. In *Proc. 19th ACM Symp. on Theory of Computing (STOC)*, pages 230–240, New York, New York, May 1987.

[Axe86] T. S. Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Computing*, 3(2):129–140, 1986.

[BBN86] BBN Laboratories. Butterfly-TM parallel processor overview. Technical Report 6149, version 2, BBN, Cambridge, Massachusetts, June 1986.

[BCF+88] M. Burke, R. Cytron, J. Ferrante, W. Hsieh, V. Sarker, and D. Shields. Automatic dicovery of parallelism: A tool and an experiment. In *Proc. 1988 ACM Symp. on Parallel Programming (PPEALS)*, pages 77–84, New Haven, Connecticut, July 1988. *SIGPLAN Notices*, 23(9), September 1988.

[BGSS89] Y. Birk, P. B. Gibbons, J. L. C. Sanz, and D. Soroker. A simple mechanism for efficient barrier synchronization in MIMD machines. Technical Report RJ 7078, IBM Almaden Research Center, San Jose, California, October 1989.

[BH85] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–145, 1985.

[Ble87] G. Blelloch. Scans as primitive parallel operations. In *Proc. 1987 International Conf. on Parallel Processing (ICPP)*, pages 355–362, August 1987.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1):39–59, 1984.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–208, 1974.

[Bro86] E. D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[BS72] B. Baker and R. Shostak. Gossips and telephones. *Discrete Mathematics*, 2:191–193, 1972.

[CF78] L. M. Censier and P. Feautrier. A new solution to the coherence problems in multicache systems. *IEEE Trans. on Computers*, C-27(12):1112–1118, 1978.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[CS88] R. Cypher and J. L. C. Sanz. Cubesort: An optimal sorting algorithm for feasible parallel computers. In J. H. Reif, editor, *VLSI Algorithms and Architectures, Lecture Notes in Computer Science, Vol. 319*, pages 456–464. Springer-Verlag, Berlin, June 1988. Proc. 3rd Aegean Workshop on Computing (AWOC).

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[CV88] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17(1):128–142, 1988.

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. 19th ACM Symp. on Theory of Computing (STOC)*, pages 1–6, New York, New York, May 1987.

[Cyp88] R. Cypher. personal communication, 1988.

[CZ89] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 169–178, Santa Fe, New Mexico, June 1989.

[Dar87] F. Darema. Applications environment for the IBM research parallel prototype (RP3). Technical Report RC 12627, IBM T. J. Watson Research Center, Yorktown Heights, New York, March 1987.

[DBMS79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *Linpack Users' Guide.* SIAM Press, Philadelphia, Pennsylvania, 1979.

[Din89] A. Dinning. A survey of synchronization methods for parallel computers. *IEEE Computer*, 22(7):66–77, 1989.

[DRGNP86] F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. Technical Report RC 11552, IBM T. J. Watson Research Center, Yorktown Heights, New York, November 1986.

[DSB88] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, 1988.

[Eck79] D. M. Eckstein. Simultaneous memory access. Technical Report TR-79-6, Computer Science Dept., Iowa State University, Ames, Iowa, 1979.

[EM89] S. Even and B. Monien. On the number of rounds necessary to disseminate information. In *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 318–327, Santa Fe, New Mexico, June 1989.

[FLMC88] R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In *Proc. 1988 ACM Workshop on Parallel and Distributed Debugging*, pages 163–173, Madison, Wisconsin, May 1988. *SIGPLAN Notices*, 24(1), January 1989.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing (STOC)*, pages 114–118, San Diego, California, May 1978.

[Gaj83] D. Gajski. Cedar - a large scale multiprocessor. In *Proc. 1983 International Conf. on Parallel Processing (ICPP)*, pages 524–529, August 1983.

[GGK+83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Trans. on Computers*, C-32(2):175–189, 1983.

[Gib87] P. B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Trans. on Software Engineering*, SE-13(1):77–87, 1987.

[Gib88] P. B. Gibbons. Towards better shared memory programming models. In *Proc. 1988 IBM-NSF Workshop on Opportunities and Constraints in Parallel Computing*, San Jose, California, December 1988.

[Gib89] P. B. Gibbons. A more practical PRAM model. In *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 158–168, Santa Fe, New Mexico, June 1989.

[GLKA84] J. W. Goodman, F. J. Leonberger, S. Y. Kung, and R. A. Athale. Optical interconnections for VLSI systems. *Proc. of the IEEE*, 72(7):850–866, 1984.

[GLO88] A. G. Greenberg, B. D. Lubachevsky, and A. M. Odlyzko. Simple, efficient, asynchronous parallel algorithms for maximization. *ACM Trans. on Programming Languages and Systems*, 10(2):313–337, 1988.

[GLR83] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. on Programming Languages and Systems*, 5(2):164–189, 1983.

[GM86] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. 1986 ACM Symp. on Compiler Construction*, pages 11–16, Palo Alto, California, June 1986. *SIGPLAN Notices*, 21(7), July 1986.

[GR84] D. B. Gannon and J. Van Rosendale. On the impact of communication complexity on the design of parallel numerical algorithms. *IEEE Trans. on Computers*, C-33(12):1180–1194, 1984.

[Gup89] A. Gupta. The Stanford DASH multiprocessor, personal communication, 1989.

[GVW89] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. 3rd International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–73, Boston, Massachusetts, April 1989. *SIGARCH Computer Architecture News*, 17(2), April 1989.

[Hil85] W. D. Hillis. *The Connection machine.* The MIT Press, Cambridge, Massachusetts, 1985.

[HKP84] H. J. Hoover, M. M. Klawe, and N. J. Pippenger. Bounding fan-out in logical networks. *Journal of the ACM*, 31(1):13–18, 1984.

[HRS88] L. S. Heath, A. L. Rosenberg, and B. T. Smith. The physical mapping problem for parallel architectures. *Journal of the ACM*, 35(3):603–634, 1988.

[Jay88] D. N. Jayasimha. Distributed synchronizers. In *Proc. 1988 International Conf. on Parallel Processing (ICPP)*, pages 23–27, August 1988.

[Jor85] H. F. Jordan. HEP architecture: programming and performance. In J.S. Kowali, editor, *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, pages 1–40. MIT Press, Cambridge, Massachusetts, 1985.

[Kar87] A. H. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–57, 1987.

[Kar88] R. M. Karp. personal communication, 1988.

[KLM+89] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg. Work-preserving emulations of fixed-connection networks. In *Proc. 21st ACM Symp. on Theory of Computing (STOC)*, pages 227–240, Seattle, Washington, May 1989.

[KM69] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[KR88] R. M. Karp and V. L. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB-CSD-88-408, Computer Science Division, University of California at Berkeley, Berkeley, California, March 1988. To appear in *Handbook of Theoretical Computer Science*, North-Holland, Amsterdam, 1990.

[KRS88] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. Technical Report RC 13572, IBM T. J. Watson Research Center, Yorktown Heights, New York, March 1988.

[KU88] A. Karlin and E. Upfal. Parallel hashing - an efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, 1988.

[Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.

[LF81] N. A. Lynch and M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13(1):17–43, 1981.

[LM86] B. D. Lubachevsky and D. Mitra. A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. *Journal of the ACM*, 33(1):130–150, 1986.

[LM89] T. Leighton and B. Maggs. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *Proc. 30th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 384–389, Research Triangle, North Carolina, October 1989.

[LMR88] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proc. 29th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 256–269, White Plains, New York, October 1988.

[LSB88] T. J. LeBlanc, M. L. Scott, and C. M. Brown. Large-scale parallel programming: Experience with the BBN Butterfly parallel processor. In *Proc. 1988*

ACM Symp. on Parallel Programming (PPEALS), pages 161–172, New Haven, Connecticut, July 1988. *SIGPLAN Notices*, 23(9), September 1988.

[LSF88] C.-C. Lee, S. Skedzielewski, and J. Feo. On the implementation of applicative languages on shared-memory, MIMD multiprocessors. In *Proc. 1988 ACM Symp. on Parallel Programming (PPEALS)*, pages 188–197, New Haven, Connecticut, July 1988. *SIGPLAN Notices*, 23(9), September 1988.

[MC88] B. P. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. In *Proc. 1988 ACM Workshop on Parallel and Distributed Debugging*, pages 141–150, Madison, Wisconsin, May 1988. *SIGPLAN Notices*, 24(1), January 1989.

[McA89] A. D. McAulay. Conjugate gradients on optical crossbar interconnected multiprocessor. *Journal of Parallel and Distributed Computing*, 6(1):136–150, 1989.

[MPS89] C. Martel, A. Park, and R. Subramonian. Optimal asynchronous algorithms for shared memory parallel computers. Technical Report CSE-89-8, Division of Computer Science, University of California, Davis, California, July 1989.

[MR85] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In *Proc. 26th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 478–489, Portland, Oregon, October 1985.

[MRK88] G. L. Miller, V. L. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal on Computing*, 17(4):687–695, 1988.

[MSGR61] R. E. Miller, R. E. Swartwout, D. B. Gillies, and J. E. Robertson. Introduction to speed independent circuits. In *Proc. 2nd IEEE Symp. on Switching Circuit Theory and Logical Design*, pages 87–110, Detroit, Michigan, October 1961.

[NA88] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? Technical Report CSG Memo 292, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1988.

[NS81] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C-30(2):101–106, 1981.

[NS89] J. Y. Ngai and C. L. Seitz. A framework for adaptive routing in multicomputer networks. In *Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, Santa Fe, New Mexico, June 1989.

[Pat85] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, 1985.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research parallel processor prototype (RP3): Introduction and architecture. In *Proc. 1985 International Conf. on Parallel Processing (ICPP)*, pages 764–771, August 1985.

[Pip79] N. Pippenger. On simultaneous resource bounds. In *Proc. 20th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 307–311, San Juan, Puerto Rico, October 1979.

[PN85] G. F. Pfister and V. A. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Trans. on Computers*, C-34(10):943–948, 1985.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communication of the ACM*, 29(12):1184–1201, 1986.

[PY88] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proc. 20th ACM Symp. on Theory of Computing (STOC)*, pages 510–513, Chicago, Illinois, May 1988.

[Ran87] A. G. Ranade. How to emulate shared memory. In *Proc. 28th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 185–194, Los Angeles, California, October 1987.

[Ran89] A. G. Ranade. *Fluent parallel computation.* PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, May 1989.

[RBJ88] A. G. Ranade, S. N. Bhatt, and S. L. Johnson. The Fluent abstract machine. In *Proc. 5th MIT Conf. on Advanced Research in VLSI*, pages 71–94, Cambridge, Massachusetts, March 1988.

[RV87] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, 1987.

[SD88] C. Scheurich and M. Dubois. Concurrent miss resolution in multiprocessor caches. In *Proc. 1988 International Conf. on Parallel Processing (ICPP)*, pages 118–125, August 1988.

[Sei85] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.

[Seq86] Sequent. Balance technical summary. Technical report, Sequent Computer Systems Inc., Burlington, Massachusetts, November 1986.

[Sha89] A. Shamir. personal communication through R. M. Karp, 1989.

[Sni88] M. Snir. personal communication, 1988.

[Sny84] L. Snyder. Parallel programming and the Poker programming environment. *IEEE Computer*, 17(7):27–36, 1984.

[TV85] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[Ull84] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Maryland, 1984.

[Upf89] E. Upfal. An $O(\log n)$ deterministic packet routing scheme. In *Proc. 21st ACM Symp. on Theory of Computing (STOC)*, pages 241–250, Seattle, Washington, May 1989.

[Val89] L. G. Valiant. Bulk-synchronous parallel computers. Technical Report TR-08-89, Harvard University, Cambridge, Massachusetts, April 1989.

[VB81] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proc. 13th ACM Symp. on Theory of Computing (STOC)*, pages 263–277, Milwaukee, Wisconsin, May 1981.

[WF84] C. Wu and T. Feng. *Tutorial: Interconnection Networks for Parallel and Distributed Processing.* IEEE Computer Society Press, Washington, D.C., 1984.

[Wol88] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, Reading, Massachusetts, 1988.