A More Practical PRAM Model

Phillip B. Gibbons
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

This paper introduces the Asynchronous PRAM model of computation, a variant of the PRAM in which the processors run asynchronously and there is an explicit charge for synchronization. A family of Asynchronous PRAM's are defined, varying in the types of synchronization steps permitted and the costs for accessing the shared memory. Algorithms, lower bounds, and simulation results are presented for an interesting member of the family.

1 Introduction

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 global memory. The processors execute in lockstep, 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 global memory location into a local memory location. In a compute step, each processor can execute a single RAM operation whose operands are in local memory, storing the result in a local memory location. In a write step, each processor can write the contents of one local memory location into a global 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: its shared memory abstraction hides the details of the interprocessor communication and synchronization.

This work was supported in part by the International Computer Science Institute, Berkeley, CA and by the IBM Almaden Research Center, San Jose, CA.

There are several difficulties that arise in mapping PRAM algorithms onto existing shared memory MIMD machines, such as the Sequent Balance, the BBN Butterfly, the NYU Ultracomputer, and the IBM RP3. First, realistic MIMD machines have more limited communication capabilities than the PRAM. The PRAM assumes that each processor can access any memory location in one step. Realistic machines are more limited in at least three respects:

- the shared memory locations are partitioned into a smaller number of memory banks, each of which can support only a constant number of accesses per cycle,
- the processors are connected to these memory banks by an interconnection network in which the shortest path between a processor and most locations requires many hops through the network, and
- a link in the network can transmit only one message per cycle, and thus messages competing for a link are serialized.

Considerable research effort has been focused on finding efficient ways to satisfy the simultaneous global memory accesses of a PRAM read or write step on realistic networks (e.g. [KU86][Ran87]). Of course, no simulation effort can overcome the fact that many global memory accesses must traverse the full diameter of the network. Indeed, most global memory accesses will take considerably longer to complete than a local operation. Recently, several researchers have explored variants of the PRAM that account for this *communication delay* (e.g. [ACS89][PY88]).

Second, existing MIMD machines are asynchronous. The PRAM assumes that the processors execute in lockstep. In order to effectively support a large number of processors, multiple users, and multiple instruction streams, realistic MIMD machines must permit each processor to execute its instructions independently of the timing of the other processors. These asynchronous machines have an advantage over synchronous machines in that they avoid making worst case assumptions on

the indeterminacy of instruction completion times (including worst case assumptions on clock skew). This indeterminacy arises in real machines 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 indeterminacy arises due to the relative speeds of instruction execution: an add is much faster than a floating point multiply or a global memory access.

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], each switch holds any global memory accesses for an instruction until all global 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, this paper introduces the Asynchronous PRAM model, a variant of the PRAM model more suited to shared memory MIMD machines. The paper is organized as follows. Section 2 defines the Asynchronous PRAM model and compares it with related models. Section 3 focuses in on the Phase PRAM, an Asynchronous PRAM in which all the processors synchronize at each synchronization step. Sections 4, 5, and 6 present new algorithms, simulation results, and lower bounds for the Phase PRAM. Finally, section 7 presents some concluding remarks and directions for future work.

2 The Asynchronous PRAM

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 global 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. 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, delay.

There are four types of instructions:

- global reads: reading the contents of a shared memory location into a local memory location,
- local operations: any RAM operation where the operands are in local memory and the result is stored in local memory,
- global writes: writing the contents of a local memory cell into a global memory cell, and
- synchronization steps.

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.

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. Thus there are no race conditions possible in the model (except in Asynchronous PRAM's that permit "arbitrary" concurrent write: see section 3).

Remark: In the PRAM model, a processor at step i can use the fact that all processors have completed step $i-1$. For example, a processor can read a memory location at step i that some other processor may have written at step $i-1$. In the Asynchronous PRAM, on the other hand, there is no global clock and arbitrary delays are possible, thus a processor at its instruction i does *not* automatically know the progress of the other processors. In order to be sure that all accesses to a global location at a point in the computation have completed, the processor must first synchronize with all other processors that might be accessing the location (see, e.g., the multiprefix problem of section 4).

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, some 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 analyze 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.

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. Consider a phase that is followed by a synchronization step among a set S of processors. The completion time for the phase (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 of the synchronization step itself. The cost of a synchronization step is $B(p)$, a nondecreasing function of p, where $p = |S|$. The running time for a program is defined to be the maximum over all processors of the completion time for its local program.

Remark: There are a wide variety of schemes for implementing synchronization steps on MIMD machines, with varying runtime overheads. Thus the estimated "cost" for a synchronization step is not some fixed value in our model, but a parameter. This permits algorithms to be designed that are more machine independent. Given a target machine, an algorithm can then be tailored to the machine by specifying an appropriate function for $B(p)$.

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. An Asynchronous PRAM with *all-processor synchronization* restricts synchronization steps to only those that include all the processors. Multiple, independent 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 paper, 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.

In all cases, for the purpose of analysis, we will assume that a processor can pipeline its instructions: it may issue instructions $i+1, i+2$, etc. 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. For example, the pipelining of global reads within a phase is limited only by the local dependencies of one read upon another: there are no interprocessor dependencies since no other processor can write into any of the locations being read. Local dependencies can occur between 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 is the case when traversing a linked list. A sequence of r global read instructions with no interdependencies, issued one after another by a processor, completes in $2d + r - 1$ time. Likewise, w write instructions issued one after another by a processor complete in $d + w - 1$ time.

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.

Both the communication delay d and the synchronization cost B are nondecreasing functions of the number of processors. The appropriate functions to use depend on the parallel machine on which the program is to run. A typical function for d is the diameter of the interconnection network, e.g. $\log p$ or $\sqrt{p}$; a typical function for B is d or $d \log p$. In designing algorithms for the Asynchronous PRAM model, we assume only that $1 \leq d \leq B \leq p$.

Although only a simple variant of the PRAM, the Asynchronous PRAM is considerably more practical. Its primary advantages are that it permits asynchronous execution of processors 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.

2.3 Comparison with related models

Most models of parallel computation studied to date have been synchronous (e.g. the PRAM, the LPRAM [ACS89]). There are asynchronous models in the world of distributed computing, but in these models, the parameters are typically the number of processes and the number of messages. Typical programming models for MIMD machines (e.g. for the IBM RP3 and the Sequent) have no notion of costs.

Recently, Cole and Zajicek have been studying a model they call the APRAM[CZ89]. The APRAM is based on an accounting scheme for asynchronous computation due to Lynch and Fisher[LF81]. In this scheme, in one time unit or "round", each processor executes at least one instruction (see also [KRS88]). 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.

Another related model 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, unlike in the message-passing model.

3 The Phase PRAM

In this section and the sections that follow, we will focus on Asynchronous PRAM's with *all-processor synchronization*. A computation is a series of global, program-wide phases in which the processors run asynchronously, separated by synchronization steps that are among all the processors. We will refer to such Asynchronous PRAM's as either *Phase PRAM*'s or *Phase LPRAM*'s. 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 a sequence of k global read (write) instructions with no interdependencies, issued one after another by a processor, completes in $2d + k - 1$ time ($d + k - 1$ time).

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 of the cost of the instructions executed by a processor during the phase. The running time for a program is simply the sum of the time costs for each phase plus B times the number of synchronization steps.

Various concurrent read/write policies are possible:

- EREW Phase PRAM or Phase LPRAM: no two processors access the same location in a phase,
- 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, and
- 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, either (a) the processors must all write the same value (common-CRCW) or (b) an arbitrary processor succeeds in writing the final value at the location (arbitrary-CRCW).

Let us turn to an example.

Prefix sum:
We consider the first half of the prefix sum computation in which we compute the sum of the n input numbers. We can compute the sum on an EREW PRAM in $O(\log_2 n)$ time, in binary tree fashion. By inserting a synchronization step after each PRAM read or write step, we get an EREW Phase PRAM or Phase LPRAM algorithm that runs in $O((B+d)\log_2 n)$ time, i.e. $O(B \log n)$ 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 values, computes their sum, writes the result, and then synchronizes. Since the reads can be pipelined, each level of the tree takes $(2d + B - 1) + (B - 1) + d + B$ time, i.e. $O(B)$ time, and there are $\log_B n$ levels.

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 parallel prefix problem.

This motivates a few simple lemmas.

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

The idea is to have each Phase PRAM or Phase LPRAM processor simulate B PRAM processors for a single step and then synchronize. This balances the time spent synchronizing with the time spent accessing memory and computing. The goal in designing algorithms for the Phase PRAM or Phase LPRAM is to beat these immediate time and processor bounds.

Lemma 2 *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 $O((p_0/p)t + B(p)s)$.*

For each phase, each processor in the smaller machine simulates the instructions in the phase for p_0/p processors of the larger machine and then synchronizes.

A more general result, corresponding to Brent's scheduling principle for the PRAM[Bre74], is as follows. Define the *work* of an algorithm to be the sum over all processors of the number of instructions performed by a processor in the algorithm, *not* counting synchronization steps.

Lemma 3 *A Phase PRAM 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 using p processors in time $O(x/p + t + B(p)s)$.*

The proof is given in section 5. As in Brent's scheduling principle, this lemma does not account for scheduling costs.

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

Lemma 4 *An Asynchronous PRAM program can be simulated with constant overhead by an Asynchronous PRAM 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. We call such reads *oblivious*. A *locally oblivious* algorithm or program is one in which, in each phase, each processor first issues a sequence of *oblivious* global reads, then issues a sequence of local computes, and finally issues a sequence of global writes.

Lemma 5 *A locally 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 r (oblivious) global reads, l local operations, and 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 locally oblivious, and so the results will apply to the more realistic Phase LPRAM.

Remark: Similar arguments show that upper bounds for locally oblivious Phase PRAM's yield upper bounds for the LPRAM model and the model in [PY88], by setting $B = d$.

4 Improved 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.

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 1 *Given n numbers, stored one per global 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 local cell and G is a global 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: 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 $n/(t_1 t_2 \cdots t_s) \leq 1$. By an easy induction proof or calculus argument, 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 to produce the sum is $t + Bs$. Let $s = \alpha \log n / \log B$. If $\alpha \geq 1$, then the time is at least $Bs \geq B \log n / \log B$. If $\alpha < 1$, then the time is at least $t \geq s n^{1/s} \geq \alpha B^{1/\alpha} \log n / \log B > B \log n / \log B$. □

This argument can be applied to any n input, 1 output associative function f, where (1) the output 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.

4.1 List ranking

Now consider the list ranking problem. Unlike the parallel 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:

(1) For each of $\log n / \log B$ rounds, repeat steps (2)-(4):

(2) Processor i makes B copies in global memory of 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 global memory the new successor of i, i.e. the B^{th} such element visited, then synchronizes.

This list-reducing procedure 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 1, 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 a locally 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. [CV86b][CV86a][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 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) The second stage reduces the list from pB elements to p elements. We use the deterministic coin tossing technique of Cole and Vishkin[CV86b], 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.

These ideas lead to an 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[CV86b], 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 ranges of B, e.g. $B \leq \log p$, the second stage can be omitted.

4.2 FFT and bitonic merge

In [PY88], an algorithm for FFT is given that runs in $O(d\log n/\log d)$ time using $n\log d/d$ processors on a model with communication delay d. The same approach leads to an $O(B\log n/\log B)$ time locally oblivious algorithm on an EREW Phase PRAM using $n\log B/B$ processors. We can improve upon this result in the related problem of bitonic merge as follows. The directed acyclic graph for a bitonic merge computation is a butterfly graph of n rows and $\log n$ columns. We 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. This leads to the following locally oblivious algorithm, which runs in $O(B\log n/\log B)$ time on an EREW Phase PRAM using only n/B processors:

Efficient Bitonic Merge algorithm:

(1) Let the n elements in the bitonic sequence be stored in a block of global memory locations. For each of $\log n/\log B$ stages in turn, do step (2).

(2) For stage j, each processor i reads the B elements corresponding to the "rows" for set $S_{j,i}$ from the global memory, sorts them locally using a sequential algorithm, synchronizes, writes back its B elements sorted order, and synchronizes. The sorting can be done in $O(B)$ time since the B elements in a set form a bitonic sequence.

Remark: By lemma 5, the running time for a Phase LPRAM with n/B processors is also $O(B\log n/\log B)$.

This is an example of a general paradigm for saving processors 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.

4.3 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 parallel prefix computation for each label. Using the parallel prefix algorithm of section 3, Ln/τ processors suffice to achieve $O(\tau) = O(B\log n/\log B)$ time. However, we can improve on this result for the case where $L > \tau$ 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 $p = n$ processors.

Multiprefix algorithm:

(1) Consider a forest of (implicit) complete B-ary trees of p leaves each. We will have L trees in all, one for each label. Processor i starts at leaf i of the tree for its label. As in the parallel prefix algorithm, we will compute a sum by progressing level-by-level up this tree. 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 elements with label j.

We repeat steps (2)-(3) once for each level in the forest, 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.

This algorithm produces the sum totals for each label; reversing the process yields the partial sums. This locally oblivious algorithm runs in $O(B \log n / \log B)$ time on an EREW Phase PRAM 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.

An algorithm for multiprefix yields an algorithm for sorting n integers of $k \log n$ bits each[Ran87]. It runs in $O(kB \log n / \log B)$ time on a locally 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.

5 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, $n = \sum_{i=1}^{k} h_i$, and $h = \max(h_1, \ldots, h_k)$. A lower bound on the completion time is $\max(n/p, h, B)$.

There is a well known strategy for the case relevant to the PRAM (B = 0), which achieves within a factor of 2 of optimal. In particular, this strategy, known as Brent's scheduling principle[Bre74], achieves $n/p + h$ completion time. If the durations of the jobs are known (and sorted: $h_1 \geq h_2 \geq \cdots \geq h_k$), then the following *nonpreemptive* schedule achieves $n/p+h+B$ completion time: assign processor i to complete jobs $i, i+p, i+2p$, etc.

Lemma 6 *The above schedule achieves $n/p + h + B$ completion time.*

Proof: Let x_j be the number of jobs with at least j steps. Processor 1 has the most work and it does $\lceil x_j/p \rceil$ unit-time steps which are the j^{th} step of some job. Thus the completion time is $\sum_{j=1}^{h} \lceil x_j/p \rceil + B \leq n/p + h + B$. □

Remark: Lemma 3 of section 3 follows from lemma 6.

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. 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:

(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.

Theorem 2 *The Balanced algorithm achieves an $O(\max(n/p, h, B) \log B / \log \log B)$ completion time.*

Proof: (sketch) 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. Each round is completed in $2B$ time (counting the B for rescheduling). Let k_i be the number of live lists per processor at the start of round i, $1 < k_i < B$. 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. The longest list is at least $B \sum_{i=1}^{m} 1/k_i$. The completion

time is $2mB$. The worst case ratio of the completion time to the optimal time (i.e. $\max(n/p, h, B)$) is $r \leq \max_{k_i,m}(\min(2mB/(B\sum k_{i+1}/k_i), 2mB/(B\sum 1/k_i), 2mB/B)) = \max_{k_i,m}(2m\min(1/(\sum k_{i+1}/k_i), 1))$, where $k_1 \geq k_2 \geq \cdots \geq k_m$, $k_1 < B$, and $k_m > 1$. For any fixed m, r is maximized when $k_{i+1}/k_i = 1/d$ for all i. Thus $m = \log_d k_1$ and so $m/(\sum(k_{i+1}/k_i)) = d$. Thus r is maximized when $d = \log_d k_1$, i.e. when $r = d = O(\log k_1 / \log\log k_1)$. □

Theorem 3 *Any deterministic strategy will be $\Omega(\log B/\log\log B)$ from optimal on some set of jobs.*

Proof: (sketch) Let there be pB lists and let $\alpha = \log\log B/\log B$. In this sketch, we assume that all processors visit up to B nodes and then reschedule. 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). Each round, the adversary will kill off all lists assigned to the $(1-\alpha)p$ processors with the fewest lists assigned. 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. The adversary repeats this approach until $k_i < 2/\alpha$, at which point 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 and 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 completion time t 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. $\Omega(1/\alpha) = \Omega(\log B/\log\log B)$ 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 paper, except to remark that an adversary can force any 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.

6 Simulation results

Theorem 4 *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 time $O(B\ D(n)/\log B)$ with $W(n)\log B$ processors.*

Proof: In what follows, we will sketch the proof for simulating the circuit family by a CREW Phase LPRAM family that has one processor per circuit gate. Generalizations of the proof to an EREW Phase LPRAM family with only $W(n)\log B$ processors are omitted here.

Let c be the maximum fan-in in the circuit (c is a constant). Intuitively, we will divide the circuit into slices of depth $\log_c B$. The output of each node in a slice depends on at most B predecessors in the circuit from earlier slices (since the fan-in is c). Processor i will first read these (up to) B values for node i into its local memory. Using local operations that mimic the circuit, it will compute the output of node i from these B values. It will write the result to the shared memory, and then synchronize.

Here is a more precise description of how uniformly to generate a Phase LPRAM program that computes the same function as the circuit. Let M_a be a log-space Turing machine that generates a description of the arithmetic circuit, starting with the output node and proceeding level-by-level back to the inputs. On input n, the following log-space Turing machine M_p can generate a description of our Phase LPRAM program, one processor at a time, starting with its final program instructions. Encode the transition function for M_a into the transition function for M_p in such a way that M_p can simulate M_a. We maintain a counter of the current depth of the circuit modulo $\log_c B$, and handle all gates in a slice at a time. For each gate, 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. This can all be done in logarithmic space, and the resulting CREW Phase LPRAM program runs in time $O(B\ D(n)/\log B)$ since there are $D(n)/\log B$ slices. □

Remark: By theorem 4, we see that the 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 4 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 that runs in $O(B \log n / \log B)$ time using n^3 processors. We omit the details.

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.*

Theorem 4 saves a $\log B$ factor off the naive $O(B \cdot D(n))$ 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 7 *Let C be an arithmetic circuit over a commutative semiring $(R, +, *, 0, 1)$ with n nodes and degree d with values assigned to each input node from the domain R. The value for each node of the circuit can be computed in $O(B \log n \log(nd) / \log B)$ 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, Kaltofen[MRK88] 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 8 *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 [Eck79] and our result for the multiprefix problem.

7 Remarks

The PRAM model is a good model for studying parallel computation. Recently, however, there has been an emphasis on a more precise analysis of algorithms where logarithmic factors are important. Thus models become important that reflect the features of real machines, encourage good programming practice for these machines, are sufficiently simple to use for algorithm design and analysis, and sufficiently general to apply to a range of parallel machines. We believe the Asynchronous PRAM is such a model.

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, we can assume the processors approximate lockstep execution. We believe this is a reasonable model for designing algorithms for machines where arbitrary delays occur, but (a) they 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 tightly-coupled multiprocessors with regular networks and identical processors. Experimental research on existing machines is needed to test the validity of our model.

Further variations on the Asynchronous PRAM model are possible. Depending on the parallel machine, it may be feasible to compute a parallel prefix while implementing a synchronization point. If so, then another type of the Asynchronous PRAM would charge only B for parallel prefix. It may also be valuable to study models which present a non-uniform view of the shared memory, in which the communication delay depends on the distance from the processor issuing the request to the requested location. Such models may or may not permit the pipelining of memory requests.

Our definition of an Asynchronous PRAM eliminates the possibility of race conditions. No processor may read the same memory location that another processor writes unless there is a synchronization step involving both processors between the two accesses. This simplifies the use of our model, since each processor sees a deterministic view of the computation. However, for certain application areas, such as operating systems, models that permit race conditions, and other forms of nondeterminism, are needed.

Synchronization is an important consideration in massively parallel machines, of great concern to the hardware and software communities, but largely ignored by the theory community. Understanding the computational power of various synchronization assumptions and primitives leads to an understanding of an important trade-off in the design of parallel machines: namely, how important are particular synchronization assumptions versus how much it costs (in dollars, memory access time, machine cycle time, etc.) for the hardware to support these assumptions.

Acknowledgement

I thank Richard Karp, Jorge Sanz, Danny Soroker, and Edith Cohen for helpful discussions.

References

[ACS89] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On communication latency in PRAM computations. In *Proc. of the Symp. on Parallel Algorithms and Architectures (SPAA)*, 1989.

[AKS83] M. Ajtai, J. Komlos, and E. Szemeredi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[AM88] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. In John H. Reif, editor, *Lecture Notes in Computer Science, Vol. 319*, pages 81–90. Springer-Verlag, Berlin, 1988. Proc. of the 3rd Aegean Workshop on Computing (AWOC).

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21:201–208, 1974.

[CV86a] Richard Cole and Uzi Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *Proc. of the Symp. on Foundations of Computer Science (FOCS)*, pages 478–491, 1986.

[CV86b] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. of the Symp. on Theory of Computing (STOC)*, pages 1–6, 1987.

[CZ89] Richard Cole and Ofer Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proc. of the Symp. on Parallel Algorithms and Architectures (SPAA)*, 1989.

[Eck79] Diane M. Eckstein. Simultaneous memory access. Technical Report TR-79-6, Computer Science Dept., Iowa State University, Ames, Iowa, 1979.

[Kar88] Richard M. Karp. personal communication, 1988.

[KR88] Richard M. Karp and Vijaya L. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB-CSD-88-408, Computer Science Division, University of California, Berkeley, March 1988. To appear in *Handbook of Theoretical Computer Science*, North-Holland, Amsterdam, 1989.

[KRS88] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. Technical Report RC 13572, IBM T. J. Watson Research Center, Yorktown Heights, New York, March 1988.

[KU86] Anna Karlin and Eli Upfal. Parallel hashing - an efficient implementation of shared memory. In *Proc. of the Symp. on Theory of Computing (STOC)*, pages 160–168, 1986.

[LF81] N. A. Lynch and M. J. Fisher. On describing the behavior of an implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[MRK88] Gary L. Miller, Vijaya L. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal on Computing*, 1988.

[PY88] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proc. of the Symp. on Theory of Computing (STOC)*, pages 510–513, 1988.

[Ran87] A. Ranade. How to emulate shared memory. In *Proc. of the Symp. on Foundations of Computer Science (FOCS)*, pages 185–194, 1987.

[Sni88] Marc Snir. personal communication, 1988.

[TV85] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14:862–874, 1985.