The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms

Phillip B. Gibbons
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974
gibbons@research.att.com

Yossi Matias
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974
matias@research.att.com

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

May 21, 1996

Abstract

This paper introduces the *queue-read, queue-write* (QRQW) parallel random access machine (PRAM) model, which permits concurrent reading and writing to shared memory locations, but at a cost proportional to the number of readers/writers to any one memory location in a given step. Prior to this work there were no formal complexity models that accounted for the contention to memory locations, despite its large impact on the performance of parallel programs. The QRQW PRAM model reflects the contention properties of most commercially available parallel machines more accurately than either the well-studied CRCW PRAM or EREW PRAM models: the CRCW model does not adequately penalize algorithms with high contention to shared memory locations, while the EREW model is too strict in its insistence on zero contention at each step.

The QRQW PRAM is strictly more powerful than the EREW PRAM. This paper shows a separation of $\sqrt{\lg n}$ between the two models, and presents faster and more efficient QRQW algorithms for several basic problems, such as linear compaction, leader election, and processor allocation. Furthermore, we present a work-preserving emulation of the QRQW PRAM with only logarithmic slowdown on Valiant's BSP model, and hence on hypercube-type non-combining networks, *even when latency, synchronization, and memory granularity overheads are taken into account.* This matches the best known emulation result for the EREW PRAM, and considerably improves upon the best known efficient emulation for the CRCW PRAM on such networks. Finally, the paper presents several lower bound results for this model, including lower bounds on the time required for broadcasting and for leader election.

*Supported in part by NSF grant CCR-90-23059 and Texas Advanced Research Projects Grants 003658480 and 003658386.

1 Introduction

The Parallel Random Access Machine (PRAM) model of computation is the most-widely used model for the design and analysis of parallel algorithms (see, e.g. [KR90, JáJ92, Rei93]). The PRAM model consists of a number of processors operating in lock-step and communicating by reading and writing locations in a shared memory. Existing PRAM models can be distinguished by their rules regarding contention for shared memory locations. These rules are generally classified into two groups:

- **Exclusive** *read/write*: Each location can be read or written by at most one processor in each unit-time PRAM step.
- **Concurrent** *read/write*: Each location can be read or written by any number of processors in each unit-time PRAM step. For concurrent writing, the value written depends on the *write-conflict rule* of the model, e.g. in the *arbitrary concurrent-write* PRAM, an arbitrary processor succeeds in writing its value.

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 this paper, we argue that neither the *exclusive* nor the *concurrent* rules accurately reflect the contention capabilities of most commercial and research machines, and propose a new PRAM contention rule, the *queue* rule, that permits concurrent reading and writing, but at an appropriate cost:

- **Queue** *read/write*: Each location can be read or written by any number of processors in each step. 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.

The queue rule more accurately reflects the contention properties of machines with simple, non-combining interconnection networks[1] than either the exclusive or concurrent rules. The exclusive rule is too strict, and the concurrent rule ignores the large performance penalty of high contention steps. Indeed, for most existing machines, including the CRAY T3D, IBM SP2, Intel Paragon, MasPar MP-2 (global router), MIT J-Machine, nCUBE 2S, Stanford DASH, Tera Computer, and Thinking Machines CM-5 (data network), the contention properties of the machine are well-approximated by the queue-read, queue-write rule. For the Kendall Square KSR1, the contention properties can be approximated by the concurrent-read, queue-write rule. Further details are in Section 3.

This paper defines the queue-read, queue-write (QRQW) PRAM model, a variation on the standard PRAM that employs the queue rule for both reading and writing. In addition, the processors are permitted to each have multiple reads or writes in progress at a time. We show that the power of the QRQW PRAM model falls strictly between the CRCW and EREW models. We show separation results between the models by considering the 2-compaction problem, the broadcasting problem, and the problem of computing the OR function. To illustrate some of the techniques used to design low-contention algorithms that improve upon the best known zero-contention algorithms, we consider algorithms for two fundamental problems, *leader election* and *linear compaction*, under various scenarios. Finally, this paper extends the work-time framework for parallel algorithms (see, e.g. [JáJ92]) into a QRQW work-time framework that considers the contention at each step, and relates the QRQW PRAM model to the QRQW work-time framework.

[1] In a *combining network*, when two messages destined for the same memory location meet at an intermediate node in the network, the messages are "combined" so that only one message continues towards the destination. For example, if two writes meet, then only a single write is sent on. In a *non-combining network*, messages are not combined, so that *all* messages destined for the same memory location are delivered to the home node for that location.

The QRQW PRAM, like the other PRAM models mentioned above, abstracts away many features of real machines, including the latency or delay in accessing the shared memory, the cost of synchronizing the processors, and the fact that memory is partitioned into modules that service requests serially. A model that incorporates these features is the Bulk-Synchronous Parallel (BSP) model of Valiant [Val90a]. In its general form this model is parameterized by its number of processing/memory *components* p, *throughput* g, and *periodicity* L. A particular case studied by Valiant sets g to be a constant and L to be $\Theta(\lg p)$; we denote this the *standard* BSP model. We show in this paper that the QRQW PRAM can be effectively emulated on the standard BSP model: A p-processor QRQW PRAM algorithm running in time t can be emulated on a $p/\lg p$-processor standard BSP in $O(t \lg p)$ time with high probability. It follows by Valiant's simulation of the standard BSP on hypercubes that the QRQW PRAM can be emulated in a work-preserving manner on parallel machines with hypercube-type, non-combining networks with only logarithmic slowdown, *even when latency, memory granularity, and synchronization overheads are taken into account.* This matches the best known emulation for the EREW PRAM on these networks given in [Val90a]. In contrast, work-preserving emulations for the CRCW PRAM on such networks are only known with *polynomial* slowdown (i.e. $O(p^\epsilon)$ slowdown, for a constant $\epsilon > 0$).

Note that the standard $\Theta(\lg p)$ time emulation of CRCW on EREW (see, e.g. [KR90]) is not work-preserving, in that the EREW performs $\Theta(\lg p)$ times more work than the CRCW it emulates. Since we consider work-preserving speed-ups to be the primary goal in parallel algorithms, with fast running times the secondary goal, this emulation is unacceptable: The $\Theta(\lg p)$ overhead in work ensures that the algorithms will not exhibit linear or near-linear speedups. Similarly, the best known emulations for the CREW PRAM (or ERCW PRAM) on the EREW PRAM (or standard BSP or hypercube) require logarithmic work overhead for logarithmic slowdown or, alternatively, polynomial slowdown for constant work overhead.

Since the QRQW PRAM is strictly more powerful than the EREW PRAM, effectively emulated on hypercube-type non-combining networks (unlike the CRCW, CREW, or ERCW PRAM models), and a better match for real machines, we advocate the QRQW PRAM with its *queue* contention rule as a more appropriate model for high-level algorithm design than a PRAM with either the *exclusive* or *concurrent* contention rules. The queue contention rule can also be incorporated into lower-level shared memory models, trading model simplicity for additional accuracy in modeling the cost of communication (e.g. explicitly modeling the communication bandwidth). In this initial paper on the queue contention rule, we restrict our focus to high-level algorithm design on PRAM models.

In addition to the QRQW PRAM model, we define in this paper the SIMD-QRQW PRAM model, a strictly weaker model suitable for SIMD machines, in which all processors execute in lock-step and each processor can have at most one read/write in progress at a time. In a subsequent paper [GMR96b] we define the QRQW ASYNCHRONOUS PRAM model, for general asynchronous algorithms running on MIMD machines (see also [GMR93]).

We present several algorithms and a lower bound for leader election and for computing the OR function. The lower bound is $\Omega(\lg n/\lg\lg n)$ time for the deterministic computation of the OR function on a concurrent-read, queue-write (CRQW) PRAM with arbitrarily many processors. The algorithms for both problems take linear work, $O(\lg n/\lg\lg n)$ time with high probability. In contrast, the OR function requires $\Omega(\lg n)$ expected time on a randomized CREW PRAM with arbitrarily many processors ([DKR94], following [CDR86]). Also presented is a linear work, $O(\sqrt{\lg n})$ time w.h.p. algorithm for the linear compaction problem. This problem has applications to automatic processor allocation for algorithms that are given in the QRQW work-time presentation. In contrast, the best linear compaction algorithm known on the EREW PRAM is the logarithmic time prefix sums algorithm [LF80]. On the other hand, for the problem of broadcasting the value of a bit to n processors, we show that we can do no better on the QRQW PRAM than the simple $\Theta(\lg n)$ time EREW PRAM algorithm. Specifically, we show a tight $\Omega(\lg n)$ expected time lower bound for the QRQW PRAM.

Important technical issues arise in designing algorithms for the queue models, that are present in neither the concurrent nor the exclusive PRAM models. For example, much of the effort in designing algorithms for the QRQW is in estimating the maximum contention in a step; our algorithms for leader election illustrate this point. In the QRQW, one high contention step can dominate the running time of the algorithm: we cannot afford to underestimate the contention significantly.

In a companion paper [GMR96a], we present a number of other algorithmic results for the QRQW PRAM. Our algorithmic results include linear work, logarithmic or sublogarithmic time randomized QRQW algorithms for the fundamental problems of multiple compaction, load balancing, generating a random permutation, parallel hashing, and sorting from $U(0,1)$. These algorithms improve upon the best known EREW algorithms for these problems, while avoiding the high-contention steps typical of CRCW algorithms. Additionally, we present new algorithms for integer sorting and general sorting.

Most of the results in [GMR96a], and some of the results in this paper, are obtained "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 $1 - 1/n^{\delta}$.

The rest of this paper is organized as follows. Section 2 defines the QRQW PRAM and SIMD-QRQW PRAM models. Section 3 gives further motivation for the queue models, and comparison with related work. Section 4 describes the extension of the work-time framework to the QRQW models. Section 5 presents our results for realizing the QRQW PRAM on feasible networks. Section 6 gives upper and lower bounds for computing the OR and leader election under various scenarios. Section 7 presents our linear work, sublogarithmic time algorithm for linear compaction on a SIMD-QRQW PRAM. Section 8 presents tight $\Omega(\lg n)$ expected time lower bounds on the QRQW PRAM for broadcasting and related problems. Concluding remarks appear in Section 9.

The results in this paper appeared in preliminary form in [GMR93, GMR94a, GMR94b].

2 The queue models

This section defines our two queue-read queue-write (QRQW) models:

- The SIMD-QRQW PRAM, for algorithms running on SIMD machines.
- The QRQW PRAM, for bulk-synchronous algorithms[2] running on MIMD machines.

In both of the QRQW models, the time cost for reading or writing a shared location, x, is proportional to the number of processors concurrently reading or writing x. This cost measure models machines in which accesses to a location *queue* up and are serviced one at a time, i.e. most current commercial and research machines. The SIMD-QRQW models machines in which processors synchronize at every step, awaiting for all the queues to clear. The QRQW models machines in which processors synchronize less frequently, awaiting for all the queues to clear only at synchronization points. In a subsequent paper [GMR96b] we define the QRQW ASYNCHRONOUS PRAM model, for general asynchronous algorithms running on MIMD machines (see also [GMR93]). This model has an *asynchronous* queue contention rule in which processors read and write locations at their own pace, without waiting for the queues encountered by other processors to clear. This model allows

[2]In a *bulk-synchronous* algorithm [Val90a, Gib89, Gib93], synchronization among the processors is limited to global synchronization barriers involving all the processors; between such barriers, processors execute asynchronously using shared memory values written prior to the preceding barrier.

the asynchronous nature of MIMD machines to be exploited, at the cost of more complexity in the model.

The complexity metric for the QRQW models will use the notion of maximum contention, defined as follows.

Definition 2.1 *Consider a single step of a* PRAM*, consisting of a read substep, a compute substep, and a write substep. The* **maximum contention** *of the step is the maximum, over all locations x, of the number of processors reading x or the number of processors writing x. For simplicity in handling a corner case, a step with no reads or writes is defined to have maximum contention 'one'.*

2.1 The SIMD-QRQW PRAM model

Definition 2.2 *The* **SIMD-QRQW PRAM** *model is a (synchronous)* PRAM *in which concurrent reads and writes to the same location are permitted, and the* **time cost** *for a step with maximum contention κ is κ. If there are multiple writers to a location x in a step, an arbitrary write to x succeeds in writing the value present in x at the end of the step. The* **time** *of a* SIMD-QRQW PRAM *algorithm is the sum of the time costs for its steps. The* **work** *is its processor-time product.*

This cost measure models, for example, a SIMD machine such as the MasPar MP-1 [Mas91] or MP-2, in which each processor can have at most one read/write in progress at a time, reads/writes to a location *queue* up and are serviced one at a time, and all processors await the completion of the slowest read/write in the step before continuing to the next step. Existing SIMD machines provide for the required synchronization of all processors at each step, regardless of the varying contention encountered by the individual processors. Unlike previous PRAM models, the work is not the number of operations, because with the SIMD-QRQW time metric, operations encountering non-constant contention are charged non-constant time.

If a PRAM model is to be used to design bulk-synchronous algorithms on MIMD machines, then the SIMD-QRQW PRAM is unnecessarily restrictive. A better model for this scenario is the QRQW PRAM, defined next.

2.2 The QRQW PRAM model

Definition 2.3 *The* **QRQW PRAM** *model consists of a number of processors, each with its own private memory, communicating by reading and writing locations in a shared memory. Processors execute a sequence of synchronous steps, each consisting of the following three substeps:*

1. *Read substep: Each processor i reads r_i shared memory locations, where the locations are known at the beginning of the substep.*
2. *Compute substep: Each processor i performs c_i* RAM *operations involving only its private state and private memory.*[3]
3. *Write substep: Each processor i writes to w_i shared memory locations (where the locations and values written are known at the beginning of the substep).*

[3]As in the existing PRAM models, each processor is assumed to be a sequential random access machine. See, e.g. [Rei93]. For the QRQW PRAM, a processor may perform multiple RAM operations in a compute substep, e.g. summing c_i numbers stored in its private memory, and is charged accordingly.

Concurrent reads and writes to the same location are permitted in a step. In the case of multiple writers to a location x, an arbitrary write to x succeeds in writing the value present in x at the end of the step.

Definition 2.4 *Consider a* QRQW PRAM *step with maximum contention κ, and let $m = \max_i\{r_i, c_i, w_i\}$ for the step, i.e. the maximum over all processors i of its number of reads, computes, and writes. Then the* **time cost** *for the step is* $\max\{m, \kappa\}$. *The* **time** *of a* QRQW PRAM *algorithm is the sum of the time costs for its steps. The* **work** *of a* QRQW PRAM *algorithm is its processor-time product.*

This cost measure models, for example, a MIMD machine such as the Tera Computer [ACC+90], in which each processor can have multiple reads/writes in progress at a time, and reads/writes to a location *queue* up and are serviced one at a time. Neither the EREW PRAM nor the CRCW PRAM model allows a processor to have multiple reads/writes in progress at a time, as this generalization is unnecessary when reads/writes complete in unit time. This feature, which distinguishes the QRQW PRAM from the SIMD-QRQW PRAM as well as the EREW PRAM and CRCW PRAM, enables the processors to do useful work while awaiting the completion of reads/writes that encounter contention. Nevertheless, as we show below, the CRCW PRAM can simulate the QRQW PRAM to within constant factors.

The restriction that the processors in a read substep know, at the beginning of the substep, the locations to be read reflects the intended emulation of the QRQW PRAM model on a MIMD machine in which the reads are issued in a pipelined manner, to amortize against the delay (latency) on such machines in reading the shared memory. Likewise writes in a write substep are to be pipelined in the intended emulation. On the other hand, each of the local operations performed in a compute substep can depend on compute operations in the same substep; since these operations are assumed to take constant time in the intended emulation, there is no need for pipelining (to within constant factors). The emulation inserts a barrier synchronization among all the processors between every read and write substeps, so that the processors notify each other when it is safe to proceed with the next substep. This synchronization is accounted for in the emulation. A formal description of the intended emulation and its performance appears in Section 5.

On existing parallel machines, there are a number of factors that determine the time to process shared memory read and write requests, including contention in the interconnection network and at the memory modules. Often, reads and writes to *distinct* shared memory locations may delay one another. Moreover, issued memory requests cannot be withdrawn. To reflect these realities of existing machines, the QRQW PRAM (as well as the SIMD-QRQW PRAM) does not permit processors to make inferences on the contention encountered based on the delays incurred. In addition, issued memory requests may not be withdrawn, and an algorithm has not completed until all issued memory requests have been processed. In this way, the QRQW models, although explicitly accounting only for the delays resulting from multiple requests to the *same* locations, can be efficiently emulated on models that account for these additional concerns, as shown in Section 5.

As with the SIMD-QRQW PRAM, the work is not the number of operations since operations encountering non-constant contention may be charged non-constant time. (In fact, the only situation where the work is a good reflection of the number of operations is when pipelining is extensively employed, i.e. when the average over i of $(r_i + c_i + w_i)$ is $\Omega(\kappa)$.)

Also, as with the SIMD-QRQW PRAM, there is no explicit metric for the number of steps in an algorithm. As we show in Section 5, there is no need for such a metric in the context of the intended emulation. On the other hand, the synchronization at the end of each bulk-synchronous step is a source of overhead on existing machines, and hence one may wish to include this additional metric when analyzing algorithms on the QRQW models.

2.3 Relations between models

The primary advantage of the QRQW PRAM model over the SIMD-QRQW PRAM model is that the QRQW permits processors each to perform a series of reads and writes in a step while incurring only a single penalty for the contention of these reads and writes. In the SIMD-QRQW, a penalty is charged after each read or write in the series; often the resulting aggregate charge for contention is far greater than the single charge under the QRQW model. On the other hand, by adding more processors to the SIMD-QRQW, we can match the time bounds (but not the work bounds) obtained for the QRQW:

Observation 2.1 *A p-processor* QRQW PRAM *algorithm running in time t can be emulated on a pt-processor* SIMD-QRQW PRAM *in time $O(t)$.*

Proof. For each QRQW processor $i \in [1..p]$, we assign a team, T_i, of t SIMD-QRQW processors, with each team having a leader, l_i. Each leader l_i maintains the entire local state of QRQW processor i during the emulation. For each team T_i, we have an auxiliary array, A_i, of size t for communications between l_i and each member of its team. Consider the jth step of a QRQW PRAM algorithm, with time cost t_j and maximum contention $k_j \leq t_j$. For each QRQW processor i, let r_i, c_i, and w_i be the number of reads, RAM operations, and writes performed by processor i this step. Processor i is emulated as follows: (1) The leader l_i writes the r_i locations to be read to A_i, one location per cell. (2) Each member of T_i reads its cell in A_i, reads the designated location (if any) in the shared memory, and then writes the value read to its cell in A_i. (3) The leader l_i reads the values in A_i, performs the c_i RAM operations, and then writes the w_i locations and values to be written to A_i, one per cell. Finally, (4) each member of T_i reads its cell in A_i, and then writes the designated value to the designated location (if any) in the shared memory. Step 1 takes $O(r_i)$ time, step 2 takes $O(k_j)$ time, step 3 takes $O(r_i + c_i + w_i)$ time, and step 4 takes $O(k_j)$ time. Thus the overall time to emulate the jth QRQW step is $O(t_j)$, and the observation follows. ∎

Note that in fact only $p \cdot \tau$ processors are needed in the above emulation, where $\tau \leq t$ is the maximum time for any one step of the QRQW PRAM algorithm.

The SIMD-QRQW PRAM model permits each processor to have at most one shared memory request outstanding at a time, as in the standard PRAM model. This places an upper bound on the number of requests that must be handled by the interconnection network of the parallel machine. For most MIMD machines, permitting only one request per processor is artificially restrictive, and the QRQW PRAM model has no such restriction. On the other hand, since there is no bound in the QRQW PRAM on the number of outstanding requests, there is a danger that QRQW PRAM algorithms will flood the network with requests beyond its capacity to efficiently process them. One approach towards alleviating this potentially serious problem is to divide steps with many shared memory requests into a sequence of steps with fewer requests per step. In general one could indicate, for each QRQW PRAM algorithm, the maximum number of requests in any one step of the algorithm. Then when implementing the algorithm on a given parallel machine, this number could be compared with the maximum effective network capacity of the machine to determine if the memory requests can be efficiently processed by the network.

Let M_1 and M_2 be two models. We define $M_1 \preceq M_2$ to denote that any one step of M_1 with time cost $t \geq 1$ can be emulated in $O(t)$ time on M_2 using the same number of processors. For concurrent and queue writes we assume throughout this paper that an arbitrary processor succeeds in the write; however, the relations stated below hold as long as both machines use the same write-conflict rule.

Observation 2.2 EREW PRAM $\preceq$ SIMD-QRQW PRAM $\preceq$ QRQW PRAM $\preceq$ CRCW PRAM

QRQW Separation Results				
weaker model	**stronger model**	**time separation**	**problem**	**reference**
{det.,rand.} EREW	{det.,rand.} SIMD-QRQW	$\Omega(\sqrt{\lg n})$	2-compaction	§7
det. QRQW	det. CRCW	$\Omega(\lg n/\lg\lg n)$	OR function	§6.1
{det.,rand.} QRQW	{det.,rand.} CRCW	$\Omega(\lg n)$	broadcasting	§8

Table 1: *Results separating the* QRQW *from the* EREW *and* CRCW *models, including both deterministic time (*det.*) and randomized expected or w.h.p. time (*rand.*).*

Proof. By straightforward emulation. For the CRCW emulating a QRQW step of time cost t: (1) for $j = 1, \ldots, \max_i\{r_i\}$, perform the jth read operation (if any) at each processor in one step using CR, then (2) for $j = 1, \ldots, \max_i\{c_i\}$, perform the jth compute operation (if any) at each processor, then (3) for $j = 1, \ldots, \max_i\{w_i\}$, perform the jth write operation (if any) at each processor in one step using CW. This takes time $\max_i\{r_i\} + \max_i\{c_i\} + \max_i\{w_i\} \leq 3t$. ■

Results separating the QRQW from the EREW and CRCW models appear in Table 1.

2.4 A family of queue models

The definitions of SIMD-QRQW PRAM and QRQW PRAM can be generalized so that the charge for maximum contention κ is $f(\kappa)$, a non-decreasing function of κ. When $f(\kappa) = 1$ for all κ, both models are equivalent to the CRCW PRAM. Likewise, when $f(1) = 1$ and $f(\kappa) = \infty$ for $\kappa \geq 2$, both models are equivalent to the EREW PRAM. Note that the distinction between the SIMD-QRQW PRAM and the QRQW PRAM arises only when $f(\kappa) > 1$ and is finite for some κ.

Another possible cost function is $f(\kappa) = \lg \kappa$; such a function may occur in a *hypothetical* variant of combining networks, but it is not known to be relevant to any existing machines (there are no known techniques for achieving this cost function for an arbitrary set of readers/writers). The log cost function may prove to be relevant to future machines that employ an optical crossbar to interconnect the processors [GMR94c, MR96]. However, in this paper, we will focus our attention on the cost function, $f(\kappa) = \kappa$, that reflects the realities of proven technologies. (For some machines that do not handle contention well, super-linear functions such as $f(\kappa) = \kappa \lg \kappa$ may be appropriate; such cost functions are not considered in this paper.) Other possible variants of the model permit write-conflict rules other than *arbitrary*; however, we note that the *arbitrary* rule reflects the realities of most current commercial and research machines.

As the queue rule can be applied independently to reads or writes, we can also consider models such as the SIMD-CRQW or CRQW PRAM. For each such hybrid model, the PRAM version can trivially simulate the SIMD version with no loss. Figure 1 depicts the relative power of the various models immediately apparent from the definitions, extending the results in Observation 2.2 to the hybrid models. Likewise, Table 2 presents additional separation results for the hybrid models.

3 Why QRQW?

The PRAM model was introduced in 1978 [FW78], with the CREW contention rule. Since that time, a variety of contention rules have been proposed and studied, with the most-widely studied being the EREW, CREW and CRCW rules. Variants of the CRCW PRAM such as ARBITRARY, COLLISION, COMMON, PRIORITY, ROBUST, and TOLERANT have been proposed and studied (see e.g., [Mat92] for definitions); these differ in their write-conflict rules. Given the plethora of contention rules already

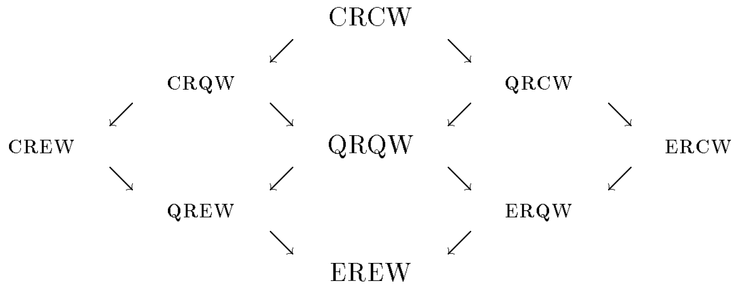


Figure 1: *The relative power of various* PRAM *concurrency rules. The same relationships hold for the* SIMD *versions of the queue models. For concurrent write, we assume an arbitrary processor succeeds in writing. In this figure an arrow denotes that the* PRAM *model, M_1, at the tail of the arrow can simulate the* PRAM *model, M_2, at its head with at most a small constant loss in performance (and possibly some improvement), i.e. $M_2 \preceq M_1$. Our results characterize more precisely the relative power of some of the concurrency rules.*

in the literature, it is reasonable to ask if there is a need for yet another contention rule, and in particular, whether the QRQW PRAM is an important new PRAM model.

The QRQW PRAM is a fundamental departure from standard PRAM models because it is *the first* PRAM *model to properly account for contention*, as reflected in most current commercial and research machines. By permitting contention, it reflects the realities of current machines, and enables simpler and more efficient algorithms for many basic problems. By charging for contention, it reflects the realities of machines with non-combining networks, i.e. most current commercial and research machines. In the remainder of this section, we elaborate on these points, and then compare the QRQW models to related work. We begin with a critique of the exclusive and concurrent rules.

3.1 EREW is too strict

The *exclusive* contention rule is almost universally considered by PRAM proponents to be a realistic rule for parallel machines. In the EREW PRAM, it is forbidden to have two or more processors *attempt* to read or write the same location in the same step. We know of no existing shared memory parallel machine with this restriction on its global communication. Moreover, the exclusive rule leads to

Separation Results for Hybrid Models				
weaker model	**stronger model**	**time separation**	**problem**	**reference**
{det.,rand.} EREW	{det.,rand.} ERQW	$\Omega(\sqrt{\lg n})$	2-compaction	§7
det. {QR,CR}EW	det. {QR,CR}QW	$\Omega(\lg\lg n)$	2-compaction with n procs	§7
rand. CREW	rand. CRQW	$\Omega(\lg\lg n)$	OR function	§6.3
det. {ER,QR,CR}QW	det. {ER,QR,CR}CW	$\Omega(\lg n/\lg\lg n)$	OR function	§6.1
det. QR{EW,QW} rand. QR{EW,QW}	det. CR{EW,QW} rand. CR{EW,QW}	$\Omega(\lg n)$ $\Omega(\lg n)$	broadcasting broadcasting	§8

Table 2: *Separation results for the hybrid queue models, including both deterministic time (*det.*) and randomized expected time or w.h.p. time (*rand.*). All results listed above hold for the* SIMD *versions as well.*

unnecessarily slow algorithms. A simple example is the *2-compaction* problem, in which there are two nonempty cells at unknown positions in an array of size n, and the contents of these cells must be moved to the first two locations of the array. An EREW PRAM requires $\Omega(\sqrt{\lg n})$ time to solve the 2-compaction problem; an n-processor CREW PRAM requires $\Omega(\lg\lg n)$ time [FKL+92]. However, as shown in Section 7, there is a trivial constant time n-processor QRQW PRAM algorithm for this problem.

The exclusive contention rule eliminates many *randomized* algorithmic techniques. Randomization used to determine *where* a processor should read or write (e.g. random sampling, random hashing) cannot avoid some small likelihood of concurrent reading or writing, and hence cannot be incorporated directly into EREW algorithms.[4] Likewise, most *asynchronous* algorithms cannot avoid scenarios in which concurrent reading or writing occur. Hence existing ASYNCHRONOUS PRAM models (e.g. [CZ89, Gib89, Nis90, MPS92]) do not enforce the exclusive rule, assuming instead a CRCW cost measure.[5]

3.2 CRCW may be too powerful

At the other extreme, the *concurrent* contention rule may be too powerful. In the CRCW PRAM, each step takes unit time, independent of the amount of contention in the step. Thus no distinction is made between low contention and high contention algorithms. On parallel machines with non-combining networks, high contention read steps or write steps can be quite slow, as each of the requests for a highly contended location is serviced one-by-one, creating a serial bottleneck or "hot spot" [PN85]. Moreover, intermediate nodes on the path to the contended destination become congested as well, so a single hot spot can even delay requests destined for other nodes in the network. If all p processors request the same location, a common occurrence in CRCW PRAM algorithms, a direct implementation of the algorithm can incur a p-fold loss in speedup due to contention, sometimes becoming no better than a sequential algorithm.

An active area of research is how to execute a CRCW step that includes high contention reads or writes without creating hot spots. Software approaches, e.g. using sorting [Val90a], may incur an overhead considered unacceptable in practice, even on machines that support them. This is arguably true of the MasPar MP-1, for example, where the concurrent-write primitive provided for the MP-1 is around 20 times slower than writing according to a random permutation [Pre92]. As indicated in Section 1, the asymptotically best work-preserving emulation known for simulating the CRCW PRAM on machines with non-combining networks suffers polynomial slowdown [Val90a, Val92]. Thus, the running time on the parallel machine will be a polynomial factor slower than the running time indicated by the CRCW model.

Hardware approaches for executing high contention CRCW steps without hot spots incorporate combining logic into the interconnection network. Ranade's work [Ran89] shows that any CRCW step can be simulated on certain hypercube-based networks in the same asymptotic time as an EREW step, and development of machines based on his technique have been reported (e.g. [AKP91, DS92]). It is an open question whether the system cost of supporting CRCW efficiently in hardware is justified, particularly on MIMD machines, and work continues in this area (e.g. [DK92]). Existing commercial machines are primarily designed to process low contention steps efficiently; high contention steps are slow operations.

Note that the weaknesses of the exclusive and concurrent contention rules apply independently to reading and writing. Thus hybrids such as the CREW PRAM or the ERCW PRAM are too strict for

[4] These techniques can be incorporated into CRCW algorithms, and emulated on the EREW, but at logarithmic cost in time and work.

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

writing (reading, respectively) and may be too powerful for reading (writing, respectively).

3.3 Most existing machines are QRQW

Table 3 classifies some existing multiprocessors according to the concurrent read and write capabilities of their interprocessor communication. As seen from the table, the contention rule for most of these machines, including the CRAY T3D, IBM SP2, Intel Paragon, MIT J-Machine, nCUBE 2S, Stanford DASH, and Tera Computer, is well-approximated by the QRQW rule. For the synchronous MasPar MP-1 and MP-2, the contention rule is well-approximated by the SIMD-QRQW rule.

For the Kendall Square KSR1, the contention rule is well-approximated by the CRQW rule. The Thinking Machines CM-5 provides a second network that can be used to perform fast SCAN operations [Ble89]. An appropriate model for this machine would be a QRQW model with unit-time SCAN operations.

Note that each of the asynchronous machines (marked A in Table 3) allow for general asynchronous algorithms. Thus their contention rule in its full generality is well-approximated by the asynchronous queue contention rule provided by the QRQW ASYNCHRONOUS PRAM [GMR96b] (except for the KSR1, which is well-approximated by an asynchronous CRQW contention rule). On the other hand, their contention rule with respect to bulk-synchronous algorithms is well-approximated by the (bulk-synchronous) queue contention rule provided by the simpler QRQW or CRQW PRAM.

A number of these machines, such as Stanford DASH, provide caches local to each processor; on reading a shared memory location, a copy is stored in the processor's cache for future reuse. Multiple processors with cached copies of a location may then request to read the location, and will be serviced in parallel from their local caches. To maintain a single consistent value for a location, these machines typically invalidate all cached copies of the location before permitting a processor to write to the location. This fast concurrent re-reading of memory locations is not modeled in the QRQW models, due to the following. If the contents of a shared memory location is stored in a private memory location when first read by a processor, then there is no need to issue a subsequent shared memory read for this location unless some other processor *may have* changed the value: the private copy may be used instead. Moreover, if some other processor *did* change the value, then fast re-reading is not possible and there will be a penalty for high contention with or without the caches. Thus fast re-reading of memory locations seems to have only a secondary effect on the contention encountered in parallel algorithms, and hence has been omitted from the model, for simplicity.

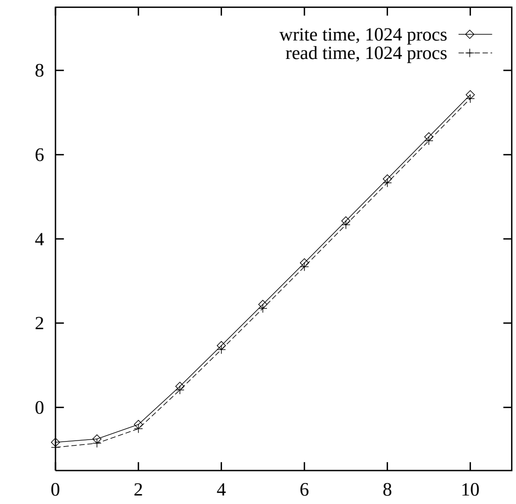
We have conducted experiments to measure the effect of contention on a 16,384 processor MasPar MP-1. The results of these experiments are given in Figure 2. The experiments show that the SIMD-QRQW rule is a far more accurate reflection of running time on the MasPar MP-1 than a CRCW contention rule. Indeed, the overall time for the read (write) step is dominated by the cost of contention at a fairly small value for the contention, and then the time grows nearly linearly with the contention. In contrast, the CRCW contention rule would predict that the overall time would not change with the contention. The differences between the left and right plots in the figure demonstrate that charging k for contention k, as in the SIMD-QRQW rule, becomes an accurate reflection of the running time only when each processor has its own global router port; otherwise, a more complicated metric would be more accurate. Note that the MasPar MP-2, the successor of the MP-1, provides additional router ports, to help alleviate the bottleneck in the MP-1 caused by having one port for every 16 processors. Thus we would expect the MP-2 to behave more like the plot on the left, i.e. more according to the SIMD-QRQW rule.

Contention Rules of Some Existing Multiprocessors			
Cray T3D [KS93]	$	A	QRQW
IBM SP2 [IBM94]	$	A	QRQW
Intel Paragon [Bel92]	$	A	QRQW
Kendall Square KSR1 [FBR93]	$	A	CRQW
MasPar MP-1 [Mas91], MP-2	$		
global router		S	QRQW
xnet		S	limited CREW
nCUBE 2S [SV94]	$	A	QRQW
Thinking Machines CM-5 [Lei92b]	$		
data network		A	QRQW
control network		S	fast SCAN ops
Bus-based machines	$	A	limited CRQW
Fluent [Ran89, AKP91]	P	S	CRCW
MIT J-Machine [DKN93]	P	A	QRQW
Stanford DASH [LLG$^+$92]	P	A	QRQW
Tera Computer [ACC$^+$90]	P	A	QRQW

Table 3: *Contention rules of some existing multiprocessors. We have included message-passing machines, as well as shared memory ones, since they are often used to run (slightly modified versions of) shared memory algorithms or programs. The second column indicates commercial product ($) or working prototype (P). The third column indicates synchronous (S) or asynchronous (A) machines. In the last column, ER or EW denotes that programs for the machine are forbidden from having multiple requests for a location. QR or QW denotes that multiple requests to a location may be issued, and requests are generally serviced one-at-a-time. CR or CW denotes that multiple requests to a location may be issued, and requests are combined in the network.*

A few entries do not quite fit the taxonomy and require further explanation. In the XNET *of the MP-1 and MP-2, processors are limited to reading or writing values stored at nodes a given distance away in a given compass direction; each processor may broadcast a value to all intermediate nodes on the path. The control network of the CM-5 provides fast* SCAN *primitives [Ble89]; such primitives provide concurrent reading and writing and more (only) for well-structured sets of requests that fit the segmented-*SCAN *paradigm [Ble93]. In bus-based machines, the bus typically services only one shared memory location at a time; all processors requesting to read the location can be serviced at the same time without penalty. Finally, a number of these machines provide caches that permit fast concurrent re-reading of shared memory locations: once a set of processors have read a location, they may subsequently re-read the location without incurring a penalty for contention, as long as no processor has written to the location in the meantime.*

MasPar running times (in milliseconds)				
contention	1024 processors		16384 processors	
in step	write	read	write	read
1	0.563	0.518	7.321	6.849
2	0.595	0.554	7.435	6.957
4	0.755	0.703	7.415	6.944
8	1.414	1.332	7.449	6.976
16	2.765	2.589	7.870	7.369
32	5.445	5.090	10.283	9.636
64	10.784	10.116	15.354	14.391
128	21.503	20.167	25.952	24.329
256	42.922	40.271	47.127	44.205
512	85.761	80.459	89.746	84.194
1024	171.441	160.846	175.485	164.635
2048	—	—	346.781	325.357
4096	—	—	689.218	646.656
8192	—	—	1374.849	1289.970
16384	—	—	2744.192	2574.748



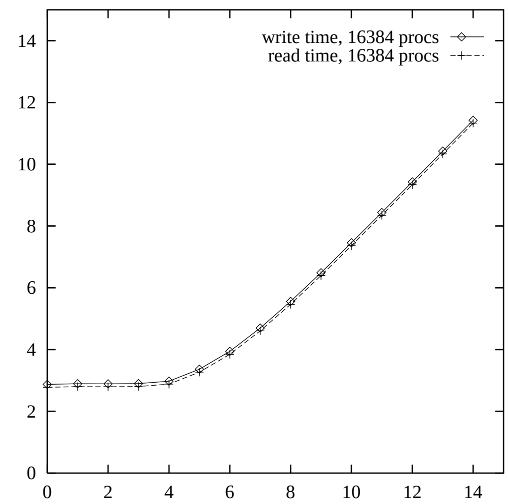


Figure 2: *Performance measurements on the MasPar MP-1 for a read or write step, under increasing contention to a location: Top, timing measurements. Bottom, plot of the measurements on a log-log scale, showing the running time (y-coordinate) as a function of the contention in the step (x-coordinate). Results for* 2^{10} *and* 2^{14} *processors are shown. In the base experiment (contention 1, x-coordinate 0), each processor reads (writes) according to a random permutation. In the general experiment (contention* 2^i*, x-coordinate* i*), the first* 2^i *processors read (write) the same location* M*, while the remaining processors read (write) according to the original random permutation. Shown are the cumulative times of repeating the experiment on 20 different random permutations. In the plots, the y-coordinate depicts the base 2 logarithm of the number of milliseconds needed.*

The experiments show that high contention steps are several orders of magnitude slower than random permutations, and moreover, that doubling the contention nearly doubles the running time, at least for medium to high contention steps. The dependence of the running time on the contention is more dramatic in the experiments with 1024 processors than with 16,384 processors, for the following reason. In the 16,384 processor MasPar MP-1, each global router port is shared by 16 processors, creating an additional serial bottleneck. The experiments with 1024 processors use only one processor per port, thereby avoiding this serial bottleneck.

3.4 Related work

In an early related work, Greenberg [Gre82] considered broadcast communication schemes, such as the Ethernet, that have queues for submitted messages. More recently, Cypher [Cyp88] analyzed the performance of a maximum-finding algorithm under assumptions similar to the SIMD-QRQW PRAM. Dietzfelbinger, Kutylowski and Reischuk [DKR94] defined the *few-write* PRAM, that permits one-step concurrent writing of up to κ writes, where κ is a parameter of the model, as well as unlimited concurrent reading. Valiant [Val90a] introduced the BSP model (see Section 5), and studied a specialization of the model with logarithmic periodicity and constant throughput, which we call here the *standard* BSP model. In [Val90a] it is shown that a v-processor PRAM step with contention κ can be simulated on a p-processor standard BSP in $O(v/p + \kappa \lg p)$ time w.h.p. A large number of papers have studied the *Distributed Memory Machine*, in which the shared memory is partitioned into modules such that at most one memory location within each module can be accessed at a time. Concurrent reads and writes may or may not be allowed depending on the model. (See [Lei92a, Val90b] and the references therein.) An early example is the *CTA* (or *Candidate Type Architecture*) machine model proposed by Snyder [Sny86] which consists of a set of processors connected by a sparse communication network of unspecified topology and linked to a controller. The CTA is parametrized by the number of processors and the latency of interprocessor communication. Aumann and Rabin [AR92] showed that a PRAM algorithm can be simulated on a very general asynchronous parallel system that permits $O(\lg n)$ contention to a location in unit time.

There have been several recent papers presenting independent work in related areas. Culler *et al.* [CKP+93] proposed the *LogP* model, a lower-level message-passing model in which there is limited communication bandwidth: a processor can send or receive at most one message every g cycles, where g is a parameter of the model. There is also a limit on the number of messages in the network at the same time. The LogP model permits general asynchronous algorithms. Liu, Aiello and Bhatt [LAB93] studied a message-passing model in which messages destined for the same processor are serviced one-at-a-time in an arbitrary order. Their 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 models capture *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. Unlike each of these models, the QRQW PRAM does not explicitly limit the number of outstanding requests. The SIMD-QRQW PRAM, on the other hand, has the same restriction as the Liu *et al.* and Dwork *et al.* models, namely, one request per processor.

In contrast to many of the models mentioned above, the QRQW model focuses on the contention to locations, rather than to memory modules or processors. Any algorithm with high location contention will perform poorly on machines with non-combining networks, regardless of the number of memory modules; any lower bound on location contention is a lower bound on memory module contention. By focusing on locations, the QRQW model is independent of the particular layout of memory on the machine, e.g. the number of memory modules. Moreover, it is more relevant to cache-only memory architectures (COMA), such as the KSR1, that dynamically map memory locations to processors as the computation proceeds. Location contention is also a relevant metric for cache coherence overhead, since the number of invalidates or updates that must be sent on a write is often proportional to the number of processors concurrently accessing the location being written [LLG+90]. The QRQW models, like the standard PRAM and other similar models, are true shared memory models, providing a simple view of the shared memory as a collection of independent cells.

4 Adding contention to the work-time framework

In the *work-time* presentation, a parallel algorithm is described in terms of a sequence of steps, where each step may include any number of concurrent read, compute, or write operations [JáJ92]. In this context, the *work* is the total number of operations, and the *time* is the number of steps. This is sometimes the most natural way to express a parallel algorithm, and forms the basis of many data parallel languages (e.g. NESL [BCH+93]). For standard PRAM models, Brent's scheduling principle [Bre74] can often be applied to obtain an efficient $O(work/p + time)$ time algorithm for a p-processor PRAM.

4.1 The QRQW work-time framework

We show here that the work-time paradigm can be used to advantage for the QRQW PRAM. It is extended into a QRQW work-time presentation by adding at each parallel step i the additional parameter k_i, the maximum contention at this step. Given an algorithm $\mathcal{A}$ in the QRQW work-time presentation, define the *work* to be the total number of operations[6] and the *time* to be the sum over all steps of the maximum contention k_i of each step (as in the SIMD-QRQW PRAM model). We note that one of the useful features of the traditional work-time presentation is that the time evaluation is independent of the work evaluation. Perhaps somewhat surprisingly, in the QRQW work-time presentation, too, the time evaluation (which is based on the contention at each step) is independent of the work evaluation: there is no benefit or loss in having steps with high contention also have high work, as long as the total contention and work remain the same. An algorithm given in the QRQW work-time presentation can be transformed into an efficient QRQW PRAM algorithm, as follows:

Theorem 4.1 *Assume processor allocation is free. Any algorithm in the* QRQW *work-time presentation with x operations and time t (where t is the sum of the maximum contention at each step) runs in at most $x/p+t$ time on a p-processor* QRQW PRAM.

Proof. Let the number of parallel steps in the algorithm be r. Let x_i be the number of operations in the ith parallel step, and let $k_i \geq 1$ be the maximum contention in the ith parallel step, $1 \leq i \leq r$. Hence $t = \sum_{i=1}^{r} k_i$. We map the operations in the ith step uniformly onto the p QRQW PRAM processors. Thus each QRQW PRAM processor will receive at most $n_i = \lceil x_i/p \rceil$ operations. The maximum contention at any memory location remains the same as in the original work-time algorithm, i.e. at most k_i. Hence the time cost for the ith step on a p processor QRQW PRAM is $\max\{n_i, k_i\}$. The overall algorithm, therefore, takes time

$$\sum_{i=1}^{r} \max\{\lceil x_i/p \rceil, k_i\} \leq \sum_{i=1}^{r} ((x_i/p) + k_i) = x/p + t.$$

∎

Thus Brent's scheduling principle can indeed be extended to the QRQW work-time framework.

4.2 Automatic processor allocation

The mechanism of translating an algorithm from a work-time presentation into a PRAM description is not addressed by Theorem 4.1, which assumes processor allocation is free. If the PRAM model is extended to include a unit time SCAN operation [Ble89], as may be appropriate for some machines

[6] This contrasts with the *work* in a QRQW PRAM or SIMD-QRQW algorithm, which is the processor-time product.

such as the CM-5, then the processor allocation issue can be resolved with only small overhead. The rest of this section deals with the standard PRAM models that do not incorporate the SCAN operation.

Traditionally, the processor allocation needed to implement Brent's scheduling principle has been devised in an ad-hoc manner. However, it is known that in several common situations an efficient automatic implementation is feasible, especially on the CRCW, often using linear compaction and load balancing algorithms as essential tools (see [Mat92] and references therein). In this section, we adapt these techniques to the QRQW PRAM model.

Rather than tracing the details of each technique, it would be helpful to show that in general the contention parameter on the QRQW does not change the validity of these CRCW techniques. Indeed, the fact that time evaluation and work evaluation are done independently in the QRQW work-time presentation suggests that scheduling techniques on the CRCW PRAM should be useful for the QRQW PRAM as well. Next we elaborate on this issue.

Let $\mathcal{A}$ be a class of algorithms given in the QRQW work-time presentation. A QRQW *scheduling scheme* $\mathcal{S}_{\mathcal{A}}$ for $\mathcal{A}$ is a scheme that maps any algorithm A in $\mathcal{A}$ into a QRQW PRAM algorithm. If algorithm A has work-time bounds of w and t, then $\mathcal{S}_{\mathcal{A}}$ will convert A into a p-processor QRQW PRAM algorithm for some suitable number of processors p that runs in time $\tau = t + t_{\mathcal{A}} + (w + w_{\mathcal{A}})/p$ and work $\tau \cdot p$, where $t_{\mathcal{A}}$ and $w_{\mathcal{A}}$ are the overhead in time and work for the scheduling scheme $\mathcal{S}_{\mathcal{A}}$. The scheduling scheme $\mathcal{S}_{\mathcal{A}}$ is *work preserving* if $\tau \cdot p = O(w)$.

Similar definitions hold for a scheduling scheme for a class of CRCW PRAM algorithms given in the work-time presentation.

Consider a class of algorithms $\mathcal{B}$ given in a CRCW work-time presentation, and let $\mathcal{S}_{\mathcal{B}}$ be a scheduling scheme that adapts each algorithm B in $\mathcal{B}$ into a CRCW PRAM algorithm B'. Let $\mathcal{A}$ be the class of algorithms in the QRQW work-time presentation corresponding to $\mathcal{B}$. That is, each algorithm A in $\mathcal{A}$ is identical to an algorithm B in $\mathcal{B}$ except that the time of each parallel step is taken to be the maximum contention of that step. Thus algorithms A and B perform the same amount of work, though the running time of algorithm A could be larger. Let $\mathcal{S}_{\mathcal{A}}$ be a scheduling scheme on a QRQW PRAM corresponding to the CRCW PRAM scheduling scheme $\mathcal{S}_{\mathcal{B}}$. That is, the scheduling scheme $\mathcal{S}_{\mathcal{A}}$ adapts each algorithm A in $\mathcal{A}$ into a QRQW PRAM algorithm A' which, except for the scheduling overhead, is identical in execution (but not necessarily in time complexity) to the CRCW PRAM algorithm B' derived by $\mathcal{S}_{\mathcal{B}}$ from the algorithm B in $\mathcal{B}$ to which algorithm A corresponds.

Lemma 4.2 *Let $w_{\mathcal{A}}, t_{\mathcal{A}}$ and $w_{\mathcal{B}}, t_{\mathcal{B}}$ be the work-time overhead of $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{B}}$ respectively. If $\mathcal{S}_{\mathcal{B}}$ is work-preserving on the* CRCW PRAM *and $w_{\mathcal{A}} = O(w_{\mathcal{B}})$ then $\mathcal{S}_{\mathcal{A}}$ is work-preserving on the* QRQW PRAM*. In particular, an algorithm A in $\mathcal{A}$ with work-time bounds of w and t will run optimally on a* QRQW PRAM *in time $O(w/q)$ using q processors when $q \leq w/(t + t_{\mathcal{A}})$.*

Proof. Let A correspond to a CRCW work-time algorithm B in $\mathcal{B}$ that runs in time t' with work w'. Note that $t \geq t'$ and $w = w'$ since A corresponds to B. On a p-processor CRCW PRAM, $\mathcal{S}_{\mathcal{B}}$ maps algorithm B to run in time $t' + t_{\mathcal{B}} + (w + w_{\mathcal{B}})/p$. Thus $p \cdot (t' + t_{\mathcal{B}} + (w + w_{\mathcal{B}})/p) = O(w)$ for some value of p, since $\mathcal{S}_{\mathcal{B}}$ is work-preserving. This implies that $w_{\mathcal{B}} = O(w)$, and hence $w_{\mathcal{A}} = O(w)$.

Now let $\mathcal{S}_{\mathcal{A}}$ map algorithm A into a QRQW PRAM algorithm A' with $q \leq w/(t + t_{\mathcal{A}})$ processors. Then algorithm A' will run in time $\tau = t + t_{\mathcal{A}} + (w + w_{\mathcal{A}})/q$ on the q-processor QRQW PRAM which gives the desired work preserving schedule since

$$q \cdot \tau = q \cdot (t + t_{\mathcal{A}}) \; + \; w \; + \; w_{\mathcal{A}} \; \leq \; w + w + O(w) = O(w).$$

■

Note that we can always transform a CRCW PRAM scheduling scheme into an *equivalent* QRQW PRAM scheduling scheme simply by viewing the overhead of the CRCW scheduling scheme in the work-time framework and interpreting it as a (possibly slower) QRQW scheduling scheme with the same work overhead. This leads to the following corollary to the above lemma.

Corollary 4.3 *Let $\mathcal{B}$ be a class of algorithms given in a* CRCW *work-time presentation and let $\mathcal{A}$ be a class of algorithms in the* QRQW *work-time presentation corresponding to $\mathcal{B}$. Let $\mathcal{S}_{\mathcal{B}}$ be a* CRCW *scheduling scheme for $\mathcal{B}$ and let $\mathcal{S}_{\mathcal{A}}$ be the equivalent* QRQW *scheduling scheme for $\mathcal{A}$. If $\mathcal{S}_{\mathcal{B}}$ is work-preserving on the* CRCW PRAM *then $\mathcal{S}_{\mathcal{A}}$ is work-preserving on the* QRQW PRAM*.*

The above corollary shows that it is always possible to derive a work-preserving QRQW scheduling scheme for a class of QRQW work-time algorithms corresponding to a class of CRCW work-time algorithms that have a work-preserving schedule. However, such a QRQW scheduling scheme can be very slow. In particular if the algorithm for the CRCW scheduling scheme has a read or write with concurrency $\Theta(w_{\mathcal{B}})$, where $w_{\mathcal{B}}$ is the work overhead of the CRCW scheduling scheme, then the work-preserving QRQW scheduling scheme degenerates into a sequential algorithm. A more useful way to apply Lemma 4.2 is to substitute a fast work-preserving QRQW PRAM algorithm for the QRQW scheduling scheme in place of the CRCW scheduling scheme.

In what follows, we give three examples of general classes of algorithms for which automatic processor allocation techniques can be applied to advantage: geometric-decaying algorithms, general task-decaying algorithms, and spawning algorithms. Processor allocation is done by a scheduling scheme that uses an algorithm for *linear (approximate) compaction*. The *linear compaction* problem generalizes the 2-compaction problem, as follows: Given k nonempty cells at unknown positions in an array of size n, with k known, move the contents of the nonempty cells to an output array of $O(k)$ cells. The linear compaction problem can be solved by a randomized CRCW PRAM algorithm in time $T'_{lc}(n) = O(\lg^* n)$ time and linear work w.h.p. [GMV91]. In Section 7 (Theorem 7.6) we show that the linear compaction problem can be solved by a randomized SIMD-QRQW PRAM algorithm in time $T_{lc}(n) = O(\sqrt{\lg n})$ and linear work w.h.p.

Sometimes the linear compaction algorithm is used under the assumption that the number of non-empty cells is at most k. An unsuccessful termination of the algorithm is used to determine that the input consists of more than k non-empty cells. To make such determination possible, it is necessary to employ an algorithm for computing the OR function, as well as an algorithm for the broadcasting problem. Furthermore, recall that a subtle property of the QRQW models is that unsuccessful steps may turn out to be overly expensive if they incur (unexpected) high contention. (This is a rather significant technical issue in the algorithms of Section 6.) We assume here that the number of non-empty cells never exceeds αk for some constant $\alpha > 0$, where k is the estimated upper bound. In such cases, the running time of the linear compaction algorithm of Theorem 7.6 will increase by at most a constant factor. Let $T_{lcd}(n)$ be the running time of a linear compaction algorithm followed by a determination of whether the algorithm was successful or not on an n-processor QRQW PRAM, and let $T''_{lcd}(n)$ be the corresponding running time on a CRQW PRAM.

In Section 8 we show that on the QRQW PRAM broadcasting requires $\Omega(\lg n)$ expected time. Therefore when it is necessary to determine if a run of linear compaction was unsuccessful on the QRQW PRAM, it is best to use a $\Theta(\lg n)$ time EREW PRAM algorithm for prefix sums [LF80]. Hence, $T_{lcd}(n) = \Theta(\lg n)$. Performing a broadcast on the SIMD-CRQW PRAM is trivial in constant time. In Section 6 (Theorem 6.8) we show that the OR problem can be solved by a SIMD-CRQW PRAM in time $O(\lg n / \lg\lg n)$ and linear work w.h.p. Hence, $T''_{lcd}(n) = O(\lg n / \lg\lg n)$ w.h.p.

Task-decaying algorithms. A *task-decaying* algorithm (or simply a *decaying* algorithm) is one that starts with a collection of unit tasks. Each of these tasks progresses for a certain number of steps of the algorithm, and then *dies*. A task is said to be a *live* task until it dies. No other tasks are created during the course of the algorithm. The *work load* w_i is the number of live tasks at step i of the algorithm.

Geometric-decaying algorithms: A decaying algorithm in either the QRQW or the CRCW work-time presentation is *geometric-decaying* if the sequence of work loads $\{w_i\}$ is upper bounded by a decreasing geometric series. Typically the work w of such algorithms is $O(n)$ where n is the problem size. Let $\mathcal{A}$ and $\mathcal{B}$ be the class of geometric-decaying algorithms in the QRQW and CRCW work-time presentations respectively. Using techniques from [GM91, GM96, MV91] and Lemma 4.2 we have:

Theorem 4.4 *Let A be a geometric-decaying algorithm in a* QRQW *work-time presentation with time t and work n. Then Algorithm A can be implemented on a p-processor* QRQW PRAM *to run in time $O(n/p)$ when $p = O(n/(t + T_{lc}(n)\lg(T_{lc}(n))))$.*

Proof. Let B be a geometric decaying algorithm in the CRCW work-time presentation to which Algorithm A corresponds. A work-preserving scheduling scheme $\mathcal{S}_{\mathcal{B}}$ that can adapt Algorithm B into a p-processor CRCW PRAM algorithm B' is given in [MV91]. The scheduling scheme $\mathcal{S}_{\mathcal{B}}$ consists of $\lg(n/p)$ applications of an algorithm for linear compaction problem of size p. On the QRQW PRAM we will use a scheduling scheme $\mathcal{S}_{\mathcal{A}}$ corresponding to $\mathcal{S}_{\mathcal{B}}$. When mapping into a p-processor QRQW PRAM, scheduling scheme $\mathcal{S}_{\mathcal{A}}$ will consist of $\lg(n/p)$ applications of a QRQW PRAM algorithm for linear compaction problem of size p. The time overhead incurred by scheduling scheme $\mathcal{S}_{\mathcal{A}}$ is $t_{\mathcal{A}} = O(T_{lc}(p)\lg(n/p))$, and the work overhead is $p \cdot t_{\mathcal{A}}$. We observe, as in [MV91], that if $T_{lc}(p)\lg(n/p) \geq n/p$, then $\lg(n/p) = O(\lg(T_{lc}(n)))$, and hence for $p \leq n/(T_{lc}(p)\lg(T_{lc}(n)))$, scheduling scheme $\mathcal{S}_{\mathcal{A}}$ has a work overhead of $O(n)$. Therefore, by Lemma 4.2, $\mathcal{S}_{\mathcal{A}}$ maps algorithm A into a p-processor QRQW PRAM algorithm A' to run in time $O(n/p)$ provided $p = O(n/(t + T_{lc}(p)\lg(T_{lc}(n)))$. ∎

By Theorem 7.6 we obtain

Corollary 4.5 *Algorithm A in Theorem 4.4 can be implemented on a p-processor* QRQW PRAM *to run in time $O(n/p)$ w.h.p. when $p = O(n/(t + \sqrt{\lg n}\lg\lg n))$.*

General task-decaying algorithms: Recall that in a task-decaying algorithm in either the QRQW or the CRCW work-time presentation, the sequence of work loads $\{w_i\}$ is a monotonically non-increasing series. Thus, task-decaying algorithms generalize geometric-decaying algorithms. A task-decaying algorithm is *predicted* if an approximate bound on the sequence of work loads $\{w_i\}$ is known in advance; specifically, if a sequence $\{w_i'\}$ is given such that for all i, $w_i' \geq w_i$ and $\sum_i w_i' = O(\sum_i w_i)$. Let $\mathcal{A}$ and $\mathcal{B}$ be the class of general task-decaying algorithms in the QRQW and CRCW work-time presentations respectively.

Theorem 4.6 *Let A be a task-decaying algorithm in a* QRQW *work-time presentation with time t and work n. Then Algorithm A can be implemented to run in time $O(n/p)$ on a p-processor* QRQW PRAM *when $p = O(n/(t + T_{lcd}(n)\lg(T_{lcd}(n))))$ and on a p-processor* CRQW PRAM *when $p = O(n/(t + T''_{lcd}(n)\lg(T''_{lcd}(n))))$. If Algorithm A is also predicted then it can be implemented on a p-processor* QRQW PRAM *to run in time $O(n/p)$ when $p = O(n/(t + T_{lc}(n)\lg(T_{lc}(n))))$.*

Proof. Let B be a predicted task-decaying algorithm in a CRCW work-time presentation to which Algorithm A corresponds. A work-preserving scheduling scheme $\mathcal{S}_{\mathcal{B}}$ that can adapt Algorithm B into a p-processor CRCW PRAM algorithm B' is given in [MV91]. The scheduling scheme $\mathcal{S}_{\mathcal{B}}$ is based on several applications of an algorithm for the linear compaction problem of size p. The analysis in [MV91] is based on showing that the cost of all but $\lg(n/p)$ applications of the linear compaction algorithm can be amortized against the execution of Algorithm B, with only a constant factor overhead. Hence the time overhead of $\mathcal{S}_{\mathcal{B}}$ is $t_{\mathcal{B}} = O(T'_{lc}(n)\lg(n/p))$. As for the geometric decaying-algorithm, the time overhead can be shown to be $t_{\mathcal{B}} = O(T'_{lc}(n)\lg(T'_{lc}(n)))$.

Consider a scheduling scheme $\mathcal{S}_{\mathcal{A}}$, corresponding to $\mathcal{S}_{\mathcal{B}}$, which adapts Algorithm A to a p-processor QRQW PRAM algorithm A'. An amortization argument similar to the one used for $\mathcal{S}_{\mathcal{B}}$

implies that the cost of all but $\lg(n/p)$ applications of the linear compaction algorithm can be amortized against the execution of Algorithm A, with only a constant factor overhead. The time overhead of $\mathcal{S}_{\mathcal{A}}$ is therefore $t_{\mathcal{A}} = O(T_{lc}(n)\lg(n/p))$, and hence $t_{\mathcal{A}} = O(T_{lc}(n)\lg(T_{lc}(n)))$, and the work overhead is $p \cdot t_{\mathcal{A}}$. Hence for $p = O(n/(T_{lc}(n)\lg(T_{lc}(n)))$ this schedule has a work overhead of $O(n)$. By Lemma 4.2 the scheduling scheme $\mathcal{S}_{\mathcal{A}}$ maps A into a p-processor QRQW PRAM in $O(n/p)$ time provided $p = O(n/(t + T_{lc}(n)\lg(T_{lc}(n))))$.

If Algorithm B is not predicted then each application of the linear compaction algorithm must be followed by a detection of whether or not there was a successful termination. In such case, the underestimation is by at most a factor of 2. Similar arguments to the above imply that the corresponding algorithm A can be adapted to a QRQW PRAM algorithm with running time $O(n/p)$ provided $p \leq n/(t + T_{lcd}(n)\lg(T_{lcd}(n)))$ and to a CRQW PRAM algorithm with running time $O(n/p)$ provided $p \leq n/(t + T''_{lcd}(n)\lg(T''_{lcd}(n)))$ ∎

By the result stated above we have:

Corollary 4.7 *Algorithm A in Theorem 4.6 can be implemented to run in time $O(n/p)$ w.h.p. on a p-processor* QRQW PRAM *when $p = O(n/(t + \lg n \lg\lg n))$ and on a p-processor* CRQW PRAM *when $p = O(n/(t + \lg n))$. If Algorithm A is predicted then it can be implemented on a p-processor* QRQW PRAM *t run in time $O(n/p)$ w.h.p. when $p = O(n/(t + \sqrt{\lg n}\lg\lg n))$.*

Spawning algorithms. A *spawning* algorithm starts with a collection of unit tasks, and at each step of the algorithm, each task can

i. progress to the next step of the algorithm;

ii. progress to the next step of the algorithm *and* spawn another new task; or

iii. not progress to the next step and die.

The total number of tasks in a spawning algorithm may increase or decrease in each step. Thus, the spawning model generalizes the model for task-decaying algorithms. As in the task-decaying model, a spawning algorithm is *predicted* if an approximate bound on the sequence of work loads $\{w_i\}$ is known in advance; specifically, if a sequence $\{w'_i\}$ is given such that for all i, $w'_i \geq w_i$ and $\sum_i w'_i = O(\sum_i w_i)$.

Theorem 4.8 *Let A be a spawning algorithm in a* QRQW *work-time presentation running in time t and work n, and let t' be the number of parallel steps in A. Then Algorithm A can be implemented to run in time $O(n/p)$ on a p-processor* QRQW PRAM *when $p = O(n/(t + t' \cdot T_{lcd}(n)))$ and on a p-processor* CRQW PRAM *when $p = O(n/(t + t' \cdot T''_{lcd}(n)))$. If Algorithm A is also predicted then it can be implemented to run in time $O(n/p)$ on a p-processor* QRQW PRAM *when $p = O(n/(t + t' \cdot T_{lc}(n)))$.*

Proof. Let B be a predicted spawning algorithm in a CRCW work-time presentation to which Algorithm A corresponds. Then, the running time of Algorithm B is t'. A work-preserving scheduling scheme $\mathcal{S}_{\mathcal{B}}$ that can adapt Algorithm B into a p-processor CRCW PRAM algorithm B' is given in [Mat92]. The scheduling scheme $\mathcal{S}_{\mathcal{B}}$ consists of applying an algorithm for a linear compaction problem of size p a constant number of times after each parallel step. The time overhead of $\mathcal{S}_{\mathcal{B}}$ is therefore $O(t' \cdot T'_{lc}(n))$.

Consider a scheduling scheme $\mathcal{S}_{\mathcal{A}}$, corresponding to $\mathcal{S}_{\mathcal{B}}$, which adapts Algorithm A to a p-processor QRQW PRAM algorithm A'. The scheduling scheme $\mathcal{S}_{\mathcal{A}}$ consists of applying an algorithm for a linear compaction problem of size p a constant number of times after each parallel step. The time overhead incurred by $\mathcal{S}_{\mathcal{A}}$ is thus $t_{\mathcal{A}} = O(t' \cdot T_{lc}(n))$ and the work overhead is $w_{\mathcal{A}} = p \cdot t_{\mathcal{A}}$. Hence by Lemma 4.2 algorithm A' runs in time $O(n/p)$ on a p-processor QRQW PRAM provided $p = O(n/(t + t' \cdot T_{lc}(n)))$.

If Algorithm B is not predicted then, as in the case of the task-decaying algorithm of Theorem 4.6, each application of the linear compaction algorithm must be followed by a detection of whether or not there was a successful termination. Similar arguments to the above imply that the corresponding algorithm A can be adapted to a p-processor QRQW PRAM algorithm running in time $O(n/p)$ provided $p = O(n/(t + t' \cdot T_{lcd}(n)))$, and to a p-processor CRQW PRAM algorithm running in time $O(n/p)$ provided $p = O(n/(t + t' \cdot T''_{lcd}(n)))$. ∎

Corollary 4.9 *Algorithm A in Theorem 4.8 can be implemented to run in time $O(n/p)$ w.h.p. on a p-processor* QRQW PRAM *when $p = O(n/(t + t' \cdot \lg n))$ and on a p-processor* CRQW PRAM *when $p = O(n/(t+t' \cdot \lg n/\lg\lg n))$. If Algorithm A is predicted then it can be implemented to run in time $O(n/p)$ w.h.p. on a p-processor* QRQW PRAM *when $p = O(n/(t + t' \cdot \sqrt{\lg n}))$.*

The spawning model can be further generalized to include a *start* operation in which one task may spawn n new tasks to begin in the next time step. This extended model is called V-PRAM in [Goo91] where it was suggested. It was shown in [Goo91] that the work-preserving scheme for the spawning model can be extended to the V-PRAM model as well, with the same overhead. Accordingly, Theorem 4.8 and Corollary 4.9 apply to the V-PRAM model.

A more general type of spawning algorithm, the *L-spawning algorithm*, is studied in [GMR96a]. In the L-spawning model, each task can spawn up to $L - 1$ additional tasks at each step. It is shown in [GMR96a] that an L-spawning algorithm with time t, work n, and t' parallel steps can be implemented on a p-processor QRQW PRAM to run in time $O(n/p)$ w.h.p. when $p = O(n/(t + t'\sqrt{\lg n}\lg\lg L + t'\lg L))$. This implementation applies a more general load balancing algorithm given in [GMR96a].

5 Realization on feasible networks

The Bulk-Synchronous Parallel (BSP) model was introduced by Valiant [Val90a, Val90b] as a model of parallel computation that takes into account overheads incurred by latency, synchronization and memory granularity. It consists of components that can perform local RAM computations and communicate with one another through a router which delivers messages between pairs of components. Messages to a component are serviced one-at-a-time. The BSP provides facilities for synchronizing the components at regular intervals. There are three parameters to the model: p, the number of components, *periodicity* L, the number of time units between synchronizations, and *throughput* g, a measure of the bandwidth limitations of the router. A particular case studied by Valiant is one that sets g to be a constant and L to be $\Theta(\lg p)$, and has each synchronization involve all the components; we denote this the *standard* BSP model.

A standard BSP computation consists of a sequence of supersteps, with each superstep separated from the next by a global synchronization point among all the components. In each superstep, each component sends messages, receives messages, and performs local RAM steps. Operations at a component (message initiations, message receipts, RAM operations) are assumed to take constant time. No assumption is made about the relative delivery times of messages within a superstep, and local operations may only use data values locally available to the component prior to the start of the superstep. If the operations in a superstep, including message deliveries, do not complete in L time units, additional intervals of L time units are allocated to the superstep until it completes.

The BSP model has been advocated as one that forms a bridge between software and hardware in parallel machines; that is, between abstract models for algorithm design and realistic parallel machines. This approach is supported in [Val90a, Val90b] by providing a fast, work-preserving emulation of the standard BSP model on hypercube-type non-combining networks on the one hand, and a fast, work-preserving emulation of the EREW PRAM on the standard BSP on the other hand. In particular, it is shown that the EREW PRAM can be emulated in a work-preserving manner with logarithmic slowdown on the standard BSP, while the standard BSP can be emulated in a work-preserving manner with constant slowdown on, e.g., the multiport hypercube. In the *multiport* hypercube on p nodes, each node can receive a message on each of its $\lg p$ incoming wires and route them along the appropriate outgoing wires in constant time, subject to the constraint that at most one message can be sent along each outgoing wire. These emulations show that the choice of $L = \Theta(\lg p)$ and $g = \Theta(1)$ used in the standard BSP is sufficient to hide the latency, synchronization, and memory granularity overheads occurring in the emulations.

Valiant [Val90a] shows that a v-processor PRAM step with contention κ can be simulated on a p-processor standard BSP in $O(v/p + \kappa \lg p)$ time w.h.p. It follows readily from this result that a p-processor SIMD-QRQW PRAM algorithm running in time t can be emulated on a $(p/\lg p)$-component standard BSP model in $O(t \lg p)$ time w.h.p.

In this section we show that the more powerful QRQW PRAM can also be emulated in a work-preserving manner with only logarithmic slowdown on the standard BSP as well as on hypercube-type networks. The proof of this result is complicated by the fact that a QRQW step with time cost k may have up to $2kp$ reads and writes, whereas in the previous emulation results, the PRAM step being emulated had at most $2p$ reads and writes, independent of k. As in the previous emulations of PRAM models on the standard BSP given in [Val90a], we apply a random hash function to map the PRAM shared memory onto the BSP components; this function is assumed to map each shared memory location to a component chosen uniformly and independently at random.

Theorem 5.1 *A p-processor* QRQW PRAM *algorithm (or* SIMD-QRQW PRAM *algorithm) running in time t can be emulated on a $(p/\lg p)$-component standard* BSP *model in $O(t \lg p)$ time w.h.p.*

Proof. Consider the ith step of the QRQW PRAM algorithm, with time cost k_i. For simplicity of exposition, we assume that each processor has exactly k_i shared memory accesses, where an access is either a read or a write. Let $m_1, \ldots, m_d$ be the different memory locations accessed in this step, and let q_j be the number of accesses to location m_j, $1 \le j \le d$. For the purpose of this analysis we add $\delta p k_i$ memory accesses to this step, for a constant $\delta \ge 23$, consisting of accesses with contention k_i to locations $m_{d+1}, \ldots, m_{d'}$, where $d' = d + \delta p$. With this addition, the ith step has $v_i{}' = (\delta+1)pk_i$ concurrent accesses to d' different memory locations, and the maximum contention is k_i. We set $q_j = k_i$ for $d+1 \le j \le d'$ and note that $v' = \sum_{j=1}^{d'} q_j$. We now show that the bound stated in the theorem holds for this augmented problem. Clearly, this implies that the bound holds for the original problem.

We assume that the memory has been randomly hashed onto the $p/\lg p$ components of the BSP. Consider a fixed component C. As in [Val90a], we define a random variable $x_j, 1 \le j \le d'$, where $x_j = q_j/k_i$ if m_j is hashed onto C and zero otherwise. Let $X = \sum_{j=1}^{d'} x_j$. We note that $x_j = q_j/k_i$ with probability $\lg p/p$, and $k_i \cdot X$ is the number of messages sent to C in the ith step. Then

$$E(x_j) = q_j \lg p/(pk_i), \quad 1 \le j \le d' \ .$$

Let μ be the mean of the expectations of the x_j:

$$\mu = \sum_{j=1}^{d'} (q_j \lg p)/(pk_i d') = v_i{}' \lg p/(pk_i d') = (\delta+1)pk_i \lg p/(pk_i d') \ .$$

So $\mu = (\delta + 1) \lg p/d'$. By Hoeffding's inequality [Hoe63],

$$Pr(X > (\mu + z)d') \le e^{-z^2 d'/3\mu} \ ,$$

provided $z < \min(\mu, 1-\mu)$. Let $z = \mu/2$. Then

$$Pr(X > 3\mu d'/2) \leq e^{-\mu d'/12} = e^{-(\delta+1)\lg p/12} = 1/p^{\Theta(\delta)} \ .$$

By choosing δ sufficiently large, we have that each component receives at most $3\mu d' k_i/2 = \Theta(k_i \lg p)$ messages in the ith QRQW step w.h.p.

Each BSP component emulates $\lg p$ QRQW PRAM processors. It sends $O(k_i \lg p)$ "read" messages and receives $O(k_i \lg p)$ (w.h.p.) such messages. In the next superstep, it sends $O(k_i \lg p)$ (w.h.p.) "read reply" messages and receives $O(k_i \lg p)$ such replies. Finally, in the next superstep, it performs $O(k_i \lg p)$ local RAM operations, sends $O(k_i \lg p)$ "write" messages, and receives $O(k_i \lg p)$ (w.h.p.) such messages, updating the values of the appropriate locations. Since the periodicity L is $\Theta(\lg p)$ and the gap g is constant, the time taken to complete the ith step on the BSP is $O(k_i \lg p)$ w.h.p.

Thus, w.h.p. the BSP completes the emulation of the QRQW computation in $O(\sum_{i=1}^{m} k_i \lg p)$ time, i.e. in $O(t \lg p)$ time. ∎

Note that unlike Valiant's emulation of the EREW PRAM on standard BSP, the emulation above may result in a rather uneven distribution of messages among the components whenever there is an uneven distribution of contention among the locations. This raises concerns regarding possible contention in routing the messages between the components. However, the (standard) BSP model ignores all issues of routing other than the number of messages sent and received at each component, and hence the proof of Theorem 5.1 addresses only these same routing issues.

Further issues in routing do arise in emulating the PRAM or BSP on models such as the multiport hypercube. Valiant defines the *slackness* of a parallel algorithm being emulated to be the ratio of the number of virtual processors in the algorithm to the number of "physical" processors in the emulating model. In [Val90a], Valiant showed that a p-component standard BSP algorithm with slackness at least $\lg p$ and running in time t can be emulated on a p-node multiport hypercube in $O(t)$ time w.h.p. Since the slackness in the emulation in Theorem 5.1 is $\lg p$, we have the following:

Theorem 5.2 *A p-processor* QRQW *algorithm (or* SIMD-QRQW PRAM *algorithm) running in time t can be emulated on a $(p/\lg p)$-node multiport hypercube in $O(t \lg p)$ time w.h.p.*

Thus the uneven distribution of messages that may result from emulating a QRQW PRAM algorithm on the standard BSP does not prevent a fast, work-preserving emulation of the QRQW PRAM on the multiport hypercube.

6 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.

In this section we present several randomized and deterministic algorithms for solving these problems on queue-write PRAMs. Our main result is a randomized algorithm for the two problems on the CRQW PRAM that performs linear work and runs in $O(\lg n/\lg\lg n)$ time with high probability. This result is somewhat surprising since it improves on the best possible time bound (which is $\Theta(\lg n)$) for any deterministic or randomized CREW PRAM algorithm for the two problems.

Most of the randomized algorithms we present are of the Las Vegas type, while a few are of the Monte Carlo type. A *Las Vegas* algorithm is a randomized algorithm that always outputs a correct answer, and obtains the stated bounds with some stated probability. A *Monte Carlo* algorithm, in contrast, is a randomized algorithm that outputs a correct answer with some stated probability. In the analysis of some of our randomized algorithms, we apply the Chernoff bound stated in Section 5:

$$Pr\{X \geq \beta E[X]\} \leq e^{(1-1/\beta-\ln\beta)\beta E[X]}, \quad \text{for all } \beta > 1.$$

In particular, we use the following corollary to this Chernoff bound:

Observation 6.1 *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. Furthermore, if $E[X] \leq 1$ then $X = O(\lg n/\lg\lg n)$ w.h.p.*

Proof. Let $\beta = c\lg n/fE[X]$, for a constant $c > \max\{2, f/\lg n\}$ to be determined. Then $\beta > 1/E[X] \geq 2^f$. By the Chernoff bound,

$$\begin{aligned} Pr\{X \geq c\lg n/f\} &\leq e^{(1-1/\beta-\ln\beta)\cdot(c/f)\lg n} < e^{-(c/2f)\ln\beta\lg n} \\ &= e^{-(c/2f)\lg\beta\cdot\ln n} = 1/n^{(c/2f)\lg\beta} < 1/n^{c/2}\ . \end{aligned}$$

Hence for any $\delta > 1$, there exists a constant $c = \max\{2\delta, f/\lg n\}$ such that $Pr\{X \geq c\lg n/f\} < 1/n^{\delta}$.

If $E[X] \leq 1$, we take $\beta = c\lg n/\lg\lg nE[X]$, for a constant $c > 2$ to be determined. Then $\lg\beta \geq \lg\lg n - \lg\lg\lg n \geq 2\lg\lg n/3$. By the Chernoff bound,

$$\begin{aligned} Pr\{X \geq c\lg n/\lg\lg n\} &\leq e^{(1-1/\beta-\ln\beta)\cdot(c/\lg\lg n)\lg n} < e^{-(c/2\lg\lg n)\ln\beta\lg n} \\ &= e^{-(c/2\lg\lg n)\lg\beta\cdot\ln n} = 1/n^{(c/2\lg\lg n)\lg\beta} \leq 1/n^{c/3}\ . \end{aligned}$$

Hence for any $\delta > 1$, there exists a constant $c = 3\delta$ such that $Pr\{X \geq c\lg n/\lg\lg n\} < 1/n^{\delta}$. ∎

6.1 Deterministic algorithms

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 SIMD-ERQW 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 the natural EREW PRAM algorithm for leader election which uses a parallel prefix algorithm to compute the location of the first 1 in the input; this takes $\Theta(\lg n)$ time and $\Theta(n)$ work.

We can derive an $\Omega(\lg n/\lg\lg n)$ lower bound for the OR function 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. We begin by proving a more general result for emulating the CRQW on the few-write PRAM, and then provide the OR lower bound.

Observation 6.2 *A p-processor* CRQW 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 algorithm runs in time at most t on all inputs, then the maximum write contention is at most t on all inputs. Hence the t-write PRAM can be used to emulate each write substep, and the emulation proceeds as was done for the CRCW (Observation 2.2). ∎

Theorem 6.3 *Any deterministic algorithm for computing the* OR *function on a* CRQW 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 PRAM. Then by Observation 6.2, 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 $T = \Omega(\lg n/\lg\lg n)$. ∎

Since the ERCW PRAM can compute the OR function in constant time, Theorem 6.3 implies the following separation result:

Corollary 6.4 *There is an $\Omega(\lg n/\lg\lg n)$ time separation between a deterministic* {ER,QR,CR}QW PRAM *with arbitrarily many processors and a deterministic* {ER,QR,CR}CW PRAM.

Cook, Dwork and Reischuk [CDR86] proved that any deterministic algorithm for computing the OR function on a CREW PRAM with arbitrarily many processors requires $\Omega(\lg n)$ time. Dietzfelbinger, Kutylowski and Reischuk [DKR94] later proved a similar lower bound for randomized CREW PRAM algorithms. The difficulty in extending either of these results to the CRQW PRAM is that in the CRQW PRAM, the running time of a step may be different on different inputs. Thus in a CRQW write step with contention k for a given input I, the lower bound argument of [CDR86, DKR94] will allow processors to gain knowledge about input I as a function of the maximum contention, K, for the step over *all* inputs, and K could be much larger than k.

6.2 Randomized algorithms for special cases

In this subsection, we present a series of randomized leader election algorithms, under various scenarios. First, consider the leader election problem when the value of k is known. On the SIMD-QRQW PRAM, a simple, fast, randomized algorithm for this problem is to have the k processors whose input bits are 1 write to the output cell with probability $1/k$. This runs in constant time on the SIMD-QRQW, and, as a low-contention algorithm, will run fast in practice. The failure probability can be reduced by repeating the algorithm.

Observation 6.5 *Consider the problem of electing a leader bit from among the k out of n bits that are 1, where k is known. There is a (randomized) Monte Carlo* SIMD-ERQW PRAM *algorithm that runs in $O(1)$ expected time and $O(n)$ expected work, and probability of failure less than $1/e$. There is a (randomized) Las Vegas* SIMD-CRQW PRAM *algorithm that runs in $O(1)$ expected time and $O(n)$ expected work.*

Proof. The index of each bit whose value is 1 is written into the output cell with probability $1/k$. This has constant expected contention, and the probability that no value is written is $(1 - 1/k)^k < 1/e$. To obtain a Las Vegas algorithm, the write step is repeated until there is at least one writer. Termination is detected by using the concurrent-read capability. The expected time is $O(1 + 1/e + 1/e^2 + 1/e^3 + \ldots)$ which is $O(1)$. ∎

The expected time for this algorithm is constant; however, we are interested in high probability results. The next two theorems deal with high probability randomized algorithms for the case when a good estimate for k is known, and the case when a good upper bound for the value of k is known.

Given a good estimate for k. In the following, we describe a fast leader election algorithm when the number of bits competing for leadership is known to within a multiplicative factor of $2^{\sqrt{\lg n}}$.

Theorem 6.6 *Consider the problem of electing a leader bit from among the k out of n bits that are 1. Let $\hat{k}$ be known to be within a factor of $2^{\sqrt{\lg n}}$ of k, i.e. $\hat{k}/2^{\sqrt{\lg n}} \le k \le \hat{k}2^{\sqrt{\lg n}}$. There is a*

Monte Carlo SIMD-ERQW PRAM *algorithm that, w.h.p., elects a leader in* $O(\sqrt{\lg n})$ *time with* $O(n)$ *work. On the* SIMD-CRQW PRAM*, or if* $\hat{k} \leq 2^{\sqrt{\lg n}}$*, the same bounds can be obtained for a Las Vegas algorithm.*

Proof. We describe the algorithm for $n/\sqrt{\lg n}$ processors. Let $p = \min(1, 2^{c\sqrt{\lg n}}/\hat{k})$, 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. The input bits are partitioned among the processors such that each processor is assigned $\sqrt{\lg n}$ bits.

Step 1. Each processor selects a leader from among its input bits that are 1, if any.

Step 2. Each processor with a leader writes, with probability p, 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.

If $\hat{k} \leq 2^{\sqrt{\lg n}}$ then $p = 1$ and this is a Las Vegas algorithm. Else 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.

Step 1 takes $O(\sqrt{\lg n})$ time. Since $m = 2^{O(\sqrt{\lg n})}$, an EREW binary fanin approach can be used to obtain the same time bounds for step 3. For step 2, we will show that the contention is $O(\sqrt{\lg n})$ w.h.p. Let X_i be the number of writers to cell i of A. Then

$$E[X_i] \leq kp/m \leq k2^{c\sqrt{\lg n}}/\hat{k}m \leq k/\hat{k}2^{2\sqrt{\lg n}} \leq 1/2^{\sqrt{\lg n}} \ .$$

It follows from Observation 6.1 that the maximum contention over all cells of A is $O(\sqrt{\lg n})$ w.h.p.

It remains to show that w.h.p., there is at least one writer to A (assuming that $k > 0$). If $\hat{k} \leq 2^{c\sqrt{\lg n}}$, then $p = 1$ and hence there will be one writer to A for each processor that has an input bit that is 1. Else $\hat{k} > 2^{c\sqrt{\lg n}}$, and the probability that there are no writers to A is at most

$$\begin{aligned}(1-p)^{k/\sqrt{\lg n}} &= ((1-1/(1/p))^{1/p})^{pk/\sqrt{\lg n}} < (1/e)^{pk/\sqrt{\lg n}}, \\ &= (1/e)^{(k/\hat{k})2^{c\sqrt{\lg n}}/\sqrt{\lg n}} \leq (1/e)^{2^{(c-1)\sqrt{\lg n}}/\sqrt{\lg n}}.\end{aligned}$$

It follows that c can be chosen so that there is at least one writer w.h.p. ∎

Given an upper bound on k. We next consider the case where we only have an upper bound, k_{max}, on the number of input bits that are 1; the results we obtain are not quite as good as when k is known to within a factor of $2^{\sqrt{\lg n}}$, but better than the case when no bound on k (other than n) is known. The algorithm is a straightforward modification of the previous algorithm (Theorem 6.6).

Theorem 6.7 *Consider the problem of electing a leader bit from among k out of n bits that are 1, given an upper bound, k_{max}, on k. There is a Las Vegas* SIMD-ERQW PRAM *algorithm that runs in* $O(\lg k_{\max} + \sqrt{\lg n})$ *time with* $O(n)$ *work w.h.p.*

Proof. We describe the algorithm 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 $0 < \epsilon \leq 1$, apply the EREW parallel prefix algorithm, as mentioned in Section 6.1, 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. The first and third steps take $O(\lg k_{max} + \sqrt{\lg n})$ time. In the second step, the expected contention to a cell i in A is at most $1/2^{\sqrt{\lg n}}$. It follows from Observation 6.1 that the maximum contention over all cells of A is $O(\sqrt{\lg n})$ w.h.p. ∎

6.3 A general randomized algorithm

It is shown in [DKR94] that the OR function on n bits requires $\Omega(\lg n)$ time on a randomized CREW PRAM. (This lower bound is for randomized algorithms that have zero probability of a concurrent write, and correctly compute the OR with probability bounded away from 1/2.) In contrast to this lower bound, we show in this subsection that a randomized SIMD-CRQW PRAM can compute the OR function on n bits in $O(\lg n/\lg\lg n)$ time and linear work w.h.p.

Theorem 6.8 *There is a Las Vegas* SIMD-CRQW PRAM *algorithm for the leader election problem (and the* OR *function) that runs in $O(\lg n/\lg\lg n)$ time and linear work w.h.p.*

Proof. We first show the time bound using $n\lg\lg n$ processors. We describe the algorithm for the OR function, which can be trivially modified to solve the leader election problem. Since the number, k, of contending 1-bits is unknown, we will search for the true value of k. We take larger and larger samples until we either find a sample that contains at least one input bit that is 1, or learn that all input bits are 0. We must ensure that w.h.p., there will be at least one writer (with a 1) prior to the iteration in which there are too many writers (i.e. the iteration where the contention would *not* be $O(\lg n/\lg\lg n)$). The new algorithmic result below is a technique for amplifying probabilities on the SIMD-QRQW model so that this occurs.

1. Let $s = c\lg n/\lg\lg n$, with $c \geq 1$ a constant determined by the analysis. Let A be an array of $s^2\lg\lg n$ memory cells, A' be an array of $s\lg\lg n$ memory cells, and A'' be an array of $\lg\lg n$ memory cells, each initialized to all zeros. The output is to be written in memory cell x. We assign $\lg\lg n$ processors to each input bit. Each processor reads its input bit. Let $p = s^2/n$.
2. Each processor with input bit 1 is active with probability p. Each such active processor writes its index to some cell i of A chosen uniformly at random, and then reads that cell. If the cell contains its index (i.e. no other processor overwrote it), then it writes its index to cell i' of A', $i' = i \bmod s\lg\lg n$, and then reads that cell. If the cell contains its index, then it writes its index to cell i'' of A'', $i'' = i' \bmod \lg\lg n$, and then reads that cell. If the cell contains its index, then it writes a 1 into memory cell x.
3. Each processor reads x. If $x = 0$, repeat steps 2 and 3 with $p = ps$. If $p \geq 1$, repeat one last time with $p = 1$ and then stop.

Note that x is set to 1 only if there is a processor with a 1. Conversely, each processor whose input bit is 1 either writes a 1 into x, writes its index in a cell of A in the iteration that x is set to 1, or stops when $x = 1$; hence the algorithm always outputs the correct answer. There are $O(\lg n/\lg s)$ iterations. If no processor writes to A in an iteration, then the iteration takes $O(1)$ time. Else there is one last iteration in which writes to A, A', A'', and x occur.

We now analyze the contention of these last four write steps. Let p_j be the probability used at iteration j; i.e. $p_j = s^{j+1}/n$. Let k be the number of (original) input bits that are '1'. Since we have a write step, $1 \leq k \leq n$. Let $t \geq 0$ be an integer such that $n/s^t \geq k > n/s^{t+1}$. Consider iteration $t+1$, if it occurs. The probability that no processor writes is at most

$$\begin{aligned}(1-p_{t+1})^{k \lg\lg n} &< (1/e)^{p_{t+1} k \lg\lg n} \\ &= (1/e)^{k s^{t+2} \lg\lg n/n} \\ &< (1/e)^{s \lg\lg n} < (1/e)^{c \lg n} \\ &= (1/e)^{c' \cdot \ln n} = 1/n^{c'},\end{aligned}$$

for some constant c'. Hence, if $k > 0$, there will be no iteration $t+2$ w.h.p.

Let W be the number of active processors at iteration $t+1$, if it occurs. Then

$$E[W] = p_{t+1} k \lg\lg n = s^{t+2} k \lg\lg n/n.$$

By the choice of t, $s \geq s^{t+1}k/n > 1$, and hence $s^2 \lg\lg n \geq E[W] > s \lg\lg n$. Let X_i be the number of writers to cell i of A in iteration $t+1$. Then

$$E[X_i] = E[W]/s^2 \lg\lg n \leq 1.$$

By Observation 6.1, and since there are $s^2 \lg\lg n = o(n)$ cells, the maximum contention for this write is $O(\lg n/\lg\lg n)$ w.h.p.

This bounds as well the contention of any iteration less than $t+1$ in which a write to A occurs (and hence is the last iteration). Since there is at most one winner from each cell of A and exactly s cells of A that map to one cell of A', the maximum contention to a cell of A' is s. Likewise, the maximum contention to a cell of A'' is s and the maximum contention to cell x is $\lg\lg n$.

It follows that the overall running time is $O(\lg n/\lg\lg n)$ w.h.p.

Finally, in order to make the algorithm work-optimal, we should achieve the same time bound using only $n \cdot \lg\lg n/\lg n$ processors. For this we use an initial computation phase in which we reduce the size of the input from n to $n/\lg n$. For this we divide the processors into $n/\lg n$ groups of $\lg\lg n$ processors, and assign to each group the simple task of finding the OR of a block of $\lg n$ input bits in $O(\lg n/\lg\lg n)$ time. We then apply the algorithm described above to the reduced array of $n/\lg n$ bits. This gives us the desired work-optimal randomized algorithm for the OR function on n bits in $O(\lg n/\lg\lg n)$ time w.h.p. ∎

We note that the only large concurrent-read in the previous algorithm is the reading of x in step 3 of the algorithm.

Corollary 6.9 *There is an $\Omega(\lg\lg n)$ time separation between a randomized* CREW PRAM *with arbitrarily many processors and a randomized* SIMD-CRQW PRAM.

7 Linear compaction

Consider an array of size n with k nonempty cells, with k known, but the positions of the k nonempty cells not known. The *k-compaction* problem is to move the contents of the nonempty cells to the first k locations of the array. The *linear compaction* problem is to move the contents of the nonempty cells to an output array of $O(k)$ cells. The best known EREW PRAM algorithms for both problems take $\Theta(\lg n)$ time, using parallel prefix sums [LF80]. Even for the case $k = 2$, there is a randomized $\Omega(\sqrt{\lg n})$ expected time lower bound for the EREW PRAM ([Mac94], following [FKL+92]), and a deterministic lower bound of $\Omega(\lg\lg n)$ for an n-processor CREW PRAM [FKL+92].

The simple deterministic SIMD-ERQW PRAM algorithm for leader election discussed in Section 6.1 can be trivially extended to the k-compaction problem as follows:

Observation 7.1 *There is a deterministic* SIMD-ERQW PRAM *algorithm for the k-compaction problem that runs in $O(k^2)$ time with $O(n)$ work.*

Proof. The input is partitioned into subarrays of k^2 cells. Each of the n/k^2 processors reads the cells in its subarray and creates a linked list of the items in its nonempty cells. Since there are only k nonempty cells, no processor can have more than k items in its linked list. The algorithm proceeds in k rounds, in which processors attempt to place each item on their list. At round i, each processor with an unplaced item writes its index to cell i of the array. A designated processor then reads the cell, and if the index found is j, it signals processor j (by writing to a cell designated for j), which then transfers the contents of its current item to the cell and continues to the next round with its next unplaced item (if any). All other processors continue with the same item as before. The contention in round i is at most $k - i + 1$, so the algorithm runs in $O(k^2)$ time. ∎

By taking $k = 2$, and recalling the lower bounds mentioned earlier for the EREW and CREW PRAM, we obtain the following two results, which are cited in Table 1 and Table 2 in Section 2:

Corollary 7.2 *There is an $\Omega(\sqrt{\lg n})$ time separation between a (deterministic or randomized)* EREW PRAM *with arbitrarily many processors and a (deterministic or randomized)* SIMD-ERQW PRAM.

Corollary 7.3 *There is an $\Omega(\lg\lg n)$ time separation between a deterministic* {QR,CR}EW PRAM *with n processors and a deterministic* SIMD-{QR,CR}QW PRAM *with n processors.*

In the remainder of this section, we develop a SIMD-QRQW PRAM algorithm for the linear compaction problem that runs in $O(\sqrt{\lg n})$ time with linear work w.h.p. Within our algorithm, we will employ the following well-known technique for k-compaction, which runs in $O(\lg n)$ time using only k processors on an EREW PRAM.

Observation 7.4 *The k-compaction problem with one processor assigned to each nonempty cell can be solved by an* EREW PRAM *algorithm in $O(\lg n)$ time.*

Proof. View the n elements as leaves of a full binary tree. At the ith step we work at level i above the leaves, and inductively, for each node v at this level, we have the solution (in the form of a linked list) for the leaves in the subtrees rooted at the two children of v. To combine these solutions at v we only need to make the last distinguished element in the subtree of the left child of v as the successor of the first distinguished element in the right subtree of v. This can be performed by a constant time EREW computation. Finally we perform list ranking on the linked list of distinguished elements (using Wyllie's pointer jumping approach [KR90]) and transfer the elements to their location in the output array.

Note that the input array need not be initialized: since we have an active processor for each distinguished element, we can detect distinguished elements by a *change* in the value of a memory cell. ∎

To prove our SIMD-QRQW PRAM result, we start by proving the following lemma, which shows how to achieve the desired time bound. However, the algorithm performs superlinear work when k is large. We then show how to use this lemma to obtain a linear work algorithm with the same time bound.

Lemma 7.5 *There is a Las Vegas* SIMD-QRQW PRAM *algorithm for linear compaction that runs in $O(\sqrt{\lg n})$ time w.h.p. if $\sqrt{\lg n}$ processors are assigned to each nonempty cell.*

Proof. Let an *item* denote a nonempty input cell. Let $r = \sqrt{\lg n}$, the number of processors assigned to each item. Let A be an auxiliary array of size $m = c_1 r k 2^{c_2\sqrt{\lg n}}$, for constants $c_1 \geq 2$, $c_2 \geq 1$ determined by the analysis. View the array A as partitioned into $k/\lg n$ subarrays of size $m' = c_1 r 2^{c_2\sqrt{\lg n}} \lg n$.

1. For each item, select a subarray of A uniformly at random. Each processor assigned to the item selects a cell in that subarray uniformly at random and tries to claim that cell.
2. At this point, between zero and r cells of A have been claimed on behalf of each item. Denote an item *successful* if at least one cell of A has been claimed on its behalf. For each successful item, select one of its cells in A, and mark the rest as unclaimed.
3. In parallel for all subarrays, compact the claimed cells within each subarray using Observation 7.4. We compact within subarrays here since, for large k, compacting all of A is too slow.
4. View the output array as partitioned into $k/\lg n$ subarrays of size $c_1 \lg n$. For each j, if there are n_j unclaimed cells in subarray j of the output, then the contents of (up to) n_j claimed cells in subarray j of A are transferred to output subarray j. (In the first pass of the algorithm, $n_j = c_1 \lg n$, but in any subsequent pass, n_j may be smaller.) If there are more than n_j claimed cells in a subarray j, then for $i > n_j$, the item associated with the ith claimed cell in subarray j of A is denoted unsuccessful.
5. For each unsuccessful item, each of its r processors returns to step 1.

Since the processors assigned to an item repeat the algorithm until at least one of them has successfully claimed an output cell, this is a Las Vegas algorithm. (Note that processors may complete their participation in the algorithm at different times, not knowing when all processors have terminated.) Let X_j be the number of items selecting subarray j of A in step 1. Then $E[X_j] = k/\lceil k/\lg n\rceil \leq \lg n$. By Chernoff bounds, for $c_1 \geq 2$ defined above,

$$Pr\{X_j \geq c_1 \lg n\} \leq e^{(1-1/c_1 - \ln\ c_1)c_1 \lg n} < e/c_1^{c_1 \lg n} < 1/n^{c_1}.$$

After step 2, there is at most one claimed cell for each item, so w.h.p., there are at most $c_1 \lg n$ claimed cells in a subarray. A processor tries to claim a cell in step 1 by first writing its index to the cell, then reading the cell: if it reads its index, it has claimed the cell, and it writes the contents of its input cell to the claimed cell. For each subarray j, let $Y_{j,i}$ be the number of processors selecting cell i of subarray j of A in step 1. Then $E[Y_{j,i}] \leq r \cdot c_1 \lg n/m' \leq 1/2^{c_2\sqrt{\lg n}}$. It follows from Observation 6.1 that the time for step 1 is $O(\sqrt{\lg n})$ w.h.p.

Step 2 can be done in $O(\lg r)$ time. Step 3 applies Observation 7.4, and runs in $O(\lg m')$ time, which is $O(\sqrt{\lg n})$ time. For step 4, for each j, the current value of n_j, as well as the index of the first unclaimed output cell in subarray j, can be broadcast in $O(\lg\lg n)$ time; the transferring takes constant time.

As for step 5, there are two types of unsuccessful items. As argued above, w.h.p., there are at most $c_1 \lg n$ claimed cells in a subarray. It follows that the probability that an item is unsuccessful in step 1 is less than $(r \cdot c_1 \lg n/m')^r = (1/2^{c_2\sqrt{\lg n}})^{\sqrt{\lg n}} < 1/n^{c_2}$. Moreover, it follows that, w.h.p., no cells are marked unsuccessful in step 4. So w.h.p., all cells are successful in the first pass of the algorithm. ∎

Theorem 7.6 *There is a Las Vegas* SIMD-QRQW PRAM *algorithm for linear compaction that runs in $O(\sqrt{\lg n})$ time with $O(n)$ work w.h.p.*

Proof. We describe the algorithm for $n/\sqrt{\lg n}$ processors. Let an *item* denote a nonempty input cell. Note that we make no assumption on the distribution of the items within the input array.

1. View the n input cells as partitioned into subarrays of size $2\lg^2 n$. Assign $2\lg^{1.5} n$ processors per subarray. In parallel for all subarrays compact the items in each subarray, using parallel prefix.
2. For subarrays with at most $2\lg n$ items, we assign $\sqrt{\lg n}$ processors per item, and apply Lemma 7.5.
3. For subarrays with more than $2\lg n$ items, we view the items as partitioned into blocks of size $\lg n$. There are at most $2\lg n$ such blocks in a subarray, so we assign $\sqrt{\lg n}$ processors per block. Viewing each block as a "super-item", apply Lemma 7.5 to compact the super-items into an array of size $O(k/\lg n)$. Then we transfer the items in each block to the output array of size $O(k)$, in the obvious way.

Each of steps 1–3 takes $O(\sqrt{\lg n})$ time w.h.p. ∎

8 Broadcasting

Given $b \in \{0, 1\}$ in a single memory location, the *broadcasting* problem is to copy b into n fixed memory locations. There is a simple linear work, $O(\lg n)$ time EREW PRAM algorithm for this problem. In this section we show that this algorithm is the best possible even for the (randomized) QRQW PRAM by providing an $\Omega(\lg n)$ lower bound on the expected running time of any deterministic or randomized QRQW PRAM algorithm for this problem.

Our lower bound exploits the fact that the input domain for the broadcasting problem has only two values. We show that for any problem with an input domain of size 2, a SIMD-QRQW PRAM algorithm is no faster than the best EREW PRAM algorithm for the problem, and even a QRQW PRAM algorithm is at most two times faster than the best EREW PRAM algorithm for the problem. We also show that a randomized algorithm for the problem is at most two times faster than the best deterministic algorithm for the problem. These results, in turn, imply our lower bound for broadcasting and related problems due to a lower bound for broadcasting on the EREW PRAM given by [BKK94].

Our simulation of the SIMD-QRQW PRAM and the QRQW PRAM on the EREW PRAM results in a *non-uniform* algorithm on the EREW PRAM. An algorithm is non-uniform if it consists of different programs for different input sizes, and the program for a given input size i cannot be generated easily simply by specifying the value of i. Most algorithms used in practice are *uniform* (i.e., not non-uniform), in which a single program works for all input sizes. A non-uniform algorithm is not desirable from a practical point of view, since the time bound for the algorithm is not guaranteed to be achieved on a given input unless we have already generated the program for that input size. However, the lower bound of [BKK94] holds for both uniform and non-uniform algorithms (as is the case with most lower bounds), and hence our simulation result gives the desired lower bound for the SIMD-QRQW PRAM and the QRQW PRAM.

8.1 Constant size input domain problems

We first deal with the SIMD-QRQW PRAM. We show that any SIMD-QRQW PRAM algorithm for a problem defined on a domain with only two values that runs in time T can be converted into an EREW PRAM algorithm that also runs in time T. The EREW PRAM may be non-uniform and may have a description that is of unbounded size. For an exact definition of the model see [CDR86].

Lemma 8.1 *Let T be the running time for an algorithm A that solves a problem P with input domain of size 2 on a* SIMD-QRQW PRAM*. Then, there exists an algorithm B that solves P in time T on an* EREW PRAM*, using the same number of processors and the same working space. Algorithm B is non-uniform and its description is of size $O(T)$ memory locations per processor.*

Proof. Assume, without loss of generality, that the input domain is $\{0,1\}$. The lemma is proved by constructing the EREW PRAM Algorithm B from Algorithm A. Consider the ith step in Algorithm A, and let $\kappa_i(b)$ be the maximum contention in this step on input b. Let $\kappa'_i = \min\{\kappa_i(0), \kappa_i(1)\}$ (if $\min\{\kappa_i(0), \kappa_i(1)\} = 0$ then $\kappa'_i = 1$). Step i will be implemented in Algorithm B in at most κ'_i substeps, as described below. Therefore, the running time of algorithm B is at most $\sum_i \kappa'_i = \sum_i \min\{\kappa_i(0), \kappa_i(1)\} \leq T$. We describe first the construction for the read step.

Let $\Phi_{i,j,b}$ be the set of processors that read from memory cell j in step i on input $b \in \{0,1\}$. Let $\Phi_{i,j} = \Phi_{i,j,0} \cap \Phi_{i,j,1}$. For processors in each set $\Phi_{i,j,b} \setminus \Phi_{i,j}$, we can prepare a priori copies of the contents of memory cell j, $c(i,b)$, so that they can do the read operation from their appropriate copies without conflict, as described below.

For processors in each set $\Phi_{i,j}$, we serialize their computation by providing an a priori ranking from $[1..|\Phi_{i,j}|]$ to all the processors in $\Phi_{i,j}$, and scheduling the processors according to their ranks. The program for Algorithm B includes for each processor a sequence $\langle i, M(i,b), r(i,b), \phi_i, c(i,b)\rangle$, $i = 1,\ldots,T$, $b \in \{0,1\}$, where $M(i,b)$ is the memory cell from which the processor reads in step i on input b; $r(i,b)$ is the rank of the processor at step i if the processor is in $\Phi_{i,M(i,b)}$, and is null otherwise; $c(i,b)$ is the contents at step i of memory cell $M(i,b)$ if the processor is in $\Phi_{i,M(i,b),b} \setminus \Phi_{i,M(i,b)}$, and is null otherwise; and $\phi_i = \max_j |\Phi_{i,j}|$. (Note that the processor does not need to know the value of b. If, however, $M(i,0) \neq M(i,1)$ or $r(i,0) \neq r(i,1)$ then it implicitly knows the value of b at this stage; this knowledge can be made explicit by replacing the quintuple above by the sextuple $\langle i, M(i,b), r(i,b), \phi_i, c(i,b), b'\rangle$ where $b' \in \{0,1,*\}$.) This sequence can be specified in $O(T)$ memory locations; the program is non-uniform and of unbounded size. In step i, each processor whose $r(i,b)$ is not null will execute its read operation from memory location $M(i,b)$ in substep $r(i,b)$. Each processor whose $r(i,b)$ is null will read $c(i,b)$. After a total of ϕ_i substeps, all processors proceed to step $i+1$.

It remains to show how to handle the write steps. Consider a memory location j in step i, and let $\Phi_{i,j}$, $\Phi_{i,j,0}$, and $\Phi_{i,j,1}$ be defined as for the read step. On input b, it is sufficient to select a priori one processor from $\Phi_{i,j,b}$ that will do the write step to location j. If $\Phi_{i,j}$ is not empty then one of the processors in $\Phi_{i,j}$ will be arbitrarily selected. If $\Phi_{i,j}$ is empty, one of the processors in $\Phi_{i,j,b}$ will be arbitrarily selected, unless it is empty. The write operation will be executed by the selected processor at substep ϕ_i. Thus, all the read operations will be completed before the write operation is executed; moreover, there is no additional time overhead due to the execution of the write operations.

With this scheme, the ith step of Algorithm A is executed in $\phi_i \leq \kappa'_i$ steps by Algorithm B, thus giving the desired result. ∎

We now strengthen the above result for the SIMD-QRQW PRAM to work for the QRQW PRAM with only a constant factor increase in the running time of the simulating EREW PRAM algorithm.

Lemma 8.2 *Let T be the running time for an algorithm A that solves a problem P with input domain of size 2 on a* QRQW PRAM*. Then, there exists an algorithm B that solves P in time $O(T)$ on an* EREW PRAM*, using the same number of processors and the same working space. Algorithm B is non-uniform and its description is of size $O(T)$ memory locations per processor.*

Proof. We show how to handle the read steps of Algorithm A; write steps are treated similarly. Consider the ith read step in Algorithm A on input b. Let the time cost of this step be t_i. Let R_k be the set of reads for processor p_k, and let M_j be the set of read requests for memory location m_j in step i on input b. Note that t_i is the maximum cardinality of the sets R_k, M_j, over all processor and memory indices k, j.

We construct a bipartite graph $B_{i,b} = (P, M, E_{i,b})$, where P contains a vertex for each processor, M contains a vertex for each memory location, and there is an edge $(p_k, m_j) \in E_{i,b}$ if and only if processor p_k reads memory location m_j in step i on input b.

The maximum degree of any vertex in the graph $B_{i,b}$ is t_i. Since $B_{i,b}$ is bipartite it has a proper edge coloring with t_i colors (Theorem 6.1 in [BM76]), i.e. a mapping $c : E_{i,b} \rightarrow \{1, 2, \ldots, t_i\}$ such that for any pair of edges e, f incident on the same vertex, $c(e) \neq c(f)$. Thus for a given input b we can serialize the ith step of Algorithm A into t_i exclusive read substeps by performing the read corresponding to the edges colored l in the lth substep.

Since the input domain is of size 2, b can take on only two values, say 0 and 1, and each processor can be in at most two different states at a given time step, no matter what the input is. In Algorithm B for each step, we run the serialization of the step on input $b = 0$ followed by the serialization of the step on input $b = 1$. If processor p_k is in a state that corresponds only to input $\hat{b} \in \{0, 1\}$ then it performs the read only in the serialization for $b = \hat{b}$. If p_k is in the same state whether $b = 0$ or $b = 1$, then p_k performs the read only in the serialization for $b = 1$. This results in a (non-uniform) EREW PRAM algorithm that performs the same computation as Algorithm A, using the same number of processors and the same working space, and runs in time $O(T)$. The length of the program is the length of the serialization, which is $O(T)$.

There was no attempt to minimize the constants in the above algorithm. Techniques similar to those applied in the proof of Lemma 8.1 can be used here to reduce the constants. ∎

We now show that randomization cannot help too much when the input domain is small.

Lemma 8.3 *Let T_d be a lower bound on the time required by a deterministic algorithm to solve a problem P with input taken from a domain of size $|I|$. Then, for any randomized algorithm that solves P, the expected running time T_r on any input is bounded by $T_r \geq T_d/|I|$.*

Proof. Let T_a be the average running time for the uniform input distribution, minimized over all possible deterministic algorithms, to solve P. Clearly, since the number of possible inputs is $|I|$, $T_a \geq T_d/|I|$. Further, by a classic result of Yao [Yao77], $T_r \geq T_a$. (Yao's result is more general; for a short proof of this claim see [FMRW85].) Therefore, $T_r \geq T_a \geq T_d/|I|$. ∎

8.2 Lower bounds for broadcasting and related problems

Beame, Kik and Kutylowski [BKK94] showed that computing the broadcasting problem on a non-uniform EREW PRAM with unbounded program size, an unbounded number of processors, and unbounded space requires $\Omega(\lg n)$ time. The results of the previous subsection give us the following theorem.

Theorem 8.4 *Any deterministic or randomized algorithm that computes the broadcasting problem into n memory locations on a* QRQW PRAM *with an unbounded number of processors and unbounded space requires expected time $\Omega(\lg n)$.*

Proof. The lower bound for deterministic algorithms follows by the lower bound in [BKK94] and Lemma 8.2 since the size of the input domain for the broadcasting problem is 2. The lower bound for randomized algorithms follows by Lemma 8.3. ∎

Since a CREW PRAM can broadcast into n memory locations in constant time, Theorem 8.4 immediately implies the following separation results:

Corollary 8.5 *There is an $\Omega(\lg n)$ time separation between a (deterministic or randomized)* QRQW PRAM *with arbitrarily many processors and a (deterministic or randomized)* CRQW PRAM. *The same separation result holds for a Queue-Read, Exclusive-Write (*QREW*)* PRAM *relative to a* CREW PRAM.

In the section on load balancing we will make use of the following generalization of the broadcasting problem.

Theorem 8.6 *Any deterministic or randomized algorithm that broadcasts the value of a bit to any subset of k processors in a* QRQW PRAM *requires expected time $\Omega(\lg k)$.*

Proof. Let Algorithm A be a QRQW algorithm that succeeds in broadcasting the value of a bit to some subset of k processors in time t. We use Algorithm A to derive a (non-uniform) QRQW PRAM algorithm for the broadcasting problem into k fixed memory locations as follows. We first run Algorithm A to broadcast the value of the bit to some subset of k processors. We then transmit the value of the bit from the ith processor in the subset to the ith output memory location, $1 \leq i \leq k$. This can be performed in one step with time cost 1 since we can precompute from Algorithm A the exact indices of the k processors to which the value of the bit will be transmitted. Thus we can solve the broadcasting problem in $t+1$ steps. It follows from Theorem 8.4 that $t = \Omega(\lg k)$. ∎

9 Conclusions

This paper has proposed a new model for shared memory machines, the QRQW PRAM model, that takes into account the amount of contention in memory accesses. This model is motivated by the contention characteristics of currently available commercial machines. We have presented several results for this model, including a fast, work-preserving emulation of the QRQW PRAM on hypercube-type, noncombining networks, a work-time framework and some automatic processor allocation schemes for the model, several linear work, sub-logarithmic time algorithms for the fundamental problems of leader election on a CRQW PRAM and linear compaction on a QRQW PRAM, and some lower bounds.

In a companion paper [GMR96a], we present many new results for the QRQW PRAM. Among the algorithmic results presented are low-contention, fast, work-optimal QRQW algorithms for multiple compaction, load balancing, generating a random permutation and parallel hashing. These results and the results presented in this paper demonstrate the advantage of the QRQW over the EREW. Together with the penalty in running high-contention CRCW or CREW algorithms on existing machines, this supports the QRQW PRAM as a more appropriate model for high-level algorithm design.

Finally, in related work [GMR96b] we explore the properties of the asynchronous QRQW PRAM.

Acknowledgements

Richard Cole, Albert Greenberg, Maurice Herlihy, Honghua Yang, and the anonymous referees provided useful comments on this work.

References

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. 1990 International Conf. on Supercomputing*, pages 1–6, June 1990.

[AKP91] F. Abolhassan, J. Keller, and W. J. Paul. On the cost-effectiveness of PRAMs. In *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing*, pages 2–9, December 1991.

[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.

[BCH+93] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proc. 4th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 102–111, May 1993.

[Bel92] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):26–47, 1992.

[BKK94] P. Beame, M. Kik, and M. Kutyłowski. Information broadcasting by exclusive-write PRAMs. *Parallel Processing Letters*, 4(1&2):159–169, 1994.

[Ble89] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, C-38(11):1526–1538, 1989.

[Ble93] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *A Synthesis of Parallel Algorithms*, chapter 1, pages 35–60. Morgan-Kaufmann, San Mateo, CA, 1993.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier Science Publishing Co., Inc., New York, 1976.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–208, 1974.

[CDR86] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[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.

[Cyp88] R. Cypher. Valiant's maximum algorithm with sequential memory accesses. Technical Report TR 88-03-08, Department of Computer Science, University of Washington, Seattle, Washington, April 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.

[DK92] S. R. Dickey and R. Kenner. Hardware combining and scalability. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 296–305, June-July 1992.

[DKN93] W. J. Dally, J. S. Keen, and M. D. Noakes. The J-Machine architecture and evaluation. In *Proc. 1993 IEEE Compcon Spring*, pages 183–188, February 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.

[DS92] R. Drefenstedt and D. Schmidt. On the physical design of butterfly networks for PRAMs. In *Proc. 4th IEEE Symp. on the Frontiers of Massively Parallel Computation*, pages 202–209, October 1992.

[FBR93] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR1: Bridging the gap between shared memory and MPPs. In *Proc. 1993 IEEE Compcon Spring*, pages 285–294, February 1993.

[FKL+92] F. Fich, M. Kowaluk, K. Loryś, M. Kutylowski, and P. Ragde. Retrieval of scattered information by EREW, CREW, and CRCW PRAMs. In *Proc. 3rd Scandinavian Workshop on Algorithm Theory, Springer LNCS 621*, pages 30–41, 1992.

[FMRW85] F.E. Fich, F. Meyer auf der Heide, P.L. Ragde, and A. Wigderson. One, two, three,..., infinity: Lower bounds for parallel computation. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 48–58, 1985.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, May 1978.

[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.

[Gib93] P. B. Gibbons. Asynchronous PRAM algorithms. In J. H. Reif, editor, *A Synthesis of Parallel Algorithms*, chapter 22, pages 957–997. Morgan-Kaufmann, San Mateo, CA, 1993.

[GM91] J. Gil and Y. Matias. Fast hashing on a PRAM—designing by expectation. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 271–280, 1991.

[GM96] J. Gil and Y. Matias. An effective load balancing policy for geometric decaying algorithms. *Journal of Parallel and Distributed Computing*, 1996. To appear.

[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.

[GMR94a] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. In *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 236–247, June 1994.

[GMR94b] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 638–648, January 1994.

[GMR94c] L. A. Goldberg, Y. Matias, and S. Rao. An optical simulation of shared memory. In *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 257–267, June 1994.

[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 asynchronous PRAM model. In *Proc. of EURO-PAR'96, Springer LNCS*, August 1996. To appear. Preliminary version appears in *QRQW: Accounting for Concurrency in PRAMs and Asynchronous PRAMs*, AT&T Bell Laboratories Technical Report, March 1993.

[GMV91] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 698–710, October 1991.

[Goo91] M.T. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 711–722, 1991.

[Gre82] A.G. Greenberg. On the time complexity of broadcast communication schemes. In *Proc. 14th ACM Symp. on Theory of Computing*, pages 354–364, 1982.

[Hoe63] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.

[IBM94] IBM Corporation. *IBM Scalable POWERparallel Systems 9076 SP2 and Enhancements for SP1*, April 1994. Hardware announcement.

[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.

[KS93] R. E. Kessler and J. L. Schwarzmeier. CRAY T3D: A new dimension for Cray research. In *Proc. 1993 IEEE Compcon Spring*, pages 176–182, February 1993.

[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.

[Lei92a] F. T. Leighton. Methods for message routing in parallel machines. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 77–96, May 1992. Invited paper.

[Lei92b] C. E. Leiserson *et al.* The network architecture of the Connection Machine CM-5. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 272–285, June-July 1992.

[LF80] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th International Symp. on Computer Architecture*, pages 148–159, May 1990.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[Mac94] P. D. MacKenzie. Personal communication, August 1994.

[Mas91] MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, CA 94086. *MasPar System Overview, document 9300-0100, revision A3*, March 1991.

[Mat92] Y. Matias. *Highly Parallel Randomized Algorithmics.* PhD thesis, Tel Aviv University, Israel, 1992.

[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.

[MR96] P.D. MacKenzie and V. Ramachandran. ERCW PRAMs and optical communication. In *Proc. of EURO-PAR'96, Springer LNCS*, August 1996. To appear.

[MV91] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 307–316, May 1991.

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

[PN85] G. F. Pfister and V. A. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Trans. on Computers*, C-34(10):943–948, 1985.

[Pre92] L. Prechelt. Measurements of MasPar MP-1216A communication operations. Technical report, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Karlsruhe, Germany, November 1992.

[Ran89] A. G. Ranade. *Fluent parallel computation.* PhD thesis, Department of Computer Science, Yale University, New Haven, CT, May 1989.

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

[Sny86] L. Snyder. Type architecture, shared memory and the corollary of modest potential. *Annual Review of CS*, I:289–317, 1986.

[SV94] M. Schmidt-Voigt. Efficient parallel communication with the nCUBE 2S processor. *Parallel Computing*, 20(4):509–530, 1994.

[Val90a] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Val90b] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A*, pages 943–972. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[Val92] L. G. Valiant. A combining mechanism for parallel computers. Technical Report TR-24-92, Harvard University, Cambridge, Massachusetts, November 1992.

[Yao77] A.C. C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 222–227, 1977.