The Queue-Read Queue-Write Asynchronous PRAM Model

Phillip B. Gibbons
Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974
gibbons@research.bell-labs.com

Yossi Matias
Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974
matias@research.bell-labs.com

Vijaya Ramachandran*
Dept. of Computer Sciences
University of Texas at Austin
Austin TX 78712
vlr@cs.utexas.edu

Abstract

This paper presents results for the *queue-read, queue-write asynchronous parallel random access machine* (QRQW ASYNCHRONOUS PRAM) model, which is the asynchronous variant of the QRQW PRAM model. The QRQW PRAM family of models, which was introduced earlier by the authors, permit concurrent reading and writing to shared memory locations, but each memory location is viewed as having a queue which can service at most one request at a time. In the basic QRQW PRAM model each processor executes a series of reads to shared memory locations, a series of local computation steps, and a series of writes to shared memory locations, and then synchronizes with all other processors; thus this can be viewed as a bulk-synchronous model. In contrast, in the QRQW ASYNCHRONOUS PRAM model discussed in this paper, there is no imposed bulk-synchronization between processors, and each processor proceeds at its own pace. Thus, the QRQW ASYNCHRONOUS PRAM serves as a better model for designing and analyzing truly asynchronous parallel algorithms than the original QRQW PRAM.

In this paper we elaborate on the QRQW ASYNCHRONOUS PRAM model, and we demonstrate the power of asynchrony over bulk-synchrony by presenting a work and time optimal deterministic algorithm on the QRQW ASYNCHRONOUS PRAM for the leader election problem and a simple randomized work and time optimal algorithm on the QRQW ASYNCHRONOUS PRAM for sorting. In contrast, no tight bounds are known on the QRQW PRAM for either deterministic or randomized parallel algorithms for leader election and the only work and time optimal algorithms for sorting known on the QRQW PRAM are those inherited from the EREW PRAM, which are considerably more complicated. Our sorting algorithm is an asynchronous version of an earlier sorting algorithm we developed for the QRQW PRAM, for which we use an interesting analysis to bound the running time to be $O(\lg n)$. We also present a randomized algorithm to simulate one step of a CRCW PRAM on a QRQW ASYNCHRONOUS PRAM in sublogarithmic time if the maximum contention in the step is relatively small.

*Supported in part by NSF grant CCR-90-23059 and Texas Advanced Research Projects Grant 003658480.

1 Introduction

The Parallel Random Access Machine (PRAM) model of computation (see, e.g., [KR90, JáJ92, Rei93]) consists of a number of processors operating in lock-step and communicating by reading and writing locations in a shared memory. Standard PRAM models can be distinguished by their rules regarding contention for shared memory locations. These rules are generally classified into the *exclusive* read/write rule in which each location can be read or written by at most one processor in each unit-time PRAM step, and the *concurrent* read/write rule in which each location can be read or written by any number of processors in each unit-time PRAM step. These two rules can be applied independently to reads and writes; the resulting models are denoted in the literature as the EREW, CREW, ERCW, and CRCW PRAM models.

In a previous paper [GMR96b], we argued that neither the *exclusive* nor the *concurrent* rules accurately reflect the contention capabilities of most commercial and research multiprocessors: The exclusive rule is too strict, and the concurrent rule ignores the large performance penalty of high contention steps. We proposed instead the *queue* rule, in which each memory location can be read or written by any number of processors in each step, but concurrent reads or writes to a location are serviced one-at-a-time. Thus the worst case time to read or write a location is linear in the number of concurrent readers or writers to the same location. As discussed in [GMR96b], the contention properties of most existing multiprocessors are well-approximated by the queue-read, queue-write rule.

In this paper we consider the *Queue-Read Queue-Write* (QRQW) ASYNCHRONOUS PRAM model. The QRQW ASYNCHRONOUS PRAM [GMR93] was introduced by the authors as the asynchronous variant of the QRQW PRAM family of models [GMR96b], suitable for designing algorithms for asynchronous (MIMD) multiprocessors. The QRQW family of models includes the SIMD-QRQW PRAM model, the QRQW PRAM model and the QRQW ASYNCHRONOUS PRAM model. All models in the QRQW family incorporate the queue rule described above, and permit concurrent reading and writing of shared memory locations at a cost that is linear in the number of such readers and writers. Each memory location is viewed as having a queue which can service at most one request at a time. Unlike related models accounting for contention (e.g. [DHW93, LAB93]), the QRQW PRAM and the QRQW ASYNCHRONOUS PRAM models permit pipelining: individual processors may have multiple requests in progress concurrently. Some of the results presented here are mentioned without any details in earlier extended abstracts by the authors on QRQW PRAM results.

The QRQW PRAM model is the basic model in the QRQW family of models and is well suited for the design and analysis of bulk-synchronous algorithms on machines such as the Cray C90, the Cray J90, and the forthcoming Tera MTA multiprocessor. These machines provide a QRQW contention rule at the memory cells, support the pipelining of memory requests, and provide sufficient processor-to-memory bandwidth to support communication at each step (as is needed for PRAMs). Efficient bulk-synchronization is an option on these machines, but is not imposed. An extensive study of algorithms and results for the QRQW PRAM can be found in [GMR96b, GMR96a]. In addition, experimental results for the QRQW PRAM on the Cray C90 and J90 can be found in [BGMZ95].

The model we study in this paper, the QRQW ASYNCHRONOUS PRAM model, permits more asynchronous behavior than the bulk-synchrony imposed by the QRQW PRAM. Thus it can be used to design and analyze algorithms for machines such as the MTA in contexts in which bulk-synchrony is not employed. Indeed, Burton Smith, Chairman and Chief Scientist of Tera Computer, refers to the MTA as "roughly a QRQW ASYNCHRONOUS PRAM" [Smi95] (and to our knowledge makes no such claims about other models).

In more detail, the differences between the QRQW PRAM and the QRQW ASYNCHRONOUS PRAM are as follows. In the (bulk-synchronous) QRQW PRAM, each processor executes a series of reads to shared memory locations, a series of local computation steps, a series of writes to shared memory locations, and then synchronizes with all the other processors. The time for such a bulk-synchronous step is the maximum of the number of reads, compute steps, and writes by any one processor and the maximum contention at any one location. Thus each processor waits until all queues have emptied. In contrast, in the QRQW ASYNCHRONOUS PRAM, each processor executes reads, compute steps, and writes, but processors proceed at their own pace with no intervening synchronization. Algorithms must be correct under worst case assumptions on the finite delays incurred by processors and in processing memory requests. However, the running times of algorithms are analyzed using an optimistic (synchronous) time metric in which, at each unit-time step, (1) each processor performs a local computation step or issues a shared memory request, with all requests to the same location appended to the queue for that location, in an arbitrary order, and then (2) each nonempty queue services the request at the head of its queue. Thus each processor may proceed as soon as its own requests clear their respective queues.

We present a number of algorithmic results for the QRQW ASYNCHRONOUS PRAM. First, we show how QRQW PRAM algorithms for problems such as generating a random permutation and multiple compaction can be readily adapted to QRQW ASYNCHRONOUS PRAM algorithms with the same time bounds. Then, in the bulk of the paper, we present QRQW ASYNCHRONOUS PRAM algorithms that achieve better time bounds than the best known QRQW PRAM algorithms, by exploiting the power of asynchrony over bulk-synchrony.

First, we present a simple deterministic algorithm for computing the OR of n bits on the QRQW ASYNCHRONOUS PRAM that exploits the lack of bulk-synchrony and runs in $O(\lg n/\lg\lg n)$ time, linear work. A similar algorithm was found independently by Armen and Johnson [AJ96]. We also present a matching lower bound. In contrast, no $o(\lg n)$ time QRQW PRAM algorithm is known, and even on a Concurrent-Read Queue-Write (CRQW) PRAM, no deterministic $o(\lg n)$ time algorithm is known for this problem.

Next, we present a simple randomized $O(\lg n)$ time, $O(n \lg n)$ work QRQW ASYNCHRONOUS PRAM algorithm to sort an array of n elements. The algorithm is almost exactly the same as the $\Theta(\lg^2 n/\lg\lg n)$ time, $O(n\lg n)$ work randomized sorting algorithm we developed earlier for the QRQW PRAM [GMR96a], but we exploit asynchrony by allowing elements to flow through the 'binary search fat-tree' data structures employed by the sorting algorithm at their own pace. We describe here a new analysis that is interesting in its simplicity; it is based on a repeated use of a seemingly quite useful lemma, regarding the sum of Poisson-like random variables.

Finally, we show that a single step of an n-processor FETCH&ADD PRAM, and hence also a CRCW PRAM, can be emulated on an n-processor QRQW ASYNCHRONOUS PRAM in $O(\lg n/\lg\lg n + \lg k)$ time with high probability, where k is the maximum memory contention of the CRCW PRAM step; in this emulation, the value of k is not known to the emulating algorithm. The emulation algorithm is rather involved, and demonstrates some of the difficulties that may arise when designing algorithms that avoid global synchronization.

The rest of the paper is organized as follows: In Section 2 we state some basic results in probability that we will use in later sections. In Section 3 we define the QRQW ASYNCHRONOUS PRAM model and discuss its features. In Section 4 we show how certain QRQW PRAM algorithms can be readily adapted to the QRQW ASYNCHRONOUS PRAM. In Section 5 we present our results on leader election. In Section 6 we present our randomized work- and time-optimal sorting algorithm on the QRQW ASYNCHRONOUS PRAM, including the detailed analysis of the contention encountered in the binary search fat-tree. In Section 7 we present our simulation of a FETCH&ADD PRAM step on a QRQW ASYNCHRONOUS PRAM. Some further discussion on the QRQW ASYNCHRONOUS PRAM cost measure appears in Section 8.

2 Probability facts and notations

A *Las Vegas* algorithm is a randomized algorithm that always outputs a correct answer, and obtains the stated bounds with some stated probability. All of the randomized algorithms in this paper are Las Vegas algorithms, obtaining the stated QRQW PRAM bounds with high probability. A probabilistic event occurs *with high probability (w.h.p.)*, if, for any prespecified constant $\delta > 0$, it occurs with probability $1 - 1/n^\delta$, where n is the size of the input. Thus, we say a randomized algorithm runs in $O(f(n))$ time w.h.p. if for every prespecified constant $\delta > 0$, there is a constant c such that for all $n \geq 1$, the algorithm runs in $c \cdot f(n)$ steps or less with probability at least $1 - 1/n^\delta$. Often, we can test whether the algorithm has succeeded, and if not repeat it. In this case, it suffices to design an algorithm that succeeds with probability $1 - 1/n^\epsilon$ for some positive constant ϵ, since we can repeat the algorithm δ/ϵ times if necessary, to boost the algorithm success probability to the desired $1 - 1/n^\delta$. With this in mind, we will freely use "with high probability" in this paper to refer to events or bounds that occur with probability $1 - 1/n^\epsilon$ for some positive constant ϵ.

A Bernoulli trial is an experiment which succeeds with probability p and fails with probability $1 - p$. A random variable X that gives the number of successes in a sequence of n independent Bernoulli trials is a *binomial random variable* with expectation $E[X] = np$. In Section 7 we apply the following "Chernoff-type" bound on the tail of a binomial random variable X (see, e.g. [GMR96b]):

Lemma 2.1 *Let X be a binomial random variable. For all $f = O(\lg n)$, if $E[X] \leq 1/2^f$, then $X = O(\lg n/f)$ w.h.p.*

3 The QRQW Asynchronous PRAM

In this section, we present the definition of the QRQW ASYNCHRONOUS PRAM model.

An important feature of the QRQW ASYNCHRONOUS PRAM model is that the model separates correctness issues from analysis issues: Algorithms must be correct under worst case assumptions on the finite delays incurred by the processors and in processing memory requests, but the running times of algorithms are analyzed using an optimistic (synchronous) time metric. We elaborate on the correctness issues and analysis issues below, and then proceed to define the model.

Functionality and correctness. A shared memory multiprocessor supports a *consistency condition* on its memory system. The most widely-used memory consistency condition is *sequential consistency* [Lam79, ABM93], in which the memory system appears to be a serial memory, processing one read or write at a time, in an order consistent with the individual program orders at each processor. The SGI Challenge and the (now defunct) KSR machines are examples of multiprocessors supporting sequential consistency. Relaxed consistency conditions such as *release consistency* [GLL+90, GMG91] support sequential consistency for *PL* programs; these are programs with two types of accesses, synchronization and data, such that there are no race conditions between data accesses. The Stanford DASH machine and the Tera MTA are examples of multiprocessors supporting release consistency. In the QRQW ASYNCHRONOUS PRAM, the memory system is assumed to be sequentially consistent. As any program can be made *PL* by labeling sufficiently many accesses as "synchronization", our algorithms will work as well on machines providing release consistency.

Typically, the only other guarantee on inter-processor communication provided by a multiprocessor is that no request is delayed indefinitely. (We are assuming that the multiprocessor is executing without failures.) Thus algorithms must be correct under worst case assumptions on the delays incurred by processors and in processing memory requests, and the QRQW ASYNCHRONOUS PRAM reflects this reality.

Most asynchronous shared memory models of computation assume that a processor can have at most one pending memory request at a time: there is no pipelining of memory requests by a processor (e.g. [CZ89, Nis90, And92, MPS92, DHW93]).[1] On the other hand, high-performance shared memory machines such as the Tera MTA permit the pipelining of memory accesses by a processor, in order to amortize the round-trip time to memory over a collection of accesses. In the QRQW ASYNCHRONOUS PRAM, pipelining of memory accesses is permitted; a processor may have multiple shared memory operations in progress at a time.

Each processor has a private local memory, and the following types of instructions: local operations, shared memory reads, shared memory writes, and shared memory TEST&SET operations. A TEST&SET operation reads and returns the old value and writes a 1; the location is assumed to be initialized to 0. Other synchronization constructs such as barriers can be constructed using shared memory reads, writes, and TEST&SETs.

Analysis. In defining how algorithms are analyzed in the model, the QRQW ASYNCHRONOUS PRAM aims for a simple cost model that captures important realities of multiprocessors. As in Gibbons' ASYNCHRONOUS PRAM model [Gib89], our cost model assumes that processors issue instructions at the same speed, as this is presumed to be the typical scenario in a multiprocessor. A local operation takes unit time.

There is a FIFO queue associated with each memory location; only the request at the head of the queue is processed in a step. Thus requests to a location can pile up, causing a delay in their processing. If k processors issue a request to the same location at step t of an algorithm, and the queue for this location is empty at the beginning of step t, then one such request completes step t, another step $t+1$, another step $t+2$, and so forth, until the last one completes at step $t+k-1$. If additional requests to the location arrive before step $t+k-1$, these are appended to the tail of the queue: if there are two such requests, they will complete at steps $t+k$ and $t+k+1$, respectively, regardless of the exact step at which they are requested.

Note that the cost model makes optimistic assumptions on the delays encountered by shared memory requests, e.g. that requests issued earlier are queued before requests issued later; these assumptions are *not* a part of the correctness model. The reasoning behind models in which analysis makes optimistic assumptions while correctness does not is that (1) it makes sense to measure the complexity of an algorithm so as to approximate a typical performance of a machine, since this reflects directly in real life efficiency, while (2) we must be strict and assume worst case situations for correctness, since otherwise a single unexpected event may cause the entire computation to fail. Some ramifications of this reasoning are discussed in Section 8.

Model definition. The QRQW ASYNCHRONOUS PRAM model consists of a collection of processors operating asynchronously and communicating via a global shared memory. Each processor has a private local memory, and the following types of instructions: RAM operations involving only its private state and private memory, requests to read the contents of a shared memory location into a private memory location, requests to write the contents of a private memory location to a shared memory location, and requests to perform a TEST&SET operation on a shared memory location. A processor can execute any of the shared memory requests and continue without waiting for them to complete (pipelining). However, the first subsequent RAM operation that uses the result of such a shared memory request will wait for the value to be returned.

The global memory is a *sequentially consistent nonblocking shared memory* [GM92], as follows. Each processor issues shared memory requests (read, write, TEST&SET) one at a time. There is a partial order on the requests by a processor, called the *local order* for that processor. The memory system appears (for the purpose of correctness) to be a serial memory that processes one read or write at a time, in an order consistent with the individual local orders at each processor. The local

[1] Note that when all memory accesses take unit time, as in these models, there is no need for pipelining.

order for a processor must be a subrelation of the total order in which the processor issues the requests, and two requests by a processor to the same location must be ordered whenever one or both are write requests. Algorithms must be correct under worst case assumptions on the finite delays incurred by the processors and by the shared memory.

Time is defined as follows. There is a FIFO queue associated with each memory location. A single time step consists of two substeps:

1. Each processor issues an instruction. Local operations complete this step. Shared memory requests are appended to the tails of the queues for the requested locations, with requests to the same location enqueued in an arbitrary order.
2. Shared memory requests at the head of nonempty queues are dequeued and performed (at most one per queue), and either a return value or an acknowledgement is received by the processor responsible for the request.

Work is defined as the time-processor product.

Some comments on the definition follow. Because an algorithm must be correct regardless of the delays, a processor can not safely "time-out" after a certain period of time or a certain amount of polling and assume that no further reads/writes to a location are forthcoming. Any inference a processor makes regarding the length of a queue encountered by one of its requests based on the delay incurred may be completely inaccurate, since even a request encountering an empty queue may incur arbitrary delays. Once issued, a memory request can not be withdrawn; a processor has not completed its participation in an algorithm until all of its memory requests have been processed.

In the algorithms we present, the local order at each processor is defined implicitly as follows: if x and y are shared memory requests by the same processor, such that x is requested before y, then x precedes y in the local order if and only if (1) x and y are to the same location and at least one is a write request, or (2) x or y are used in a synchronization step of the algorithm. Thus, for example, if a processor issues a sequence of read requests and then synchronizes, the local order is implicitly defined to order each read before the synchronization request but not order the reads with respect to one another.

In addition to the QRQW ASYNCHRONOUS PRAM model, one can also define hybrid models such as the CRQW ASYNCHRONOUS PRAM, which permits unit time concurrent reading but applies the above queue rule for concurrent writing. The stronger CRQW ASYNCHRONOUS PRAM model is used primarily to prove stronger lower bounds.

Related work. A variety of ASYNCHRONOUS PRAM models have been studied in the literature (c.f. [CZ89, Gib89, Nis90, And92, MPS92]). These models account for contention in a manner most like a CRCW PRAM, with no penalty assessed for large contention to a location.[2] An EREW contention rule was not considered,[3] since most asynchronous algorithms cannot avoid scenarios in which concurrent reading or writing occur. Since most existing parallel machines permit contention, but at a cost, the QRQW rule is a better choice for an asynchronous model than either the CRCW or the EREW rule.

The QRQW rule can be incorporated into these previous models in a natural way. Concurrent reads and writes to a location x are queued in an arbitrary order, with each write to x updating the value

[2]For example, models based on "time slots" permit an arbitrary number of reads/writes to a location in one time slot. Models based on "interleaving" or "rounds" charge the same for an interleaving of reads/writes to the same location as for an interleaving of reads/writes to different locations.

[3]An exception is the EREW variant of Gibbons' ASYNCHRONOUS PRAM model [Gib89], which permits contention in synchronization primitives, at a cost, but enforces the EREW rule on reads and writes occurring between synchronization points.

of x when it reaches the head of the queue and each read of x returning the value present in location x when it reaches the head of the queue. Instead, we have defined a new model that incorporates the QRQW rule, which we believe to be a better model for asynchronous parallel machines.

Two other asynchronous models of parallel computation that focus on contention are the atomic message passing model of Liu, Aiello and Bhatt and the "stall" model of Dwork, Herlihy and Waarts. These models were developed independently of the QRQW ASYNCHRONOUS PRAM and differ in several important ways. The atomic message passing model [LAB93] is a message-passing model in which messages destined for the same processor are serviced one-at-a-time in an arbitrary order. The model permits general asynchronous algorithms, but each processor can have at most one message outstanding at a time. Dwork, Herlihy and Waarts [DHW93] defined an asynchronous shared memory model with a *stall* metric: If several processes have reads or writes pending to a location, v, and one of them receives a response, then all the others incur a stall. Hence the charge for contention is linear in the contention, with requests to a location being serviced one-at-a-time. Their model permits general asynchronous algorithms, but each processor can have at most one read or write outstanding at a time. Unlike their model, the QRQW ASYNCHRONOUS PRAM model captures *directly* how the contention delays the overall running time of the algorithm, and is proposed as an alternative to other PRAM models for high-level algorithm design.

4 Adapting QRQW PRAM algorithms to the QRQW Asynchronous PRAM

The computational power of the QRQW PRAM and the QRQW ASYNCHRONOUS PRAM are incomparable: the QRQW PRAM has the advantage of free global synchronization, but is restricted to bulk-synchronous operation. The naive emulation of the QRQW PRAM on the QRQW ASYNCHRONOUS PRAM performs a barrier synchronization at each step, at a cost of $O(\lg p)$ for p processors per barrier. The goal in adapting algorithms designed for the QRQW PRAM to the QRQW ASYNCHRONOUS PRAM is to make do with less synchronization so as to maintain the same complexity bounds. In this section, we sketch simple adaptations of several QRQW PRAM algorithms from [GMR96b, GMR96a], showing that the same complexities can be obtained for the QRQW ASYNCHRONOUS PRAM.

Theorem 4.1 *Consider the problem of electing a leader bit from among the k out of n input bits that are 1.*

- *Let $\hat{k}$ be known to be within a factor of $2^{\sqrt{\lg n}}$ of k, i.e. $\hat{k}/2^{\sqrt{\lg n}} \leq k \leq \hat{k}2^{\sqrt{\lg n}}$. There is a randomized Monte Carlo* QRQW ASYNCHRONOUS PRAM *algorithm that, w.h.p., elects a leader in $O(\sqrt{\lg n})$ time with $O(n)$ work. On the* CRQW ASYNCHRONOUS PRAM, *or if $\hat{k} = O(2^{\sqrt{\lg n}})$, the same bounds can be obtained for a Las Vegas algorithm.*
- *Consider the problem of computing the logical* OR *(or electing a leader) of n bits, where it is known that at most k_{max} input bits can be 1. There is a randomized Las Vegas* QRQW ASYNCHRONOUS PRAM *algorithm that runs in $O(\lg k_{\max} + \sqrt{\lg n})$ time with $O(n)$ work w.h.p.*

Proof. We describe first the QRQW PRAM algorithm from [GMR96b] for the first problem above for $n/\sqrt{\lg n}$ processors. Let $\rho = \min(1, 2^{c\sqrt{\lg n}}/\hat{k})$, for a constant $c \geq 1$, to be determined by the analysis. Let A be an array of size $m = 2^{(c+2)\sqrt{\lg n}}$, initialized to all zeros.

Step 1. Each processor reads from its $\sqrt{\lg n}$ input bits and selects a leader from among the bits that are 1, if any.

Step 2. Each processor with a leader writes, with probability ρ, the index of the leader bit to a cell of A selected uniformly at random.

Step 3. m of the processors participate to select a nonzero index from among those written to A, using a binary fanin approach.

If $\hat{k} \leq 2^{\sqrt{\lg n}}$ then $\rho = 1$ and this is a Las Vegas algorithm. Alternatively, on the CRQW, a Las Vegas algorithm is obtained by repeating steps 2 and 3 until there is a nonzero index in A; termination is detected by using the concurrent-read capability.

In [GMR96b], we show that the time on the QRQW and CRQW PRAM is $O(\sqrt{\lg n})$ w.h.p. We adapt the algorithm to the QRQW and CRQW ASYNCHRONOUS PRAM as follows. In step 2, processors perform a TEST&SET to their selected cell instead of writing an index. For step 3, consider a binary tree whose leaves are the cells of A. Each processor that succeeded in claiming a cell of A (i.e., its TEST&SET returned a 0) marches from its leaf towards the root of the tree, attempting to claim each node in turn using TEST&SET, and dropping out if its attempt fails. The leader is the leader bit of the processor that succeeds in claiming the root.

We now describe the QRQW PRAM algorithm from [GMR96b] for the second problem above, for $n/(\lg k_{\max} + \sqrt{\lg n})$ processors. The input bits are partitioned among the processors such that each processor is assigned $\lg k_{max} + \sqrt{\lg n}$ bits. If $k_{max} = \Omega(n^{\epsilon})$ for some constant $0 < \epsilon \leq 1$, then elect the leader using a binary fanin tree, to obtain the stated bounds. Otherwise, let A be an array of size $m = k_{max} \cdot 2^{\sqrt{\lg n}}$, initialized to all zeros (note that $m = O(n)$). Each processor selects a leader from among its input bits that are 1, if any. Then each processor with a leader writes to a cell of A selected uniformly at random. Finally, m of the processors participate to select a nonzero index from among those written to A. In [GMR96b], we show that the time on the QRQW PRAM is $O(\lg k_{max} + \sqrt{\lg n})$ w.h.p. We adapt the algorithm to the QRQW ASYNCHRONOUS PRAM using the same modifications as in the first problem above, to obtain the stated bounds. ∎

The general leader election problem (k unknown) is discussed in Section 5.

We next consider the problem of *generating a random permutation.*

Theorem 4.2 *There is a randomized Las Vegas* QRQW ASYNCHRONOUS PRAM *algorithm for generating a random permutation that runs in $O(\lg n)$ time with $O(n)$ work w.h.p.*

Proof. In [GMR96a], we presented the following QRQW PRAM algorithm. For each of $c \lg\lg n$ rounds, for a constant $c \geq 1$, each unplaced item selects a random cell from a subarray of an array A (a new subarray is used for each round); if no other item selects the same cell, the item has been successfully placed. The size of the subarray used in the first round is $d \cdot n$, for some constant $d > 1$, and the size decreases by a constant factor at each round. If, after $c \lg\lg n$ rounds, not all items have been placed, restart from the beginning. After all items have been placed, the array A is compacted to size n.

The algorithm is adapted to the QRQW ASYNCHRONOUS PRAM by using the TEST&SET primitive to decide which writer claims a particular cell, and judiciously inserting explicit synchronizations among subsets of processors as needed. The time and work bounds follow from the bounds for the QRQW PRAM shown in [GMR96a]. ∎

We next consider the *multiple compaction* problem. The input consists of n items given in an array; each item has a *label*, a *count*, and a *pointer*, all from $[1..O(n)]$. The labels partition the items into n sets $\Phi_1, \ldots, \Phi_n$ where Φ_i is the set of items labeled with i. The count of an item belonging to Φ_i is an upper bound, $count(\Phi_i)$, on the number of items in Φ_i, $|\Phi_i|$, such that $\sum_{i=1}^{n} count(\Phi_i) \leq \alpha n$ for some constant $\alpha > 0$. Also given is an array $B[1..\alpha' n]$, where $\alpha' \geq 4\alpha$ is a constant. Array B is partitioned into sub-arrays such that each set Φ_i has a private subarray of size at least $4 \cdot count(\Phi_i)$; the sub-arrays are assigned in some arbitrary order. The pointer of an item belonging to a set Φ_i is the starting point in B of the sub-array assigned to Φ_i. The goal is to move each item into a private cell in the sub-array of its set.

Theorem 4.3 *There is a randomized Las Vegas* QRQW ASYNCHRONOUS PRAM *algorithm for multiple compaction that runs in $O(\lg n)$ time with $O(n)$ work w.h.p.*

Proof. In [GMR96a], we presented a QRQW PRAM algorithm with these time and work bounds. The reader is referred to that paper for a description of the algorithm. As above, the algorithm is adapted to the QRQW ASYNCHRONOUS PRAM by using the TEST&SET primitive to decide which writer claims a particular cell, and judiciously inserting explicit synchronizations among subsets of processors as needed. ∎

Even more interesting than adapting QRQW PRAM algorithms to the QRQW ASYNCHRONOUS PRAM are examples of algorithms for the QRQW ASYNCHRONOUS PRAM that achieve *better* time bounds than the best known QRQW PRAM algorithms. Such algorithms exploit the computational advantage the QRQW ASYNCHRONOUS PRAM has by not being restricted to bulk-synchronous operation. We discuss three such examples in the next three sections.

5 Leader election and computing the OR

Given a Boolean array of n bits, the OR function is the problem of determining if there is a bit with value 1 among the n input bits. The *leader election* problem is the problem of electing a leader bit from among the k out of n bits that are 1 (k unknown). The output is the index in $[1..n]$ of the bit, if $k > 0$, or 0, if $k = 0$. This generalizes the OR function, as long as $k = 0$ is possible.

By having each processor whose input bit is 1 write the index of the bit in the output memory cell, we obtain a simple deterministic QRQW ASYNCHRONOUS PRAM algorithm for leader election (and similarly for the OR function) that runs in $\max\{1, k\}$ time using n processors, where k is the number of input bits that are 1 (k unknown). This is a fast algorithm if we know in advance that the value of k is small. However, for the general leader election problem, a better algorithm is to mimic the EREW PRAM parallel prefix algorithm to compute the location of the first 1 in the input; since only pairwise synchronizations are used, this takes $\Theta(\lg n)$ time and $\Theta(n)$ work on a QRQW ASYNCHRONOUS PRAM.

In this section, we present a faster, $O(\lg n/\lg\lg n)$ time deterministic QRQW ASYNCHRONOUS PRAM algorithm for leader election and computing the OR function, and a matching lower bound for the stronger CRQW ASYNCHRONOUS PRAM. A similar algorithm was found independently by Armen and Johnson [AJ96].

Theorem 5.1 *There is a deterministic* QRQW ASYNCHRONOUS PRAM *algorithm for the leader election problem (and the* OR *function) that runs in $O(\lg n/\lg\lg n)$ time and $O(n)$ work.*

Proof. Let $s = \lg n/\lg\lg n$. We describe the algorithm for n/s processors. Each processor is assigned s inputs, and elects as leader the first 1-input among its inputs (if any). Consider an s-ary tree, T, with one leaf per processor, with each location corresponding to a node in T initialized to zero. Each processor with a 1-input among its inputs begins to greedily traverse the path in T from its leaf to the root. At each node on the path, it attempts to claim the node using a TEST&SET operation. If it returns a zero, the processor has succeeded in claiming the node, and it continues on to the next node in its path. Else it drops out. The leader elected is according to the processor claiming the root node. No processor spends more than s steps being the first in the queue for a node (and hence claiming the node) and no more than s steps stuck in the queue for a node (when it drops out). Thus the time is $O(s)$ as claimed. ∎

We can derive a matching $\Omega(\lg n/\lg\lg n)$ lower bound for the OR function on the (more powerful) CRQW ASYNCHRONOUS PRAM using a lower bound result of Dietzfelbinger, Kutylowski and Reischuk [DKR94] for the *few-write* PRAM models. Recall that the few-write PRAM models are parameterized by the number of concurrent writes to a location permitted in a unit-time step. (Exceeding this number is not permitted.) Let the *κ-write* PRAM denote the few-write PRAM model that permits concurrent writing of up to κ writes to a location, as well as unlimited concurrent reading. We begin by proving a more general result for emulating the CRQW ASYNCHRONOUS PRAM on the few-write PRAM, and then provide the OR lower bound. The same two-part approach is used in [GMR96b] to prove an $\Omega(\lg n/\lg\lg n)$ time lower bound for the deterministic CRQW PRAM; here we extend the lower bound to the asynchronous model.

Lemma 5.2 *A p-processor* CRQW ASYNCHRONOUS PRAM *deterministic algorithm running in time t can be emulated on a p-processor t-write* PRAM *in time $O(t)$.*

Proof. Since the CRQW ASYNCHRONOUS PRAM algorithm runs in time at most t on all inputs, then at each step, the maximum number of writes to any one location initiated that step is no more than t, regardless of the input. Thus we will use a fixed constant number of t-write PRAM steps to emulate each CRQW ASYNCHRONOUS PRAM instruction. For each CRQW ASYNCHRONOUS PRAM processor p_j, $j \in [1..p]$, we denote by p'_j the few-write PRAM processor emulating p_j. Consider each instruction in the ASYNCHRONOUS PRAM program in turn. We show how to handle each type of instruction.

- For each processor p_j with an instruction to issue a local operation or a shared memory read, p'_j has an instruction to perform the local operation or the shared memory read, and then instructions to idle for the rest of this step.
- For each processor p_j with an instruction to issue a TEST&SET for a shared memory location X, returning the old value into a local location $r1$, p'_j has the following sequence of instructions: (1) read X into $r1$, (2) if the value is 0, then write j to X, read X into local location $r2$, and if the value is not j, set $r1$ to 1, and (3) write 1 to X.
- For each processor p_j with an instruction to issue a shared memory write, p'_j has instructions to idle and then an instruction to perform the shared memory write at the same time as (3) of the previous case.

The sequence of instructions for a TEST&SET operation ensure that if the old value is 0, then exactly one processor returns a zero, the rest return a 1, and the location is set to 1. If the old value is not 0, then all processors return the old value, and the location is set to 1. The idle steps ensure that the processors remain in sync, and do not interfere with a TEST&SET emulation in progress.

The t-write PRAM will take constant time for each CRQW ASYNCHRONOUS PRAM instruction. Thus the time on the t-write PRAM is $O(t)$. Since the ASYNCHRONOUS PRAM program is required to be correct (and terminate) regardless of the relative progress made by the processors, then in particular it is correct (and terminates) under the specific timing of events used by the t-write PRAM emulation. ∎

The above lemma leads to the following theorem that gives the desired lower bound.

Table 1: **PRAM Sorting Results**

Model	time	work	space	constant factors	Reference
EREW PRAM	$O(\lg^2 n)$	$O(n \lg^2 n)$	$O(n)$	small	[Bat68]
EREW PRAM	$O(\lg n)$	$O(n \lg n)$	$O(n)$	very large	[AKS83]
EREW PRAM	$O(\lg n)$	$O(n \lg n)$	$O(n)$	large	[Col88]
CREW PRAM	$O(\lg n)$ w.h.p.	$O(n \lg n)$ w.h.p.	$O(n^{1+\epsilon})$	moderate	[Rei85]
QRQW PRAM	$O(\lg^2 n / \lg\lg n)$ w.h.p.	$O(n \lg n)$ w.h.p.	$O(n)$	moderate	[GMR96a]
QRQW A.P.	$O(\lg n)$ w.h.p.	$O(n \lg n)$ w.h.p.	$O(n \lg n)$	moderate	this paper

Theorem 5.3 *Any deterministic algorithm for computing the* OR *function on a* CRQW ASYNCHRONOUS PRAM *with arbitrarily many processors requires* $\Omega(\lg n / \lg\lg n)$ *time.*

Proof. Dietzfelbinger, Kutylowski and Reischuk [DKR94] proved an $\Omega(\lg n / \lg \kappa)$ lower bound for the OR function on the κ-write PRAM. Let T be the time for the OR function on the CRQW ASYNCHRONOUS PRAM. Then by Lemma 5.2, the OR function can be computed on the T-write PRAM in $O(T)$ time. Thus $T = \Omega(\lg n / \lg T)$, and hence $T \lg T = \Omega(\lg n)$. Now if $T = o(\lg n / \lg\lg n)$, then $\lg T = o(\lg\lg n)$, contradicting $T \lg T = \Omega(\lg n)$. Thus $T = \Omega(\lg n / \lg\lg n)$. ∎

6 Sorting

We consider the problem of general sorting, i.e. sorting an array of n keys from a totally-ordered set. On the EREW PRAM, there are two known $O(\lg n)$ time, $O(n \lg n)$ work algorithms for general sorting [AKS83, Col88]; these deterministic algorithms match the asymptotic lower bounds for general sorting on the EREW and CREW PRAM models. Unfortunately, these two algorithms are not as simple and practical as one would like.

Another relatively simple parallel sorting algorithm is a randomized $\sqrt{n}$-sample sort algorithm for the CREW PRAM that runs in $O(\lg n)$ time, $O(n \lg n)$ work, and $O(n^{1+\epsilon})$ space [Rei85]. This algorithm consists of the following high-level steps: (1) randomly sample $\sqrt{n}$ keys, (2) sort the sample by comparing all pairs of keys, (3) each item determines by binary search its position among the sorted sample and labels itself accordingly, (4) sort the items based on their labels using integer sorting, and (5) recursively sort within groups with the same label. When the size of a group is at most $\lg n$, finish sorting the group by comparing all pairs of items.

In an earlier paper [GMR96a] we build on this $\sqrt{n}$-sample sort algorithm and obtained an $O(\lg^2 n / \lg\lg n)$ time, $O(n \lg n)$ work, $O(n)$ space randomized sorting algorithm, on the QRQW PRAM.

In this section, we present a simple sorting algorithm on the QRQW ASYNCHRONOUS PRAM that runs in $O(\lg n)$ time with $O(n \lg n)$ work w.h.p. The algorithm is almost the same as the $O(n \lg n)$-work algorithm for the QRQW PRAM given in [GMR96a], but we are able to bring down the running time from $\Theta(\lg^2 n / \lg\lg n)$ to $O(\lg n)$ by making effective use of asynchrony. In particular we analyze the progress of elements through the binary search fat-trees and establish that the time taken by all elements to proceed through the binary search fat-trees at all recursive levels is $O(\lg n)$ w.h.p. Our algorithm uses $O(n \lg n)$ space.

Table 1 summarizes the comparison between various PRAM sorting algorithms.

We start by reviewing the high-level algorithm, which is the same for the QRQW PRAM and the

QRQW ASYNCHRONOUS PRAM.

6.1 The high-level sorting algorithm

The following sorting algorithm is presented in [GMR96a].

Algorithm $\mathcal{A}$.

Let ϵ be any constant such that $0 < \epsilon < 1/2$. Let $n = n_0$ be the number of input items, and for $i \geq 1$, let

$$n_i = (1 + 1/\lg n) \cdot n_{i-1}^{\frac{1}{2}+\epsilon}.$$

W.h.p., n_i is an upper bound on the number of items in each subproblem at the ith recursive call to $\mathcal{A}$ [GMR96a].

For subproblems at the ith level of recursion:

1. Let S be the set of at most n_i items in this subproblem. Select in parallel $\sqrt{n_i}$ items drawn uniformly at random from S.
2. Sort these sample items by comparing all pairs of items, using summation computations to compute the ranks of each item, and then storing the items in an array B in sorted order. Move every (n_i^{ϵ})th item in B to an array B'.
3. For each item $v \in S$, determine the largest item, w, in B' that is smaller than v, using a binary search on B'. Label v with the index of w in B'.
4. Place all items with the same label into a subarray of size $\Theta(n_i^{1/2+\epsilon})$ designated for the label, using multiple compaction. In particular, we use a variant of multiple compaction in which (1) the size of each set is $\Omega(\lg^2 n)$ and (2) the set sizes may exceed their upper bounds, in which case the algorithm reports failure [GMR96a]. W.h.p., the number of items with the same label is at most n_{i+1} and thus the multiple compaction succeeds in placing all items in each such group into its designated subarray. If failure is reported for any subproblem, we restart the algorithm from the beginning.
5. Recursively sort the items within each group, for all groups in parallel. When n_{i+1} is at most $2^{(\lg n)^{1/2}}$, finish sorting the group using the EREW PRAM bitonic sort algorithm [Bat68]. This cut-off point suffices for n sufficiently large; for general n, the cut-off point is $\max\{2^{(\lg n)^{1/2}}, \lg^c n\}$, for $c > 6/\epsilon$ a suitable constant.

To implement Algorithm $\mathcal{A}$ on a QRQW PRAM or QRQW ASYNCHRONOUS PRAM, we must incorporate techniques that use only low-contention steps. The main obstacle is step 3, in which each item needs to learn its position relative to the sorted sample. A straightforward binary search on B' would encounter $\Theta(n)$ contention. Instead, we employed the following data structure:

Binary search fat-tree. In a *binary search fat-tree*, there are n copies of the root node, $n/2$ copies of the two children of the root node, and in general, $n/2^j$ copies of each of the 2^j distinct nodes at level j down from the root of the tree. The added fatness over a traditional binary search tree ensures that, if n searches are performed in parallel such that not too many searches result in the same leaf of the (non-fat) tree, then each step of the search will encounter low contention.

The process of fattening a search tree can be done in $O(\lg n)$ time and $O(n \lg n)$ work using binary broadcasting.

In the case of our QRQW ASYNCHRONOUS PRAM sorting algorithm, at the ith level of recursion we make $2\beta n_i$ copies of the median splitter, $2\beta n_i/2$ copies of the 1/4 and 3/4 splitters, and so forth, down to $2\beta n_i^{1/2+\epsilon}$ copies of the $n_i^{1/2-\epsilon}$ splitters in the leaves of the tree, for $\beta > 2$ a suitable constant. We will continue to call this a 'binary search fat-tree' although the number of copies in each level differs by a constant factor from the number in the original definition.

The key to our $O(\lg n)$ time implementation of algorithm $\mathcal{A}$ on the QRQW ASYNCHRONOUS PRAM is that, in the QRQW ASYNCHRONOUS PRAM, processors can proceed through the binary search fat-tree at their own pace.

Theorem 6.1 *Algorithm $\mathcal{A}$ can be implemented on the* QRQW ASYNCHRONOUS PRAM *in $O(\lg n)$ time and $O(n \lg n)$ work w.h.p., using $O(n \lg n)$ space.*

Proof. Consider all $O(n/n_i)$ subproblems at the ith level of recursion. As shown in [GMR96a] using Chernoff bounds, the maximum contention in step 1 is $O(\sqrt{\lg n})$ w.h.p. The work is $O(n/\sqrt{n_i})$. Step 2 can be done in $O(\lg n_i)$ time and $O(n)$ work by first making $\sqrt{n_i}$ copies of each item in the sample. For step 3, we build a binary search fat-tree of depth $\lg(n_i^{1/2-\epsilon})$, in $O(\lg n_i)$ time and $O(n \lg n_i)$ work. We then label each item using a random search into the fat-tree, as described above. This step is analyzed below. By the analysis in [GMR96a], step 4 takes $O(\lg^* n_i \lg n/\lg\lg n)$ time and $O(n)$ work w.h.p.

Let τ be the number of levels of recursion. The total time spent on all recursive calls excluding the fat-tree searches is, w.h.p., $\sum_{1\leq i\leq \tau} O(\lg n_i + \lg^* n_i \lg n/\lg\lg n)$. Since $\lg n_i = O((1/2+\epsilon)^i \lg n$ and $\lg^* n_i < \lg^* n$, the total time excluding the fat-tree searches is, w.h.p.,

$$O((\tau \lg^* n/\lg\lg n)\lg n) + \sum_{1\leq i\leq \tau} O((1/2+\epsilon)^i \lg n) = O(\lg n).$$

The total work is, w.h.p.,

$$\sum_{1\leq i\leq \tau} O(n \lg n_i) = O(n \lg n).$$

The time for bitonic sort on groups of size at most $2^{\sqrt{\lg n}}$ is $O(\lg n)$, while the total work performed is $O(n \lg n)$ over all groups. Broadcasting whether any failure has occurred is done only after the bitonic sort, and takes $O(\lg n)$ time and linear work.

We show in Lemma 6.4 below that for each element, the sum of the contentions it encounters during all fat-tree searches is $O(\lg n)$. Theorem 6.1 follows. ∎

6.2 Delay analysis

In this section, we complete the proof of Theorem 6.1, by presenting a detailed analysis showing that the cumulative delay of any element through fat-trees at all levels of recursion is $O(\lg n)$ w.h.p. Our analysis will repeatedly use the following useful lemma regarding the sum of Poisson-like random variable.

Lemma 6.2 *Let $\beta > 2$, $c \geq \beta - 1$, and $\alpha = \lg c/\lg(\beta/2)$. Let $x_1, \ldots, x_m$ be independent random variables over the positive integers so that* $\mathbf{Prob}\,(x_i = u) \leq c\beta^{-u}$ *for all $u > 0$. Let $S_m = x_1 + \cdots + x_m$, for $m \geq 1$ and $S_0 = 0$. Then, for all a,*

$$\mathbf{Prob}\,(S_m \geq \alpha m + a) \leq \left(\frac{\beta}{2}\right)^{-a} . \tag{1}$$

Proof. The proof is by induction on m.

The base case is $m = 0$: If $a \leq 0$ then $\mathbf{Prob}\,(S_0 \geq a) = 1 \leq \left(\frac{\beta}{2}\right)^{-a}$. If $a > 0$ then $\mathbf{Prob}\,(S_0 \geq a) = 0 < \left(\frac{\beta}{2}\right)^{-a}$.

We assume inductively that (1) holds for $m-1$ and prove the induction step for $m > 0$.

$$
\begin{aligned}
\mathbf{Prob}\,(S_m \geq \alpha m + a) \quad &= \quad \sum_{i=-\infty}^{\infty} \mathbf{Prob}\,(x_m = i \wedge S_{m-1} \geq \alpha m + a - i) \\
&\overset{\text{by independence}}{=} \quad \sum_{i=-\infty}^{\infty} \mathbf{Prob}\,(x_m = i) \cdot \mathbf{Prob}\,(S_{m-1} \geq \alpha m + a - i) \\
&\overset{\text{since } x_m > 0}{=} \quad \sum_{i=1}^{\infty} \mathbf{Prob}\,(x_m = i) \cdot \mathbf{Prob}\,(S_{m-1} \geq \alpha m + a - i) \\
&\overset{\text{by assumption}}{\leq} \quad \sum_{i=1}^{\infty} c\beta^{-i} \cdot \mathbf{Prob}\,(S_{m-1} \geq \alpha m + a - i) \\
&= \quad \sum_{i=1}^{\infty} c\beta^{-i} \cdot \mathbf{Prob}\,(S_{m-1} \geq \alpha(m-1) + a + \alpha - i) \\
&\overset{\text{by induction}}{\leq} \quad \sum_{i=1}^{\infty} c\beta^{-i} \cdot \left(\frac{\beta}{2}\right)^{-(a+\alpha-i)} \\
&= \quad \left(\frac{\beta}{2}\right)^{-a} \cdot c \cdot \left(\frac{\beta}{2}\right)^{-\alpha} \cdot \sum_{i=1}^{\infty} \beta^{-i} \cdot \left(\frac{\beta}{2}\right)^{i} \\
&= \quad \left(\frac{\beta}{2}\right)^{-a} \cdot c \cdot \left(\frac{\beta}{2}\right)^{-\alpha} \cdot \sum_{i=1}^{\infty} \frac{1}{2^i} \\
&< \quad \left(\frac{\beta}{2}\right)^{-a} \cdot c \cdot \left(\frac{\beta}{2}\right)^{-\alpha} \\
&\overset{\alpha = \lg c / \lg(\beta/2)}{\leq} \quad \left(\frac{\beta}{2}\right)^{-a} .
\end{aligned}
$$

∎

Consider an input element e in the sorting algorithm. Let $x_{i,j}$ be the number of other elements accessing the same memory location as the location accessed by e in the ith step of the search through the binary search fat-tree in the jth level of recursion, $i = 1, \ldots, \lg n_j/2$, $j = 0, \ldots, c_0 \lg\lg n$, where c_0 is chosen so that $c_0 \lg\lg n$ corresponds to the last level of recursion before we switch to bitonic sort.

Lemma 6.3 *There exist $\beta > 2$ and $c > \beta - 1$ such that for $i = 1, \ldots, \lg n_j/2$, $j = 0, \ldots, c_0 \lg\lg n$, $\mathbf{Prob}\,(x_{i,j} \geq u) \leq c\beta^{-u}$ for all $u > 0$.*

Proof. Let n' be the number of elements in the subproblem at the ith level of the binary search fat-tree in the jth level of recursion. W.h.p., $n' \leq 2n_j/2^i$. Also, size of the fat-tree array for the ith level is $2 \cdot \beta n_j/2^i$. We choose $\beta > 2$ and $c = \beta$.

$$
\begin{aligned}
&\mathbf{Prob}\,(x_{i,j} \geq u) \leq C(n', u)(2^i/2\beta n_j)^u \qquad \text{w.h.p.} \\
&\leq ((2n_j/2^i)^u/u!)(2^i/2\beta n_j)^u = (1/u!)(1/\beta)^u \leq c \cdot \beta^{-u}
\end{aligned}
$$

■

We now consider the cumulative delay of any element through fat-trees at all levels of recursion.

Lemma 6.4 *The cumulative delay of any element through fat-trees at all levels of recursion is $O(\lg n)$ w.h.p.*

Proof. Consider an element k in a subproblem in the jth level of recursion. Let $y_i = x_{i,j}$ be the contention of element k in the ith level of the fat-tree in this subproblem. By Lemma 6.3, $\mathbf{Prob}\,(y_i \geq u) < \beta \cdot \beta^{-u}$ (where we have set $c = \beta$). This assumes that the splitters are good, which is true w.h.p. We also assume $\beta/2 > 2$.

The delay of element k through all levels of the fat-tree in the subproblem at the jth level of recursion is $\sum_{i=1}^{\lg n_j} y_i$. Let $\alpha = \lg\beta/\lg(\beta/2)$. Then, by Lemma 6.2

$$\mathbf{Prob}\left((\sum_{i=1}^{\lg n_j} y_i) > (\alpha+2)\lg n_j + a \right) < (\beta/2)^{-(a+2\lg n_j)}$$

Let τ_j be the time for all elements in the subproblem to complete their search through the fat-tree in a subproblem at level j.

$$\mathbf{Prob}\,(\tau_j > \alpha\lg n_j + a) < n_j \cdot (\beta/2)^{-(a+2\lg n_j)} < (\beta/2)^{-(a+\lg n_j)}$$

The cumulative delay for element k through all levels of recursion is $t_k = \sum_{j=1}^{c_0 \lg\lg n} \tau_j$.

Let $\tau'_j = \tau_j - \alpha\lg n_j$. Let $t'_k = \sum_{j=1}^{c_0\lg\lg n} \tau'_j$. Thus

$$t_k = \sum_{j=1}^{c_0\lg\lg n} \tau_j = \sum_{j=1}^{c_0\lg\lg n} \tau'_j + \sum_{j=1}^{c_0\lg\lg n} (\alpha\lg n_j) < (w.h.p.) \quad t'_k + 2\alpha\lg n \ .$$

Now,

$$\mathbf{Prob}\,(\tau'_j > a) < (\beta/2)^{-a}$$

and since $\beta/2 > 2$, we have by Lemma 6.2,

$$\mathbf{Prob}\,(t'_k > \alpha c_0 \lg\lg n + a) < (\beta/4)^{-a}$$

and therefore

$$\mathbf{Prob}\,(t'_k > \alpha c_0 \lg\lg n + b\cdot\lg n) < (\beta/4)^{-b\lg n} < n^{-b}$$

since $\beta > 4$. This implies that

$$\mathbf{Prob}\,(\exists \text{ element } l \text{ s.t. } t'_l > \alpha c_0\lg\lg n + (b+2\alpha)\lg n) < n^{-(b-1)} \ .$$

Thus the cumulative delay of any element through fat-trees at all levels of recursion is $O(\lg n)$ w.h.p. ■

7 Emulating Fetch&Add PRAM on QRQW Asynchronous PRAM

The FETCH&ADD PRAM model [GGK+83, Vis83] is a strong, non-standard variant of the CRCW PRAM. In this model, if two or more processors attempt to write to the same cell in a given step, then their values are added to the value already written in the shared memory location and all prefix sums obtained in the (virtual) serial process are recorded. The FETCH&ADD PRAM is strictly stronger than the standard variants of the CRCW PRAM. Indeed, each step of a (standard) CRCW can be easily emulated by $O(1)$ steps of the FETCH&ADD PRAM, using the same number of processors. On the other hand, the parity and the prefix sums problems with input size n can be solved in constant time on a FETCH&ADD PRAM using n processors, while requiring $\Omega(\lg n/\lg\lg n)$ (expected) time on a CRCW PRAM when using n^c processors, for any constant $c > 0$ [BH89].

In this section we give an emulation of one step of a FETCH&ADD PRAM on a QRQW ASYNCHRONOUS PRAM that takes sub-logarithmic time for moderate contention. Our emulation result is:

Theorem 7.1 *One step of an n-processor* FETCH&ADD PRAM, *and hence of any standard n-processor* CRCW PRAM, *can be emulated on an n-processor* QRQW ASYNCHRONOUS PRAM *in $O(\lg n/\lg\lg n + \lg k)$ time with high probability, where k is the maximum contention (k unknown).*

Proof. Let $s = \lg n/\lg\lg n$ and $q = \sqrt{\lg n \cdot \lg\lg n}$. Assume that the FETCH&ADD PRAM has memory $[1..m]$. In one step of a FETCH&ADD PRAM the processors are partitioned into $n' \leq n$ sets $\Phi_{i_1}, \Phi_{i_2}, \ldots, \Phi_{i_{n'}}$, where each set Φ_{i_j} consists of the processors that read or "write" memory cell $i_j \in [1..m]$. The emulation algorithm deals with each set separately, assuming that each set has an allocated memory of size $M = O(n \cdot 2^q)$. The algorithm is described for one such set, Φ_{i_j}. The same structure is used for both the read step and the write step. In the following we denote the value that each processor in Φ_{i_j} needs to write to cell i_j as its "contents".

A leader for the set Φ_{i_j} is elected by establishing a certain structure among the processors in Φ_{i_j}: this structure enables *combining* the contents of all processors in Φ_{i_j} for the write step, and it enables *broadcasting* the information from memory cell i_j to all processors in Φ_{i_j} in the read step. The combining and broadcasting procedures both use the following structure.

The underlying structure. Consider a full binary tree T on $M = \Theta(n \cdot 2^q)$ leaves (here M is the size of the allocated memory for the set Φ_{i_j}). In the tree T we consider $O(s)$ *hopping levels*: the hopping level i in T is the level containing $\lceil M/s^i \rceil$ nodes. Thus, the leaves are in hopping level 0, and the root is in hopping level $i_r = \lceil \lg M/\lg s \rceil = \Theta(\lg n/\lg\lg n)$. The underlying structure is a hybrid structure over T consisting of a main component and a complementary structure. The main component is a "hopping tree" HT whose nodes are a subset of the nodes in the hopping levels of T. The complementary structure is a "bridging tree" BT,– a subtree of T whose leaves are a subset of the leaves of T and whose root is the root of T.

The trees HT and BT are determined (implicitly) as follows:

INITIALIZATION STEP:

- Each processor in Φ_{i_j} selects at random one leaf v of T, and moves to v.

By "moving" into a node we mean that the processor associates itself to the node (i.e., the node's name is written in a local register), but no other operation is being done. The *bridging tree BT* is defined to be the subtree of T consisting of the leaves that are selected in the initialization step and their ancestors in T. The nodes of the *hopping tree HT* are the nodes of BT that are in the hopping levels of T. The edges of HT are defined as follows: the parent (in HT) of a node in hopping level i is its ancestor in hopping level $i+1$ of T, for $i = 0, \ldots, i_r - 1$. Note that each node in the hopping tree can have up to s children.

The underlying structure through which the actual combining and broadcasting procedures occur is a combination of the hopping tree and the bridging tree. Each processor handles two processes, one for each tree. The idea is to have processes advance from the leaves of T towards its root, and have them combined whenever two or more processes arrive at the same node in T. The bridging tree consists of the nodes that would be visited if each process advances from a node to its parent in T. Such a process would clearly take $\Theta(\lg n)$ steps. The hopping tree is used to accelerate the pace at which processes advance. By moving from one hopping level to the next, a process can reach from a leaf to the root of T in $O(\lg n/\lg\lg n)$ hopping steps. The problem is that hopping steps may take non-constant time, due to contention with other processes for the same hopping nodes, and therefore using the hopping tree all the way to the root may be too expensive. In our method we use hopping steps up to the hopping level that has $k2^s$ nodes (recall that k is the maximum contention of the step), and show that this takes $O(s)$ time; we then proceed from there through the bridging tree, taking $O(\lg(k2^s)) = O(s + \lg k)$ time. The situation, however, is complicated by the fact that k is unknown, so the hopping level i_k is unknown. The combining procedure presented next uses both the hopping tree HT and the bridging tree BT in parallel so as to enable this combination to occur without relying on a knowledge of k.

The combining procedure

We would like each processor to advance from its selected leaf in T towards the root of T. The combining algorithm consists of *hopping steps* and *bridging phases*. In the hopping steps processors advance one level on the hopping tree. In the bridging phase processors advance $O(\lg s)$ levels on the bridging tree (from one hopping level up to the next).

The hopping step. At each step of the algorithm, a processor may try to write into some node v in HT. Let $\Phi(v)$ be the set of processors that try to write into the node v. If v is a leaf then $\Phi(v)$ is the set of processors that selected v in the initialization step. If v is an internal node then $\Phi(v)$ is the set of processors that have previously accessed nodes that are a children of v in HT. Thus, for an internal node v, $|\Phi(v)|$ is the number of children of v in HT. A winner from $\Phi(v)$ can be selected in constant time, and in $O(|\Phi(v)|)$ time all the processors in $\Phi(v)$ can know the identity of the winner. The contents of the processors in $\Phi(v)$ can be combined into a single word in $O(|\Phi(v)|^2)$ time; within the same time, a list of all processors in $\Phi(v)$ may be computed and stored at v. Winners in hopping level i proceed to the next hopping level $i+1$ with the combined contents; the winner of a set $\Phi(v)$ will now try to write into the parent of v in HT. The hopping step thus consists of the following substeps:

- *select a winner:* Each processor $P \in \Phi(v)$ attempts to claim the node v using the Test&Set primitive; the winner records its index with v.
- move the winner to the parent of v in HT.
- *combine data and create list:* repeat $|\Phi(v)|$ times:
    1. *select a winner*;
    2. *combine data:* the winner adds its contents into the combined data;
    3. *append list:* the winner adds its index to $list(v)$;
    4. remove the winner from $\Phi(v)$.

Note that the winner moves into the next level in HT in one step. It takes however $O(|\Phi(v)|^2)$ time to combine the data and create a list of processors, due to the contention. A slight complication arises when taking the asynchrony into account. Some processors of $\Phi(v)$ may arrive at v after other processors have already combined their data. Late arrivals are handled separately; thus, arrival time partitions $\Phi(v)$ into subsets $\Phi^1(v), \Phi^2(v), \ldots$ such that all processors in $\Phi^j(v)$ arrive at v after all processors in $\Phi^{j-1}(v)$ have finished their hopping step. For each subset there is a winner that proceeds to the next level in HT.

The bridging phase. Processors can also advance in the bridging tree similarly to the hopping tree: a winner at a node u will try to access in the next step the parent v of u in BT. Let $\Phi'(v)$ be the set of processors that try to write into the node v in BT. For each node $v \in BT$ we have $|\Phi'(v)| \leq 2$, and therefore advancing by one level at the bridging tree takes constant time. As a result, a node in hopping level i will receive the combined contents of all its children in hopping level $i-1$ (in HT) after $O(\lg s)$ steps; denote this as a *bridging phase*. Clearly, a bridging phase is faster than a hopping step for nodes v such that $|\Phi(v)| > \sqrt{\lg s}$. Asynchrony is handled in a manner similar to the method used in the hopping step.

The combined strategy. During the combining procedure we do not know in advance the values of $|\Phi(v)|$, $v \in HT$, and we therefore cannot predetermine whether HT or BT should be used from one hopping level to the other. Therefore, we let the combining procedure advance in parallel in both trees HT and BT. The combined contents may eventually arrive at the node v in two copies, one through HT and one through BT. Due to the asynchrony among the processors, we are able to proceed when the first copy arrives, without waiting for the other copy to arrive. To implement the hybrid strategy, we let a winner in a node u in hopping level i spawn into two processes. One will try to advance by a hopping step directly to u's parent v in HT (in hopping level $i+1$), and the other will advance by a bridging phase in the bridging tree BT. When either process arrives at the node v, it first checks if the other process has already arrived at this node. If this is the case, the process halts; otherwise, it marks its arrival and goes on with the combining algorithm.

The spawning technique should be controlled carefully as it is implemented by only a single processor as it advances through the two trees. Whenever a hopping step terminates before the corresponding bridging phase, the terminating process halts the bridging phase and retrieves the second process for the next hopping step. Note that in the bridging phase, a processor only accesses memory cells with constant contention, hence retrieving the second process takes constant time. We note that this is not quite the case with the processor implementing the hopping step, since an access with high contention (up to s) may occur. However, it is shown in the analysis below that the probability for the bridging phase to terminate before the hopping step is negligible, when the process is at a hopping level that contains at least $k \cdot 2^s$ nodes. Therefore, once the bridging phase terminates before the hopping step we assume that we are past the hopping level i and we go on with the bridging phases only; we do not wait for the hopping step to terminate.

There is one issue to be taken care of regarding the halting of a bridging process. Such a process may be already combined with another process in the bridging tree. Due to asynchrony, the other process may get to the next hopping level before its corresponding hopping step has terminated. To prevent possible confusion, whenever a hopping step terminates before the corresponding bridging phase, we make sure that all the hopping steps into the same node will also be considered as if they had terminated before their corresponding bridging phases. This is easy to implement for processes that have not yet terminated, by leaving an appropriate mark at the node. However, there may be processes for which both the hopping step and bridging phase have already terminated, with the latter being first. To handle such processes, we keep a list of all processes that have terminated at a node; whenever a hopping step terminates, it appends itself to this list. This list will enable a processor to notify all the appropriate processors about their new status; they will learn it when needed.

Analysis

Consider a hopping level i with $n_i = k \cdot 2^s$ nodes, and let $\tilde{\Phi}_i = \{P \in \Phi(v) : v \text{ in hopping level } i\}$; i.e., $\tilde{\Phi}_i$ is the set of processors that try to write into any node in hopping level i. After the processors in $\tilde{\Phi}_i$ arrive at hopping level i, the combining procedure for these processors will take $O(\lg n_i) = O(\lg k + s)$ time, using the bridging phases over the tree BT, and thus the computation will terminate in $O(\lg k + s)$ steps after arriving at hopping level i. In the following we analyze the time needed to arrive at hopping step i.

If $i = 0$ then the hopping level i corresponds to the leaves of the hopping tree. Each of the k processors chooses a random leaf. Consider a fixed leaf l, and consider a sequence of Bernoulli trials, one for each processor in Φ_{i_j}, where the trial is a success if the processor chooses leaf l and is a failure otherwise. The probability of success is $1/M$ and hence the expected number of processors that move into leaf l is $k/M \le 1/2^q$ since $k \le n$. Hence by Lemma 2.1, the number of processors that move into leaf l is $O((\lg n)/q) = O(\sqrt{s})$ w.h.p. It follows that the number of processors that move into any given leaf is $O(\sqrt{s})$ w.h.p., and hence the combining step at all leaves is executed in $O(s)$ time w.h.p.

For the rest of the analysis we assume that $i > 0$, i.e., the hopping level is strictly above the leaves. Recall that the time taken by a processor to advance at some node w is $\deg(w)^2$, where $\deg(w)$ is the number of children of w in HT. For a node v in HT, let L_v be the set of leaves of the sub-tree of HT rooted at v, and let $X_v = |L_v|$. Consider a path p from a node v to a leaf u in L_v. For every pair of nodes w_1 and w_2 that are not on the path p but their parents are in p, the sets L_{w_1} and L_{w_2} do not have any common leaf. Therefore, the total time it takes a processor to advance from the leaf u to node v along the path p is

$$\sum_{w \in p} \deg(w)^2 = \left(\sum_{w \in p} (\deg(w) - 1)^2 \right) + \left(\sum_{w \in p} 2\deg(w) \right) + |p| \le X_v^2 + 2X_v + O(\lg n / \lg\lg n) \ .$$

X_v is a random variable whose outcome is determined in the initialization step. Specifically, if v is in hopping level i then each processor in $\tilde{\Phi}_i$ selects a leaf in L_v with probability $1/n_i$. Since $|\tilde{\Phi}_i| \le k$, X_v is stochastically smaller than a binomial variable Y_v on a sequence of k Bernoulli trials with probability of success $1/n_i$. Since $E[Y_v] = 1/2^s$, by Lemma 2.1, $Y_v = O((\lg n)/s) = O(\lg\lg n)$ w.h.p. Thus $X_v = O(\lg\lg n)$ w.h.p.

Therefore, with high probability all processors in $\tilde{\Phi}_i$ arrive at hopping level i in $O(s + \lg\lg n)$ time, i.e., in $O(s)$ time. Moreover, if a bridging phase terminates before its corresponding hopping step then with high probability the processes have arrived at hopping level i for which $n_i \le k \cdot 2^s$, as required.

The broadcasting procedure

For the (standard) CRCW PRAM a read step is executed by broadcasting the datum at memory cell i_j to all the processors in Φ_{i_j}. The broadcasting procedure uses the structure that was built in the combining procedure, and is essentially based on reversing the execution of the combining procedure, broadcasting the data backwards from the root of T to the leaves. The data is broadcasted from hopping level i to hopping level $i - 1$ using either the analog to the hopping step, or the analog to a bridging phase, depending on which has terminated earlier during the combining procedure (this information can be recorded).

In case the hopping step is selected, the broadcasted datum x will be written at a node v by the winner of $\Phi(v)$. Then, the processors in $\Phi(v)$ read x, and each processor $P \in \Phi(v)$ writes x at the node u, the child of v in HT, in which P was a winner in the combining procedure. Letting the processors of $\Phi(v)$ probe a shared memory cell in v in search for the broadcasted datum x may be too costly. Instead, we let each processor have a designated register into which x will be written by the winner of $\Phi(v)$. Thus, each processor in $\Phi(v)$ can repeatedly probe its designated register with only constant time cost. The list $list(v)$ of $\Phi(v)$, computed at the hopping step of the combining procedure and kept at the node v, will enable the winner of $\Phi(v)$ to distribute x to the processors in $\Phi(v)$ in $O(|\Phi(v)|)$ time.

Broadcasting for the Fetch&Add PRAM. Recall that in the FETCH&ADD PRAM model, processors are ranked in arbitrary order, a prefix sums sequence of the written values together with the value x (in the memory cell) is computed, and each processor receives its appropriate prefix sum. It is straightforward to see that broadcasting the appropriate prefix sums is the reversal of the combining procedure with minor modifications, except for the fact that now the list of processors is already given and the time complexity is therefore $O(|\Phi(v)|)$. As in the combining procedure, asynchrony adds some complications due to possible late arrivals of processors. We note however that processors that arrive late can be given larger rank in the prefix sums sequence; hence their value will not affect the values that are broadcasted to processors with smaller rank. Late arrivals can therefore be handled in the broadcasting step similar to the way they are handled in the combining step.

The theorem follows. ∎

We note that the one-step emulation above cannot be used directly for multi-step emulation since a synchronization barrier is required after each step. It is an interesting open problem to see whether it may become useful for such emulation.

8 Discussion

The cost metric for the QRQW ASYNCHRONOUS PRAM is tailored towards ease of use and is meant to model asynchronous systems in which processors run at more or less the same rate. On the other hand, algorithms must be correct under worst case assumptions on the finite delays incurred by processors and in processing memory requests. This separation of correctness from analysis, with correctness accounting for asynchrony but analysis assuming synchrony, was pioneered by Gibbons [Gib89] and has been subsequently adapted by several other asynchronous models such as the LogP model [CKP+93].

As an example of the ease of designing algorithms under such a cost metric consider designing an algorithm to find the maximum of n numbers on an asynchronous parallel machine. On the QRQW ASYNCHRONOUS PRAM we have a simple linear work algorithm using $n/\lg n$ processors that basically mimics the standard EREW PRAM algorithm. Each processor works on a block of $\lg n$ elements in the input and finds their maximum. The processors then cooperate to compute the maximum in a 'binary tree' computation. In case a value that is wanted by a processor is not yet available the processor waits for that value to be written. Under the QRQW ASYNCHRONOUS PRAM cost metric this algorithm takes $2\lg n$ time. The algorithm is simple with low overheads, and correct regardless of any asynchrony among the processors of the asynchronous parallel machine. In case of small delays among the machine processors the running time will increase only slightly.

This algorithm can be contrasted with algorithms designed using asynchronous models whose cost metrics account for more general asynchrony among the processors. For example, Martel et al. [MPS92] describe a randomized algorithm to compute the maximum of n numbers on their A-PRAM model, a model whose cost metric accounts for worst case asynchrony. Despite assuming a more powerful CRCW contention rule, the expected running time of this algorithm is greater than $10 \cdot \lg n$ time even if all processors execute at the same rate. This extra overhead is due to designing for a cost metric that accounts for worst case asynchrony. The advantage of this algorithm arises only in cases of very large delays among the machine processors.

Most asynchronous models, e.g. [CZ89, CZ90, Nis90, And92, AR92, DHW93, LAB93], account for more general asynchrony among the processors in their cost metrics, and hence algorithms designed using these models suffer from similar overheads in order to more robustly handle cases with very large delays.

We should point out that the QRQW ASYNCHRONOUS PRAM cost metric is open to abuse as shown by the following example involving two processors $P1$ and $P2$:

$P1$	$P2$
$x := 0$	$y := 0$
$x := 1$	$y := 1$
if $(x = y)$ then do short computation	
else do very long computation	

Here both the short computation and the very long computation produce the same (correct) output. Under the cost metric for the QRQW ASYNCHRONOUS PRAM, this computation will take only a short time. However, if processor $P2$ is delayed more than processor $P1$ then the test $x = y$ will return false resulting in a very long computation. None of the algorithms we have presented in this paper have this property of transforming into a dramatically slower algorithm if different processors encounter slightly different delays. However the above example shows that it is possible to design such algorithms under our cost metric. It would be interesting to come up with a cost metric that penalizes large changes in running time in the presence of small delays while at the same time retaining the advantages of our current cost metric.

9 Conclusions

In this paper we have defined the QRQW ASYNCHRONOUS PRAM and presented some algorithmic results for the model. In particular, we have shown two instances in which we have better algorithms for the QRQW ASYNCHRONOUS PRAM than those known for the QRQW PRAM. The first is for computing the OR of n bits for which we described a simple deterministic linear work algorithm that runs in $O(\lg n/\lg\lg n)$ time; we also showed that this bound is tight. In contrast, no deterministic sub-logarithmic time algorithm for this problem is known for the QRQW PRAM. The second result is an implementation of the randomized sample sort algorithm that runs in $O(\lg n)$ time and $O(n \lg n)$ work on the QRQW ASYNCHRONOUS PRAM; the fastest implementation known for this algorithm on the QRQW PRAM runs in $O(\lg^2 n/\lg\lg n)$ time. We have also shown adaptations of several QRQW PRAM algorithms to the QRQW ASYNCHRONOUS PRAM with the same work-time bounds and a simulation of a FETCH&ADD PRAM on the QRQW ASYNCHRONOUS PRAM.

Additional results for the QRQW ASYNCHRONOUS PRAM can be found in a recent paper by Adler [Adl96]. That paper presents a number of new results on low-contention search structures, beyond the binary search fat-tree considered in this paper.

One interesting direction for future work is to develop a good emulation of the QRQW ASYNCHRONOUS PRAM on a distributed memory machine model such as the BSP. In [GMR96b] we presented an optimal work emulation of the QRQW PRAM on the BSP with only a logarithmic slowdown. It appears that the strategy used in that emulation does not carry over directly to the QRQW ASYNCHRONOUS PRAM and new insights are needed. Alternatively, one could consider developing good emulation results by imposing suitable restrictions on the QRQW ASYNCHRONOUS PRAM.

References

[ABM93] Y. Afek, G. M. Brown, and M. Merritt. Lazy caching. *ACM Trans. on Programming Languages and Systems*, 15(1):182–205, 1993.

[Adl96] M. Adler. Asynchronous shared memory search structures. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 42–51, June 1996.

[AJ96] C. Armen and D. B. Johnson. Deterministic leader election on the Asynchronous QRQW PRAM. *Parallel Processing Letters*, 1996. To appear.

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

[And92] R. J. Anderson. Primitives for asynchronous list compression. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 199–208, June-July 1992.

[AR92] Y. Aumann and M. O. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 147–156, October 1992.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Summer Computer Conference*, pages 307–314, 1968.

[BGMZ95] G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 84–94, July 1995.

[BH89] P. Beame and J. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, 36(3):643–670, July 1989.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 1–12, May 1993.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[CZ89] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proc. 1st ACM Symp. on Parallel Algorithms and Architectures*, pages 169–178, June 1989.

[CZ90] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, pages 85–94, July 1990.

[DHW93] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 174–183, May 1993.

[DKR94] M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk. Exact lower time bounds for computing boolean functions on CREW PRAMs. *Journal of Computer and System Sciences*, 48(2):231–254, 1994.

[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.

[Gib89] P. B. Gibbons. A more practical PRAM model. In *Proc. 1st ACM Symp. on Parallel Algorithms and Architectures*, pages 158–168, June 1989. Full version in *The Asynchronous PRAM: A semi-synchronous model for shared memory MIMD machines*, PhD thesis, U.C. Berkeley 1989.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th International Symp. on Computer Architecture*, pages 15–26, May 1990.

[GM92] P. B. Gibbons and M. Merritt. Specifying nonblocking shared memories. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 306–315, June-July 1992.

[GMG91] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[GMR93] P. B. Gibbons, Y. Matias, and V. Ramachandran. QRQW: Accounting for concurrency in PRAMs and Asynchronous PRAMs. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, March 1993.

[GMR96a] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. *Journal of Computer and System Sciences*, 1996. To appear. Preliminary version appears in *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 236-247, June 1994.

[GMR96b] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 1996. To appear. Preliminary version appears in *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 638-648, January 1994.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A*, pages 869–941. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[LAB93] P. Liu, W. Aiello, and S. Bhatt. An atomic model for message-passing. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 154–163, June-July 1993.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[MPS92] C. Martel, A. Park, and R. Subramonian. Work-optimal asynchronous algorithms for shared memory parallel computers. *SIAM Journal on Computing*, 21(6):1070–1099, 1992.

[Nis90] N. Nishimura. Asynchronous shared memory parallel computation. In *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, pages 76–84, July 1990.

[Rei85] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, 14(2):396–409, May 1985.

[Rei93] J. H. Reif, editor. *A Synthesis of Parallel Algorithms.* Morgan-Kaufmann, San Mateo, CA, 1993.

[Smi95] B. Smith. Invited lecture, *7th ACM Symp. on Parallel Algorithms and Architectures*, July 1995.

[Vis83] U. Vishkin. On choice of a model of parallel computation. Technical Report 61, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012, 1983.