Workshop 12 (16)

Theory and Models for Parallel Computing

The Queue-Read Queue-Write Asynchronous PRAM Model

Phillip B. Gibbons* Yossi Matias* Vijaya Ramachandran**

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 family of models, introduced earlier by the authors.

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

* Bell Laboratories, 600 Mountain Avenue, Murray Hill NJ 07974; {gibbons,matias}@research.bell-labs.com.

** Dept. of Computer Sciences University of Texas at Austin Austin TX 78712; vlr@cs.utexas.edu. Supported in part by NSF grant CCR-90-23059 and Texas Advanced Research Projects Grant 003658480.

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. 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).

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 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 one 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 $O(\lg n/\lg\lg n + k)$ time w.h.p., 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.

Due to space constraints, many of the details are omitted and can be found in the full paper [GMR96c].

2 The QRQW Asynchronous PRAM

In this section, we present the definition of the QRQW ASYNCHRONOUS PRAM model, and some observations on the algorithmic power of the model.

A variety of ASYNCHRONOUS PRAM models have been studied in the literature (c.f. [Gib89, CZ89, MPS92, Nis90, And92]). These models account for contention in a manner most like a CRCW PRAM, with no penalty assessed for large contention to a location.[3] An EREW contention rule was not considered, 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. Instead, we define what we believe to be a better model for asynchronous parallel machines.

2.1 The 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 real multiprocessor is that no request is delayed indefinitely. (We

[3] 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 address as for an interleaving of reads/writes to different addresses.

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 (e.g. [CZ89, MPS92, Nis90, And92, DHW93]).[4] 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. A formal definition of a sequentially consistent shared memory that permits pipelining is given by Gibbons and Merritt [GM92].

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 philosophy 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 according to 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.

[4] Note that when all memory accesses take unit time in a model, there is no need for pipelining.

Model definition. The QRQW ASYNCHRONOUS PRAM model consists of a collection of processors operating asynchronously and communicating via a global shared memory. The shared memory is sequentially consistent and supports the pipelining of memory requests by processors (i.e. each processor is permitted to have multiple pending shared memory requests; see [GM92] for a formal definition). 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. Algorithms must be correct under worst case assumptions on the finite delays incurred by processors and in processing memory requests.

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. A processor can not make inferences on the queue length encountered based on the delay incurred. 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 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.

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 interesting 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 are proposed as alternatives to other PRAM models for high-level algorithm design.

2.2 Preliminary observations

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 simulation 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.

It turns out that many of the algorithms we developed for the QRQW PRAM algorithms in [GMR96b, GMR96a] can be adapted to the QRQW ASYNCHRONOUS PRAM to run with the same work-time bounds. Details of these results can be found in [GMR96c]. In particular, we mention here that the QRQW PRAM algorithm for *multiple compaction* can be adapted to the QRQW ASYNCHRONOUS PRAM to run with the same bounds of linear work and logarithmic time. This result is used in Section 4.

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. In the remainder of this paper, we discuss three such examples.

3 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 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 linear 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 a s-ary tree, T, with one leaf per processor, with each location corresponding to a node in T initialized to zero. Each processor that elected a leader 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. 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. The proof of the following lemma is omitted due to space limitation. It can be found in [GMR96c].

Lemma 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* $O(t)$.

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

Theorem 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 2, the OR function can be computed on the T-write PRAM in $O(T)$ time. Thus $T \in \Omega(\lg n/\lg T)$, and hence $T \in \Omega(\lg n/\lg\lg n)$. ∎

4 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.

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

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 "heavy" multiple compaction [GMR96a]. W.h.p., the number of items with the same label is at most n_{i+1} and thus the heavy

multiple compaction succeeds in placing all items in each such group into its designated subarray.

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 [JáJ92]. This cut-off point suffices for n sufficiently large; for general n, the cut-off point is $\max\left\{2^{(\lg n)^{1/2}}, \lg^c n\right\}$, for $c > 6/\epsilon$ a suitable constant.

In step 4 we use a heavy multiple compaction algorithm which reports failure if a set size exceeds its upper bound count [GMR96a]. If failure is reported for any subproblem, we restart the algorithm from the beginning.

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. To obtain our result, we show that for each element, the sum of the contentions it encounters during the binary search process is $O(\lg n)$, as shown below.

Lemma 4. *Let $\beta > 2$, $c \geq \beta - 1$, $a \geq 0$, and $\alpha = \lg c / \lg(\beta/2)$. Let $x_1, \ldots, x_m$ be independent random variables over the positive integers so that $\mathbf{Pr}(x_i = u) \leq c\beta^{-u}$ for all $u > 0$. Let $S_m = x_1 + \cdots + x_m$, for $m \geq 1$ and $S_0 = 0$. Then,*

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

Proof. The proof is by induction on m.

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

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

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

■

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

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

$$\mathbf{Pr}(x_{i,j} \geq u) \leq C(n', u)(1/2\beta n_j)^u \quad \text{w.h.p.}$$

$$\leq ((2n_j)^u/u!)(1/2\beta n_j)^u = (1/u!)(1/\beta)^u \leq c \cdot \beta^{-u}$$

■

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

Lemma 6. *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 5, $\mathbf{Pr}(y_i \geq u) < \beta \cdot \beta^{-u}$ (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 4

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

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.

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

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,

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

and by Lemma 4, we have

$$\mathbf{Pr}(t'_k > \alpha c_0 \lg\lg n + a) < (\beta/2)^{-a}$$

and therefore

$$\mathbf{Pr}(t'_k > \alpha c_0 \lg\lg n + b \cdot \lg n) < (\beta/2)^{-b\lg n} < n^{-b}$$

since $\beta > 4$. This implies that

$$\mathbf{Pr}(\exists t_l > \alpha c_0 \lg\lg n + (b+2\alpha)\lg n) < n^{-(b-1)} \ .$$

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

Hence the QRQW ASYNCHRONOUS PRAM sorting algorithm terminates in $O(\lg n)$ time w.h.p.

5 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.

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

Theorem 7. *One step of an n-processor* FETCH&ADD PRAM, *and hence of an 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).*

The emulation algorithm can be found in the full paper [GMR96c].

6 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 problem on the QRQW PRAM runs in $O(\lg^2 n/\lg\lg n)$ time. Full details of these results and others reported in this paper, including adaptation 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, may be found in [GMR96c].

Additional results for the QRQW ASYNCHRONOUS PRAM can be found in a recent paper by Adler [Adl96]. That paper presents interesting 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 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*, June 1996. To appear.

[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.

[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.

[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.

[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.

[GMR96c] P.B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. Technical report, Bell Laboratories, Murray Hill, NJ, and Dept. of Comp. Sci., Univ. of Texas, Austin, TX, June 1996.

[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.