Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? *

Phillip B. Gibbons[†] Yossi Matias[‡] Vijaya Ramachandran[§]

Abstract

We consider the question of whether a shared-memory model can serve as an effective bridging model for parallel computation along the lines of a distributed-memory model such as the BSP. As a candidate for a shared-memory bridging model, we introduce the Queuing Shared Memory (QSM) model, which accounts for limited communication bandwidth while still providing a simple shared-memory abstraction. We substantiate the ability of the QSM to serve as a bridging model by providing a simple work-preserving emulation of the QSM on both the BSP, and on a related model, the $(d, \mathbf{x})$-BSP. We present evidence that the features of the QSM are essential to its effectiveness as a bridging model. In addition, we describe scenarios in which the high-level QSM is more suited for analyzing algorithms on certain machines than the more detailed BSP and LogP models. Finally, we present algorithmic results for the QSM, as well as general strategies for mapping algorithms designed for the BSP or PRAM models onto the QSM model. Our main conclusion is that shared-memory models can potentially serve as viable alternatives to existing message-passing, distributed-memory bridging models.

1 Introduction

A fundamental challenge in parallel processing is to develop effective models for parallel computation, at suitable levels of abstraction. Effective and widely-used models would provide standards that could be relied upon by application programmers, algorithm designers, software vendors, and hardware vendors, making parallel machines cheaper to build and easier to use. Effective models must balance simplicity, accuracy, and broad applicability. In particular, a simple, "bridging" model, i.e., a model that spans the range from algorithm design to architecture to hardware, is an especially desirable one.

*To be presented in SPAA'97.

†Bell Laboratories (Lucent Technologies), 600 Mountain Ave, Murray Hill NJ 07974. email: gibbons@research.bell-labs.com

‡Bell Laboratories (Lucent Technologies), 600 Mountain Ave, Murray Hill NJ 07974. email: matias@research.bell-labs.com

§Dept. of Computer Sciences, University of Texas at Austin, Austin TX 78712. email: vlr@cs.utexas.edu. This work was supported in part by NSF grant CCR/GER-90-23059.

A large number of models for parallel computation have been proposed and studied in the last twenty years (see the full version of this paper [19] and the references therein). These models differ in what aspects of parallel machines are exposed. Given this plethora of models, it is natural to seek to distinguish a few models with the most promise, and concentrate on these models. Advocates such as Vishkin [40], Kennedy [29], Smith [37], and Blelloch [8] have long presented arguments in support of the shared-memory abstraction. On the other hand, shared-memory models have been criticized for years for failing to model essential realities of parallel machines. In particular, the PRAM model [13], in which processors execute in lock-step and communicate by unit-time reads and writes to locations in a shared memory, has been faulted (among other failings) for completely ignoring the bandwidth limitations of parallel machines. Until recently, there were few attractive alternatives, so shared-memory models such as the PRAM remained the most widely used models for the design and analysis of parallel algorithms (see, e.g. [23, 28, 36]). However, in the last few years, new alternatives such as the BSP [38] and LogP [10] models have gained considerable popularity. These abstract network models support point-to-point message-passing, can directly support a distributed-memory abstraction, and account for bandwidth limitations using a 'gap' parameter. Given these new, more realistic models, there is a temptation to declare all shared-memory models too unrealistic, and not worthy of further study or consideration.

In this paper we challenge this perception and consider the question of whether a shared-memory model can in fact serve as an effective bridging model for parallel computation. In particular, can a shared-memory model be as effective as, say, the BSP? As a candidate for a bridging model, we introduce the Queuing Shared Memory (QSM) model, which accounts for limited communication bandwidth while still providing a simple shared-memory abstraction. In a nutshell, the QSM model consists of processors with individual private memory as well as a global shared memory. Access to shared memory is more expensive than access to local memory or a computation step, reflecting bandwidth limitations. The choice of the QSM model is based on the observation that while overheads due to latency, synchronization, and memory granularity can be effectively diminished by using slackness and pipelining, the bandwidth overhead is inherent and hence should be accounted for directly. Thus, the QSM is envisioned as a "minimal" shared-memory model that can be competitive with the BSP. Similarly, the memory contention rule of the QSM is the queuing contention rule, as in the QRQW PRAM [20]. This rule is strong enough to provide the QSM with an expressive power comparable to that of the BSP, but it is not too strong to prevent a fast and efficient emulation of the QSM on the BSP with the techniques we use.

As advocated in [38, 40] and elsewhere, one reasonable goal for a high-level, shared-memory model is that it allow for efficient emulation on lower-level, seemingly more realistic, models. If the overheads in the emulation are small, then the high-level model becomes an attractive general-purpose bridging model. We substantiate the ability of the QSM to serve as a bridging model by providing a simple work-preserving emulation of the QSM on both the BSP, and on a related model, the $(d, \mathbf{x})$-BSP [9], and arguing for the practicality of this emulation. Thus the QSM can be effectively realized on machines that can effectively realize the BSP, as well as on machines that are better modeled by the $(d, \mathbf{x})$-BSP. We also describe scenarios in which the high-level QSM is more suited for analyzing algorithms on certain machines than the more detailed BSP and LogP models, due to the fact that the memory layout is different than the one perceived by the BSP and LogP.

We present several algorithmic results for the QSM. We note that any EREW or QRQW PRAM algorithm can be mapped onto the QSM with a factor of g increase in time and work. We also show that for many linear-work QRQW PRAM algorithms, this increase in work in the QSM algorithm is unavoidable, and we present some other lower bounds for the QSM. We consider the mapping of the BSP onto the QSM when the bandwidth parameter, g, is the same for both models. We show that many, though not all, BSP algorithms map onto the QSM step-by-step, resulting in algorithms whose time and work bounds match the bounds on a BSP whose latency parameter, L, is set to 1. We also present a work-preserving randomized emulation of the BSP on the QSM with a logarithmic slowdown. This result implies that any n-processor BSP algorithm that takes time $t(n)$ (when L is set to 1) can be mapped onto the QSM to run in time $O(t(n) \lg n)$ w.h.p. using $n/\lg n$ processors, and more generally on a p-processor QSM to run in time $O(t(n) \cdot (n/p + \lg n))$ w.h.p.

Our main conclusion is that shared-memory models can potentially serve as viable alternatives to existing message-passing or distributed-memory bridging models. While this paper focuses on a shared-memory model that would be competitive with the BSP, a similar approach can be taken with regard to other message-passing bridging models mentioned above (or others), that may emphasize other features than the ones emphasized by the BSP.

The rest of the paper is organized as follows. In Section 2, we describe the Queuing Shared Memory model, and qualitatively compare it with previous models, and in particular, with the BSP. In Section 3, we present work-preserving emulations of the QSM on the BSP and on the $(d, \mathbf{x})$-BSP, and discuss the practicality of these emulations. In Section 4, we provide a few scenarios where the QSM is a more accurate model than the more detailed BSP and LogP. Section 5 presents algorithmic results and issues related to algorithm design on the QSM.

Finally, we refer the reader to the position paper [16], which provides a non-technical overview of much of this work in arguing the importance of shared-memory models in general and the QSM model in particular.

2 The Queuing Shared Memory model

In this section, we describe the Queuing Shared Memory model, and elaborate on some of its features.

Definition 2.1 *The* Queuing Shared Memory (QSM) *model consists of a number of identical processors, each with its own private memory, communicating by reading and writing locations in a shared memory. Processors execute a sequence of synchronized phases, each consisting of an arbitrary interleaving of the following operations:*

1. *Shared-memory reads: Each processor i copies the contents of r_i shared-memory locations into its private memory. The value returned by a shared-memory read can only be used in a subsequent phase.*
2. *Shared-memory writes: Each processor i writes to w_i shared-memory locations.*
3. *Local computation: Each processor i performs c_i RAM operations involving only its private state and private memory.*

Concurrent reads or writes (but not both) to the same shared-memory location are permitted in a phase. 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 phase.

The restrictions that (i) values returned by shared-memory reads cannot be used in the same phase and that (ii) the same shared-memory location cannot be both read and written in the same phase reflect the intended emulation of the QSM model on a MIMD machine. In this emulation, the shared memory reads and writes at a processor are issued in a pipelined manner, to amortize against the delay (latency) on such machines in accessing the shared memory, and are not guaranteed to complete until the end of the phase. On the other hand, each of the local compute operations are assumed to take unit time in the intended emulation, and hence the values they compute can be used within the same phase.

Each shared-memory location can be read or written by any number of processors in a phase, as in a concurrent-read concurrent-write PRAM model; however, in the QSM model, there is a cost for such contention. In particular, the cost for a phase will depend on the maximum contention to a location in the phase, defined as follows.

Definition 2.2 *The* maximum contention *of a* QSM *phase is the maximum, over all locations x, of the number of processors reading x or the number of processors writing x. A phase with no reads or writes is defined to have maximum contention 'one'.*

One can view the shared memory of the QSM model as a collection of queues, one per shared-memory location; requests to read or write a location queue up and are serviced one-at-a-time. The maximum contention is the maximum delay encountered in a queue. The cost for a phase depends on the maximum contention, the maximum number of local operations by a processor, and the maximum number of shared-memory reads or writes by a processor. To reflect the limited communication bandwidth on most parallel machines, the QSM model provides a parameter, $g \geq 1$, that reflects the *gap* between the local instruction rate and the communication rate.

Definition 2.3 *Consider a* QSM *phase with maximum contention κ. Let $m_{op} = \max_i\{c_i\}$ for the phase, i.e. the maximum over all processors i of its number of local operations, and likewise let $m_{rw} = \max_i\{r_i, w_i\}$ for the phase. Then the time cost for the phase is $\max\{m_{op}, g \cdot m_{rw}, \kappa\}$.*[1] *The* time *of a* QSM *algorithm is the sum of the time costs for its phases. The* work *of a* QSM *algorithm is its processor-time product.*

Note that although the model charges g per shared-memory request at a given processor (the $g \cdot m_{rw}$ term in the cost metric), it only charges 1 per shared-memory request at a given location (the κ term in the cost metric). Note also that our model considers contention only at individual memory locations, not at memory modules. Even though both of these features give more power to the QSM than would appear to be warranted by current technology, our emulation results in Section 3 show that we can obtain a work-preserving emulation of the QSM on the BSP with only a modest slowdown. Thus, these features do capture the computational power achievable by current technology. The discussion in Section 3 provides some intuition for this rather surprising result.

The particular instance of the Queuing Shared Memory model in which the gap parameter, g, equals 1 is essentially the Queue-Read Queue-Write (QRQW) PRAM model defined by the authors [20]. Previous work on the QRQW PRAM [20, 17, 9] has been focused primarily on contention issues, unlike this paper, which is primarily concerned with bridging models and bandwidth issues.

2.1 Model comparison

Table 1 compares the QSM model to a number of other models in the literature. The first column of the table gives the name of the model. The second column indicates the synchrony assumption of the model: *Lock-step* indicates that the processors are fully synchronized at each step, with no cost for the synchronization. *Bulk-synchrony* indicates that there is asynchronous execution between synchronization barriers. Typically the barriers involve all the processors, although this is not necessarily required. Models that permit more general asynchrony are denoted as *asynchronous*. The third column indicates the type of interprocessor communication assumed by the model while fourth column indicates the parameters in the model.

Unlike the previous models shown in Table 1, the QSM provides bulk-synchrony, a shared-memory abstraction, and just two parameters. In all, the key features of the QSM that make it an attractive candidate for a bridging model are:

[1] Alternatively, the time cost could be $m_{op} + g \cdot m_{rw} + \kappa$; this affects the bounds by at most a factor of 3, and the results in [9] show that at least for certain machines, taking the maximum is more accurate than taking their sum.

1. Shared-memory abstraction. The QSM provides the simplicity of a shared-memory abstraction in which the shared memory is viewed as a collection of independent cells, non-local to the processors. The shared-memory abstraction is similar to the view of memory in sequential programming (the familiar read/write semantics). It is also the abstraction of choice for the small symmetric multiprocessors (SMPs) found in current microprocessors. There are high-performance parallel machines such as the Cray C90, Cray J90, and Tera MTA that also directly support a shared-memory abstraction. Thus as a bridging model, it provides for the smoothest transition from sequential programming to programming small SMPs to programming larger parallel machines (MPPs).

2. Bulk-synchrony. The QSM supports bulk-synchronous operation, in which processors operate asynchronously between barrier synchronizations. This allows a QSM algorithm to synchronize less frequently than algorithms designed for a lock-step model, which makes for a more efficient mapping of the algorithm to MIMD machines.

3. Few parameters. For simplicity, it is desirable for bridging models to have only a few parameters. As evidenced by [10, 14, 27] and elsewhere, having additional parameters in a model can make it quite difficult to obtain a concise analysis of an algorithm. On the other hand, it is desirable to have whatever parameters are essential for a desired level of accuracy in modeling machines realizing the bridging model. The QSM has only two parameters: one reflecting the number of processors and one reflecting the limited communication bandwidth. The results in the next section provide evidence that a parameter reflecting limited bandwidth should be in a high-level model, and that other communication parameters are not necessary. For this reason, we believe that g is a better choice for a second parameter than the ℓ, s, L, or I parameters found in other models.

4. Queue contention metric. The "queue-read queue-write" (QRQW) contention rule of the QSM model more accurately reflects the contention properties of parallel machines with simple, non-combining interconnection networks than either the well-studied exclusive-read exclusive-write (EREW) or concurrent-read concurrent-write (CRCW) rules. As argued in [20], the EREW rule is too strict, and the CRCW rule ignores the large performance penalty of high contention steps. Indeed, for most existing machines, the contention properties of the machine are well-approximated by the queue-read, queue-write rule. The queue-read queue-write contention metric can lead to faster algorithms, since it does not ignore the aforementioned penalty for high contention steps and yet it allows for low-contention algorithms that are not permitted under the EREW rule.

5. Work-preserving emulation on BSP. The BSP is a distributed memory, message passing model that is gaining acceptance as a bridging model for parallel computation. Thus a work-preserving emulation of the QSM on the BSP is a strong validating point for this shared-memory model. In Section 3 we present a randomized work preserving emulation of the QSM on the BSP and the related $(d, \mathbf{x})$-BSP that works with high probability and only a small slowdown.

6. Work-preserving emulation of BSP. In addition to the work-preserving emulation of QSM on BSP we observe

Comparison of Models of Parallel Computation			
model	**synchrony**	**communication**	**parameters**
PRAM [13]	lock-step	shared memory	p
Module Parallel Computer (MPC) [34]	lock-step	distributed memory	p
LPRAM [2]	lock-step	shared memory	p, ℓ
Phase LPRAM [15]	bulk-synchrony	shared memory	p, ℓ, s
XPRAM [39]	bulk-synchrony	message-passing	p, g, L
Bulk-Synchronous Parallel (BSP) [38]	bulk-synchrony	message-passing	p, g, L
Postal model [4]	asynchronous	message-passing	p, ℓ
LogP model [10]	asynchronous	message-passing	p, g, ℓ, o
QRQW Asynchronous PRAM [18]	asynchronous	shared memory	p
QRQW PRAM [20]	bulk-synchrony	shared memory	p
Block Distributed Memory (BDM) [24]	bulk-synchrony	distributed memory	p, g, L, B
PRAM(m) model [32]	lock-step	shared memory	p, m
Interval model [31]	bulk-synchrony	message-passing	p, I
Queuing Shared Memory (QSM)	bulk-synchrony	shared memory	p, g

Table 1: *A comparison of several models of parallel computation.* Here p is the number of processors, ℓ is the latency, s is the cost for a barrier synchronization, L is the sum of ℓ and s, g is the bandwidth gap, o is the overhead at the processor to send or receive a message, B is the block size (i.e. the number of consecutive cells sent on a write or retrieved on a read), m is the number of shared-memory cells available for both reading and writing, and I is the maximum of ℓ, g, and s.

that there is a work-preserving mapping in the reverse direction as well. Many BSP algorithms map onto the QSM in a step-by-step manner with performance corresponding to the case when the periodicity parameter on the BSP is set to 1. While it is possible for BSP algorithms not to have this property, we also present a work-preserving emulation of the BSP on the QSM with only a small slowdown. This emulation holds for all BSP algorithms. This is discussed in more detail in Section 5.

The PRAM(m) model shares many of the same goals as the QSM model. As shown in the table, the PRAM(m) provides a shared-memory abstraction and just two parameters: one for the number of processors and one that captures the limited communication bandwidth ($g = p/m$). However, the PRAM(m) model is suitable only for lower bounds since it does not charge for contention to the m shared-memory locations or reads on the input values.

3 Emulations of QSM on BSP models

The Bulk-Synchronous Parallel (BSP) model [38, 39] consists of p processor/memory components communicating by sending point-to-point messages. The interconnection network supporting this communication is characterized by a bandwidth parameter g and a latency parameter L. A BSP computation consists of a sequence of "supersteps" separated by bulk synchronizations. In each superstep the processors can perform local computations and send and receive a set of messages. Messages are sent in a pipelined fashion, and messages sent in one superstep will arrive prior to the start of the next superstep. The time charged for a superstep is calculated as follows. Let w_i be the amount of local work performed by processor i in a given superstep. Let s_i (r_i) be the number of messages sent (received) by processor i. Let $w = \max_{i=1}^{p} w_i$, and $h = \max_{i=1}^{p}(\max(s_i, r_i))$. Then the cost, T, of a superstep is defined to be $T = \max(w,\ g \cdot h,\ L)$.

The $(d, \mathbf{x})$-BSP [9] is a model similar to the (distributed-memory view of the) BSP, but it provides a more detailed modeling of memory bank contention and delay. In [9], it is argued that the $(d, \mathbf{x})$-BSP more accurately models shared-memory machines with a high-bandwidth communication network and more memory banks than processors than the BSP does. Such machines include Cray C90, Cray J90 and Tera MTA (experimental validation of this accuracy claim is provided for Cray C90 and Cray J90). The $(d, \mathbf{x})$-BSP is parameterized by five parameters, p, g, L, d and $\mathbf{x}$, where p, g and L are as in the original BSP model, the *delay* d is the 'gap' parameter at the memory banks, and the *expansion* $\mathbf{x}$ is the ratio of memory banks to processors (i.e., there are $\mathbf{x} \cdot p$ memory banks). Consider a superstep where w is the maximum local work performed by a processor, h_s is the maximum number of read/write requests by a processor and h_r is the maximum number of read/write requests to a memory bank. Then the time, T, charged by the $(d, \mathbf{x})$-BSP for this superstep is $T = \max(w,\ g \cdot h_s,\ d \cdot h_r,\ L)$. The original BSP can be viewed as a $(d, \mathbf{x})$-BSP with $d = g$ and $\mathbf{x} = 1$.

In this section we present two emulations of the QSM on the $(d, \mathbf{x})$-BSP. The first emulation is for a so-called *balanced* $(d, \mathbf{x})$-BSP, in which $\mathbf{x} \geq d/g$, and is work optimal. Since the BSP is a balanced $(d, \mathbf{x})$-BSP, this optimal emulation applies also for the BSP. The second emulation is for an *unbalanced* $(d, \mathbf{x})$-BSP, in which $\mathbf{x} < d/g$. This emulation suffers from work inefficiency which is proportional to the "imbalance-factor", $d/(g\mathbf{x})$. We show by a lower bound argument that this overhead is unavoidable.

The two emulations are in fact identical, and differ only in the *slackness* parameter. We first present the algorithm, followed by the different analysis for the two cases mentioned above, and concluding with the lower bound.

3.1 The emulation algorithm

A work-preserving emulation of a model A on a model B provides a formal proof that model A can be realized on model B with only a constant factor overhead in work. If model B is considered to be reflective of an interesting class of parallel machines, then such an emulation supports the use of A as a bridging model, *as long as the emulation can be considered "practical"*. For the QSM on the $(d, \mathbf{x})$-BSP (and hence on the BSP), we present a very simple emulation algorithm and then discuss its practicality in some detail.

The emulation algorithm of a v-processor QSM on a p-processor $(d, \mathbf{x})$-BSP, $v \geq p$, is quite simple, and it is similar to emulations that were previously proposed for the PRAM. Unlike previous emulations, our analysis needs to handle the gap parameter in the emulated machine.

- The shared address space of the QSM is randomly hashed into the $\mathbf{x}p$ memory banks of the $(d, \mathbf{x})$-BSP (or to the p memory modules of the BSP).
- In each phase, each processor of the $(d, \mathbf{x})$-BSP emulates v/p processors of the QSM.

In the work-preserving emulation, each phase i of time t_i on the QSM is emulated on the $(d, \mathbf{x})$-BSP (or simply the BSP) in time $O((v/p) \cdot t_i)$, regardless of the distribution of shared memory reads and writes. The needed parallel slackness, v/p, is modest, and does not depend on the maximum contention in a phase (which may be much larger than v/p).

Note that if a computer system already hashes the data using a pseudo-random hash function, then the emulation is nothing but the straightforward implementation of an algorithm whose parallelism is larger than the number of processors. Several parallel database systems already hash their data using pseudo-random hash functions. The Tera MTA provides hardware support for hash functions to be used for pseudo-random mapping of memory locations to memory banks; the Fujitsu μ-VP on the Meiko node already has optional hardware hashing. For other machines, computing a pseudo-random hash in software is feasible. For example, it is shown in [9] that the overhead to compute a certain provably-good (i.e., 2-universal) pseudo-random hash function on the Cray C90 averages 1.8 clock cycles. Also as noted in [9], for some algorithms it is possible to get the same effect without memory hashing, by randomly permuting the input and some of the intermediate results. In others, the nature of the algorithm results in random mapping without any additional steps.

It is well known that hashing destroys spatial locality, but not temporal locality. Spatial locality enables long messages to be sent between components, thereby minimizing overheads on many machines. Some models, such as BDM [24], LogGP [3], and BSP* [6, 5], account for advantages in long messages; most others, e.g., QSM, BSP, $(d, \mathbf{x})$-BSP and LogP, do not. Thus the QSM shares with the BSP, $(d, \mathbf{x})$-BSP and LogP models a disregard for spatial locality. Spatial locality can also arise in initial data placement. Here the input can be assumed to be distributed among the private memories of the QSM processors as among the local memories of the BSP, $(d, \mathbf{x})$-BSP or LogP processors.

The emulation of v/p virtual processors by each physical processor can be done by a variety of techniques. The primary technique is multithreading, in which each virtual processor is its own process, and the physical processor context switches between these processes. The Tera MTA provides hardware support for this multithreading, minimizing the context switching costs. Alternatively, such multithreading can be performed in software. Note that in the QSM, as in other bulk-synchronous models, each virtual processor issues a series of memory requests in a phase. Instead of context switching at each memory request, the multithreading can be performed by executing all the code for the first virtual processor in this phase, then switching to the second virtual processor, and so forth, so that only v/p context switches are needed for the entire phase (this description assumes that storing values returning in response to shared-memory read requests does not require a context switch).

In order to minimize the overheads, it is very important to minimize the amount of parallel slackness required. In the worst case, multithreading v/p processes per machine processor results in v/p times the storage demand at each level of the processor's memory hierarchy, possibly resulting in various thrashing effects. The emulation of the QSM on the BSP requires only $\max(g \lg p, L/g)$ slackness; on the $(d, \mathbf{x})$-BSP, as little as $\max(d, L/g)$ slackness may be required. Note that the L/g term matches the limit on multithreading imposed by the LogP model [10].

Thus, overall, the constants hidden by the big-O notation in the emulation result are small, and hence the emulation can arguably be considered practical.

3.2 Work-preserving QSM emulation on $(d, \mathbf{x})$-BSP

The following theorem presents an emulation of the QSM on a $(d, \mathbf{x})$-BSP for the case when $\mathbf{x} \geq d/g$, where g is the gap parameter for both the QSM and the $(d, \mathbf{x})$-BSP. The emulation is work-preserving for any g (*i.e.* the work performed on the $(d, \mathbf{x})$-BSP is within constant factors of the work performed on the QSM).

Theorem 3.1 (work-preserving QSM emulation) *Consider a p-processor $(d, \mathbf{x})$-BSP with gap parameter g and periodicity factor L, such that $d_g \leq \mathbf{x} \leq p^{\bar{c}}$, for some constant $\bar{c} > 0$, where $d_g = d/g \geq 1$. Let*

$$\delta = \begin{cases} d \lg p & \text{if } d_g \leq \mathbf{x} \leq 2d_g \\ d \lg p / \lg(\mathbf{x}/d_g) & \text{if } 2d_g \leq \mathbf{x} \leq p d_g \\ d & \text{if } \mathbf{x} \geq p d_g \end{cases}$$

Then for all $p' \geq \max(\delta, L/g) \cdot p$, each step of an algorithm for the p'-processor QSM with gap parameter g with time cost t can be emulated on the p-processor $(d, \mathbf{x})$-BSP in $O((p'/p) \cdot t)$ time w.h.p.

This result is not implied by previous simulation results for the QRQW PRAM [20, 9], since these previous results considered standard PRAM models with no gap parameter and BSP or $(d, \mathbf{x})$-BSP models with a small constant gap parameter (that was hence ignored as part of the big-O notation). The question of how the work-efficiency and/or slowdown of the emulation depended upon the gap parameters was not studied. Since we are considering the same gap parameter, g, for the QSM as for the BSP, one might conjecture that considering the gap parameter does not substantially alter the bounds of the simulations without the gap parameter. However, note that the QSM model charges κ for contention κ, regardless of the gap or delay parameters, and indeed a QSM step with time t can have t/g memory requests per processor and maximum contention t. In contrast, in such cases the BSP charges at least $g \cdot t$ and the $(d, \mathbf{x})$-BSP charges at least $d \cdot t$. Viewing the mapping of memory locations to memory banks as tossing weighted balls into bins (where the weight of a ball corresponds to the contention of the location), this implies a different mix of balls than considered in previous emulations.

Since in the original BSP, $d_g = x = 1$, we obtain:

Corollary 3.2 (work-preserving QSM emulation) *A p'-processor* QSM *with gap parameter g can be emulated on a p-processor* BSP *with gap parameter g and periodicity parameter L in a work-preserving manner w.h.p. provided $p' \geq \max(g \lg p, L/g) \cdot p$.*

Proof of Theorem 3.1 (Sketch) We now prove the theorem. The proof is similar to that in [9], extended and adjusted to properly account for the gap parameter in the QSM and to improve upon the results for large values of $\mathbf{x}$, even for the case studied previously of $g = 1$.

The shared memory of the QSM is randomly hashed onto the $B = \mathbf{x} \cdot p$ memory banks of the $(d, \mathbf{x})$-BSP. In the emulation algorithm, each $(d, \mathbf{x})$-BSP processor executes the operations of p'/p QSM processors. We first assume that $\mathbf{x} \geq 2d_g$.

Consider the ith step of the QSM algorithm, with time cost t_i. Let $c > 0$ be some arbitrary constant, and let $\alpha = \max\{c + \bar{c} + 1, e\}$. We will show that this step can be emulated on the $(d, \mathbf{x})$-BSP in time at most $\alpha(p'/p)t_i$ with probability at least $1 - p^{-c}$. Note that by the QSM cost metric, the maximum number of local operations at a processor this step is t_i.

If there are no shared memory reads or writes in step i, then the $(d, \mathbf{x})$-BSP can emulate the step without communicating or synchronizing, by each processor emulating at most $(p'/p)t_i$ local operations, in time $(p'/p)t_i$. So assume that step i performs at least one shared memory read or write, and hence $t_i \geq g$.

By the definition of the QSM cost metric, we have that κ, the maximum number of requests to the same location, is at most t_i, and h_s, the maximum number of requests by any one processor, is at most t_i/g. For the sake of simplicity in the analysis, we add dummy memory requests to each processor as needed so that it sends exactly t_i/g memory requests this step. The dummy requests for a processor are to dummy memory locations, with processor ℓ sending all its dummy requests to dummy location ℓ. In this way, the maximum number of requests to the same location, κ, remains at most t_i, and the total number of requests is $Z = p't_i/g$.

Let $i_1, i_2, \ldots, i_m$ be the different memory locations accessed in this step (including dummy locations), and let κ_j be the number of accesses to location i_j, $1 \leq j \leq m$. Note that $\sum_{j=1}^{m} \kappa_j = Z$. Consider a memory bank β. For $j = 1, \ldots, m$, let x_j be an indicator binary random variable which is 1 if memory location i_j is mapped onto the memory bank β, and is 0 otherwise. Thus, $\mathbf{Prob}(x_j = 1) = 1/B$. Let $a_j = \kappa_j/t_i$; a_j is the normalized contention to location j. Since $\kappa \leq t_i$, we have that $a_j \in (0, 1]$. Let $\Psi_\beta = \sum_{j=1}^{m} a_j x_j$; Ψ_β, the normalized request load to bank β, is the weighted sum of Bernoulli trials. The expected value of Ψ_β is

$$\mathbf{E}(\Psi_\beta) = \sum_{j=1}^{m} \frac{a_j}{B} = \frac{1}{\mathbf{x}p} \sum_{j=1}^{m} \frac{\kappa_j}{t_i} = \frac{1}{\mathbf{x}p} \cdot \frac{Z}{t_i} = \frac{p'\,t_i}{\mathbf{x}\,p\,t_i\,g} = \frac{p'}{\mathbf{x}pg}.$$

Let h_r^β be the total number of requests to locations mapped to bank β. To show that it is highly unlikely that h_r^β greatly exceeds this expected value, we will use a theorem by Raghavan and Spencer [35], which provides a tail inequality for the weighted sum of Bernoulli trials, to show that

$$\mathbf{Prob}\left(\Psi_\beta > \frac{\alpha p'}{dp}\right) < \frac{p^{-(c+1)}}{\mathbf{x}} .$$

(Details are given in [19].) Note that $h_r^\beta = \sum_{j=1}^{m} x_j k_j = \Psi_\beta \cdot t_i$ and therefore $\mathbf{Prob}\left(h_r^\beta > \frac{\alpha\,p'\,t_i}{d\,p}\right) < \frac{p^{-(c+1)}}{\mathbf{x}}$. Let $h_r = \max_\beta h_r^\beta$. Then $\mathbf{Prob}\left(h_r > \frac{\alpha\,p'\,t_i}{d\,p}\right) < B \cdot \frac{p^{-(c+1)}}{\mathbf{x}} = p^{-c}$. The time of the $(d, \mathbf{x})$-BSP step to emulate QSM step i is $T_i = \max((p'/p)t_i,\ g(p'/p)(t_i/g),\ d \cdot h_r,\ L)$. Since $t_i \geq g$, we have that $(p'/p)t_i \geq (p'/p)g \geq L$ and hence it follows from the above that

$$\mathbf{Prob}\left(T_i \leq \alpha\,(p'/p)\,t_i\right) \geq 1 - p^{-c} .$$

We next consider the case where $d_g \leq \mathbf{x} \leq 2d_g$, and therefore $\delta = d \lg p$. In this case we take $\alpha = \max\{c + \bar{c} + 1, 2e\}$, and the proof proceeds as above except that we make use of the fact that

$$\left(\frac{\alpha \mathbf{x}}{e d_g}\right)^{-\frac{\alpha p'}{dp}} \leq 2^{-\frac{\alpha p'}{dp}} \leq 2^{-\frac{\alpha}{d}\max(d \lg p, L/g)} \leq p^{-\alpha} .$$

This completes the proof of Theorem 3.1. ∎

3.3 Emulating QSM on unbalanced $(d, \mathbf{x})$-BSP

We next consider the case where the bandwidth at the memory banks is less than the bandwidth at the processors and network, *i.e.* $\mathbf{x} < d_g$. We present an emulation whose work bound is within a constant factor of the best possible.

Theorem 3.3 (QSM on unbalanced $(d,\mathbf{x})$-BSP) *Consider a p-processor $(d, \mathbf{x})$-*BSP *with gap parameter g and periodicity factor L, such that $1 \leq \mathbf{x} < \min\left\{d_g, p^{\bar{c}}\right\}$, for some constant $\bar{c} > 0$, where $d_g = d/g$. Then for all $p' \geq \max(\mathbf{x}g \lg p, d, L/g) \cdot p$, each step of an algorithm for the p'-processor* QSM *with parameter g with time cost t can be emulated on the p-processor $(d, \mathbf{x})$-*BSP *in $O((d_g/\mathbf{x}) \cdot (p'/p) \cdot t)$ time w.h.p.*

Proof. (Sketch) As in the proof of Theorem 3.1, the shared memory of the QSM is randomly hashed onto the $B = \mathbf{x} \cdot p$ memory banks of the $(d, \mathbf{x})$-BSP. In the emulation algorithm, each $(d, \mathbf{x})$-BSP processor executes the operations of p'/p QSM processors.

Consider the ith step of the QSM algorithm, with time cost t_i. Let $c > 0$ be some arbitrary constant, and let $\alpha = \max\{c + \bar{c} + 1, 2e\}$. We will show that this step can be emulated on the $(d, \mathbf{x})$-BSP in time at most $\max\{(p'/p)t_i, \alpha(d_g/\mathbf{x})(p'/p)t_i\}$ with probability at least $1 - p^{-c}$.

The proof proceeds exactly as in the proof of Theorem 3.1: we add dummy requests as needed, define indicator binary random variables x_j for each memory bank j, define Ψ_β, and show that $\mathbf{E}(\Psi_\beta) = p'/(\mathbf{x}pg)$. We apply the Raghavan and Spencer theorem to obtain

$$\mathbf{Prob}\left(\Psi_\beta > \frac{\alpha p'}{\mathbf{x}p}\right) < \frac{p^{-(c+1)}}{\mathbf{x}} \ .$$

(Details are given in [19].) It follows as in the previous proof that

$$\mathbf{Prob}\left(h_r > \frac{\alpha\, p'\, t_i}{\mathbf{x}\, pg}\right) < p^{-c} \ ,$$

where h_r is the maximum number of read/write requests to a memory bank. The time, T_i, of the $(d, \mathbf{x})$-BSP superstep to emulate QSM step i is $\max((p'/p)t_i,\ g(p'/p)(t_i/g),\ d \cdot h_r,\ L)$. Since $t_i \geq g$ and $p'/p \geq L/g$, we have that

$$\mathbf{Prob}\left(T_i \leq \max\left\{\frac{p'}{p} \cdot t_i\ ,\ \alpha \cdot \frac{d}{\mathbf{x}}\frac{p'}{pg} \cdot t_i\right\}\right) \geq 1 - p^{-c} \ .$$

The theorem follows. ∎

3.4 A lower bound

The following lower bound shows that the work bound in Theorem 3.3 is tight, as well as showing the importance of having a gap parameter on the QSM. In particular, it implies that a PRAM has an inherent inefficiency overhead of g, when emulated on a BSP or $(d, \mathbf{x})$-BSP with a gap parameter g. Likewise, it implies that g is the minimum gap parameter that should be assigned to the QSM in order to allow for work-efficient emulation on a BSP and $(d, \mathbf{x})$-BSP.

Observation 3.4 *Let $p' \geq p$. Any emulation of one step of the p'-processor* QSM *with gap parameter g' with time cost t on the p-processor $(d, \mathbf{x})$-*BSP *with gap parameter g and periodicity factor L requires $T = \max(t \cdot (g/g') \cdot \lceil p'/p \rceil, d \cdot \lceil tp'/(\mathbf{x}pg') \rceil)$ time in the worst case.*

Proof. Consider a step in which each of the p' QSM processors perform t/g' memory requests, such that all $p't/g'$ requests are to distinct locations in the shared memory. Since there are $m = p't/g'$ locations distributed among $\mathbf{x}p$ memory banks, then regardless of the mapping of locations to banks, there exists at least one bank j which is mapped to by at least $\lceil m/\mathbf{x}p \rceil$ locations. Also, each $(d, \mathbf{x})$-BSP processor sends $\lceil p'/p \rceil \cdot (t/g')$ shared memory requests. Therefore, the time on the $(d, \mathbf{x})$-BSP is at least T. ∎

4 Improved accuracy through the QSM abstraction

In this section, we draw attention to cases where the extra abstraction provided by the QSM may actually result with more accurate modeling. Blelloch *et al.* [9] demonstrated pitfalls in applying existing message-passing or distributed-memory models to machines such as the Cray C90, Cray J90, SGI Power Challenge and Tera MTA. In particular, standard message-passing or distributed-memory models such as the BSP and LogP have the property that the number of memory components is equal to the number of processors. On the other hand, several parallel machines, such as those listed above, have many more memory components than processors, creating a mismatch between such standard models and machines. The abstraction of memory components to shared memory, as assumed in the QSM, make it more robust to changes in the number of memory components.

We elaborate below on how the work-preserving emulation of Section 3, together with the experimental results of [9], indicate general cases for Cray-like machines where the QSM is a more accurate model than the BSP and LogP. We then illustrate this observation by a concrete simple instance.

4.1 Suitability of QSM to Cray-like machines

Processor speeds have been increasing at over 50% a year while memory speeds have been increasing at less than 10% a year [22]. This divergence has motivated several computer manufacturers to design parallel machines with many more memory banks than processors. For example, the 16-processor Cray C90 has 1024 memory banks, the 16-processor Cray J90 has 512 memory banks, the 18-processor SGI Power Challenge has 64 memory banks, and the 256-processor Tera MTA will have 32K memory banks. In order to more accurately model such machines, Blelloch *et al.* [9] introduced the $(d, \mathbf{x})$-BSP model and showed experimentally that it models the Cray C90 and Cray J90 quite accurately, even though the model ignores many details about these machines.

As discussed in [9], the Cray C90 and Cray J90 machines are well suited to the bulk-synchronization provided by BSP-like models, since each of the processors can pipeline hundreds of shared-memory requests, thereby amortizing against the latency, the bank delay (in the absence of high contention), and the cost of synchronizing the processors. It is shown in [9] that accounting for the memory bank delay is critical in predicting running times of algorithms with high memory contention. Therefore, in some situations the BSP and LogP models may provide poor prediction for an algorithm performance, while the $(d, \mathbf{x})$-BSP may provide a good one. An example is shown in Figure 1 for the Cray J90. In this figure, predicted and measured performance are shown on a set of memory access patterns extracted from a trace of Greiner's algorithm for finding the connected components of a graph [21]. Measured times on an 8 processor Cray J90 for several patterns are shown with squares. Predicted times are given for the $(d, \mathbf{x})$-BSP, which models the Cray J90 quite accurately, and the BSP and the LogP, which do not. The contention is given on a logarithmic scale indicating the ratio between the maximum contention, k, and the total

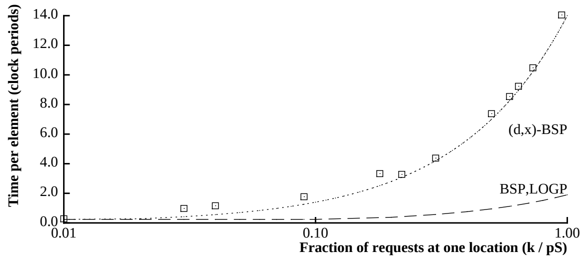

Figure 1: *Inaccuracies in the* BSP *and the* LogP *predictions, due to assuming the wrong memory layout and underestimating the cost of memory bank contention.* This figure is from [9].

number of requests, $p \cdot S$ (p is the number of processors and S is the number of requests sent by each processor).

The QSM is a more high-level model than the BSP or LogP, which in turn are more high-level models than the $(d, \mathbf{x})$-BSP. Nevertheless, the QSM is a better model for machines such as the Cray C90 and Cray J90 than the BSP or the LogP, since its shared-memory abstraction does not assume a particular memory layout. In particular our emulation result of the previous section shows that any algorithm designed for the QSM will map in a work-preserving manner onto the $(d, \mathbf{x})$-BSP given a reasonable amount of parallel slackness, and thus onto these machines. This is because the QSM cost metric accounts for contention to locations, and hence can be translated to a memory layout of any granularity. In contrast, message-passing or distributed-memory models such as the BSP and LogP account only for the aggregated contention per processor, and hence reveal insufficient information to enable a work-preserving emulation unless the slackness is $\geq \mathbf{x}$. (When the slackness is $\geq \mathbf{x}$, then the p-processor distributed-memory model is emulated on a $(d, \mathbf{x})$-BSP with at most p memory banks.) We demonstrate below a simple example where the BSP prediction is entirely inconsistent with the performance on $(d, \mathbf{x})$-BSP.

4.2 Illustrative example

We describe here memory access patterns, A and B, that are indistinguishable on the BSP but have a large difference on both the QSM and $(d, \mathbf{x})$-BSP.

Suppose k processors send one message each to a BSP component C, for arbitrary k. In access pattern A, all requests are directed to the same memory location. In access pattern B, each request is directed to a different memory location within C. In both cases, the cost of the access patterns on the BSP is $g \cdot k$. On the QSM, the cost of access pattern A is $\max(g, k)$, whereas the cost of access pattern B is g. Consider a $(d, \mathbf{x})$-BSP with $\mathbf{x} \geq k$, and suppose that each of the requests in access pattern B is to a different memory bank. Then the cost of access patterns A and B on the $(d, \mathbf{x})$-BSP is $\max(g, d \cdot k)$ and $\max(g, d)$, respectively.

The above example demonstrates a situation where the metric of the BSP is not at all consistent with that of the $(d, \mathbf{x})$-BSP, whereas the QSM maintains close consistency.

5 Algorithmic issues

Since the QSM is a high-level shared-memory model that effectively incorporates bandwidth limitations through the gap parameter, the most effective way of designing good algorithms on the QSM would be to construct them directly for the QSM. However, since we would like to leverage on the extensive literature on PRAM algorithms, in Section 5.1 we discuss the mapping of QRQW PRAM and EREW PRAM algorithms onto the QSM. In Section 5.2 we present some lower bounds, and in Section 5.3 we present some direct QSM algorithms that are faster than the ones obtained by the generic PRAM mapping.

It is also important to consider the mapping of BSP algorithms onto the QSM, for two reasons: First, a good mapping result of this type will allow us to leverage on the results and techniques that were developed for the BSP model. Second, it will demonstrate that the expressive power of QSM is no less than that of the BSP. We study this issue in Section 5.4. In view of a simple lower bound of $\Omega(n \cdot g)$ that we prove in Section 5.2 on the time needed to read n items from global memory into the QSM processors, for these algorithms we assume that the input is distributed among the local memories of the processors in a suitable way. In Section 5.4 we show that any BSP algorithm that is 'well-behaved' (as defined in that section) can be adapted in a simple way to the QSM with no loss in performance. We also demonstrate a general randomized work-preserving emulation of BSP on QSM. Unlike the simple adaptation for 'well-behaved' algorithms, this emulation consists of a fairly involved algorithm and results in logarithmic slow-down. Overall these results demonstrate that any algorithm designed for BSP could be also designed on the QSM, without substantial loss of efficiency.

Finally, in Section 5.5 we discuss the importance of the queuing metric for memory accesses in the QSM model, and note that it is central to its effectiveness as a shared-memory bridging model.

First, we consider the property of *self-simulation* for the QSM, i.e., the problem of simulating a p-processor QSM on a p'-processor QSM, where $p' < p$. The availability of an efficient self-simulation is an important feature for parallel models of computation, since it implies that an algorithm written for a large number of processors is readily portable into a smaller number of processors, without loss of efficiency.

Observation 5.1 *Given a* QSM *algorithm that runs in time t using p processors, the same algorithm can be made to run on a p'-processor* QSM, *where $p' < p$, in time $O(t \cdot p/p')$, i.e., while performing the same amount of work.*

The efficient self-simulation is achieved by the standard strategy of mapping the p processors in the original algorithm uniformly among the p' available processors. In the following, we will state the performance of a QSM algorithm in terms of the fastest time $t(n)$ achievable within a given work bound $w(n)$. When we make such a statement we imply, due to Observation 5.1, that for any p we have an explicit QSM algorithm that runs in $O(t(n) + w(n)/p)$ time using p processors.

In the following we assume that the value of the gap parameter g is less than n, the size of the input; in practice we expect g to be much smaller than n.

5.1 Mapping PRAM algorithms onto the QSM

A naive emulation of a QRQW PRAM algorithm (or an EREW PRAM algorithm, which is a special case) on a QSM with the same number of processors results in an algorithm that is slower by a factor of g. This is stated in the following observation.

Observation 5.2 *Consider a* QSM *with gap parameter g.*

1. *A* QRQW PRAM *algorithm that runs in time t with p processors is a* QSM *algorithm that runs in time at most $t \cdot g$ with p processors.*
2. *A* QRQW PRAM *algorithm in the work-time framework that runs in time t while performing work w immediately implies a* QSM *algorithm that runs in time at most $t \cdot g$ with w/t processors.*

Thus the linear-work QRQW PRAM algorithms given in [20, 17] for *leader election*, *linear compaction*, *multiple compaction*, *load balancing*, and *hashing*, as well as the extensive collection of linear-work logarithmic-time EREW PRAM algorithms reported in the literature, all translate into QSM algorithms with work $O(n \cdot g)$ on inputs of length n with a slowdown by a factor of at most g. We show in Section 5.2 that this increase in work by a factor of g on the QSM may be unavoidable if the input items are not a priori distributed across the QSM processors.

5.2 Lower bounds

If n distinct items need to be read from or written into shared memory on a p-processor QSM then the work performed by the QSM is $\Omega(n \cdot g)$ regardless of the number of processors used. To see this we note that the result is immediate if $p \geq n$ since the QSM has to execute at least one step. If $p < n$ then some processor needs to read or write $\lceil n/p \rceil$ distinct items, and hence that processor spends time $\Omega((n/p) \cdot g)$. Since p processors are used, the work, which is defined as the processor-time product, is $\Omega(ng)$. A similar observation holds for the case when n distinct memory locations are accessed. We state this in the following.

Observation 5.3 *Consider a* QSM *with gap parameter g.*

1. *Any algorithm in which n distinct items need to be read from or written into global memory must perform work $\Omega(n \cdot g)$.*
2. *Any algorithm that needs to perform a read or write on n distinct global memory locations must perform work $\Omega(n \cdot g)$.*

By Observation 5.2 and Observation 5.3, the linear-work QRQW PRAM algorithms for problems in which the input of length n resides in global memory translate into algorithms with asymptotically optimal work on the QSM that run with a slowdown of g with respect to the corresponding QRQW PRAM algorithm.

The following lower bounds for the QSM are given in [1]. The CRCW PRAM lower bound result of Beame and Hastad [7] gives a lower bound for the n-element *parity*, *summation*, *list ranking* and *sorting* problems of $\Omega(g \cdot \lg n / \lg\lg n)$ time on the QSM for either deterministic or randomized algorithms when the number of processors is polynomial in n, the size of the input. Also given in that paper is a simple lower bound with a matching upper bound of $\Theta(ng)$ for the *one-to-all* problem in which one processor has n distinct values in its local memory of which the ith value needs to be read by processor i, $1 \leq i \leq n$.

A lower bound of $\Omega(g \lg n / \lg g)$ for broadcasting to n processors is given in [1]; in contrast to an earlier lower bound for this problem on the BSP given in [25] this lower bound holds even if processors can acquire knowledge through non-receipt of messages (i.e., by reading memory locations that were *not* updated by a recent write operation). We note that the same lower bound on time holds for the problem of broadcasting to n *memory locations* since any algorithm that broadcasts to n memory locations can broadcast to n processors in additional g units of time. Further, by Observation 5.3 $\Omega(ng)$ work is necessary since writes to n distinct global memory locations are required.

5.3 Some faster algorithms for the QSM

By pipelining reads and writes to memory from different processors to amortize against the delay due to the gap parameter g at processors, it is possible to obtain an algorithm for the QSM that runs faster than g times the running time for the fastest QRQW PRAM algorithm. As an example of an algorithm that is optimized for the QSM, consider the *leader election* problem in which the input is a Boolean n-array, and the output is the first location in the array with value 1, if such a location exists, and is zero otherwise. The fastest QRQW PRAM algorithm for this problem is just the 'binary tree' EREW PRAM method that halves the number of candidates in each of $\lg n$ rounds with $O(n)$ work (there is a faster algorithm on the CRQW PRAM, but that algorithm is not known to map onto the QSM with a slowdown of only g). This QRQW PRAM algorithm will map on to the QSM as a $O(g \lg n)$ time algorithm with $O(gn)$ work. However, we can optimize further for the QSM by replacing the normal 'binary tree' method by a 'g-ary tree'. This takes advantage of the fact that requests at the memory are processed every time step, while at the processors a request can be sent only every g steps. The time taken by this algorithm to solve the leader election problem on the QSM is $O(g \lg n / \lg g)$ while still performing $O(gn)$ work. If the input is distributed evenly among $n/(g \lg n / \lg g)$ processors, then the time is $O(g \lg n / \lg g)$ and the work is $O(n)$.

A similar strategy applies to the *broadcasting* problem in which the value at one location in memory needs to be transmitted to n processors or to n memory locations.

We now consider the problem of sorting on the QSM.

Among sorting algorithms that are fairly simple, the fastest $O(n \lg n)$ work algorithm on the EREW PRAM is an $O(\lg^2 n)$ time randomized quicksort algorithm (see, e.g. [23]), and on the QRQW PRAM, a randomized $\sqrt{n}$-sample sort algorithm that runs in $O(\lg^2 n/\lg\lg n)$ time, $O(n \lg n)$ work, and $O(n)$ space [17].

On the QSM, the randomized sample sort algorithm can be mapped onto the QSM to perform $O(n \lg n)$ work provided the computation is very coarse-grained, i.e., the number of processors p is polynomially small in n and $g = o(\lg n)$; this QSM algorithm is essentially the same as the BSP algorithm based on sample sort [14]. If we look for a highly parallel sorting algorithm that is fairly simple, an adaptation of the QRQW PRAM sample sort algorithm appears to be the fastest. A straightforward analysis of this algorithm on the QSM using Observation 5.2 results in an algorithm that runs in $O(g \cdot \lg^2 n/\lg\lg n)$ time while performing $O(g \cdot n \lg n)$ work. However, an analysis of the algorithm directly for the QSM shows that it runs in $O(\lg^2 n/\lg\lg n + g \lg n)$ time while performing $O(gn \lg n)$ work. Thus, if g is moderately large, specifically, $\Omega(\lg n/\lg\lg n)$, the sample sort algorithm will run within the same time and work bounds (randomized) as the more involved algorithms obtained by mapping the asymptotically optimal EREW PRAM algorithms onto the QSM. The improvement in running time for the QSM sample sort algorithm in comparison to the QRQW PRAM sample sort comes from the fact that the $\Theta(\lg^2 n/\lg\lg n)$ term in the time bound is only due to the bound on the contention at memory locations in a dart-throwing step. Since the QSM model charges only κ time for contention κ, this term is not multiplied by g in the time bound.

5.4 Mapping BSP algorithms onto the QSM

We now turn to the issue of mapping BSP algorithms onto the QSM. For this we assume that the input is distributed across the QSM processors to conform to the input distribution for the BSP algorithm; alternatively one can add the term ng/p to the time bound for the QSM algorithm to take into account the time needed to distribute the input located in global memory across the private memories of the QSM processors.

Many of the BSP algorithms reported in the literature can be mapped back on the QSM using the version of the algorithm that results when $L = 1$. For instance for the n-element summation, parity and prefix sums problems, the BSP algorithm that takes time $(gd+L)\lg_d n$, minimized by choosing $d \geq 2$ appropriately ($d = \lceil L/g \rceil$ if $L > g$ and $d = 2$ if $L \leq g$) maps on to the QSM as a simple $O(g \lg n)$ time algorithm that performs $O(ng)$ work. Similarly the BSP sorting algorithm of [14] and the matrix multiplication algorithms of [38, 33] map onto the QSM step by step with a performance corresponding to the case when $L = 1$ in the BSP algorithms.

The QSM algorithms in the above paragraph are obtained by the following simple strategy to map each step of the BSP algorithm on to the QSM to run in the time the step would take on the BSP if $L = 1$. A message sent by processor i to a memory location m of processor j on the BSP is written into shared memory location (j, m) by processor i in the QSM and then read by processor j. We will refer to a BSP algorithm as *well-behaved* if it can be mapped onto the QSM in the above manner.

Not all BSP algorithms are well-behaved as seen in the following example. The elements of an array $A[1..n]$ are distributed uniformly over p BSP processors. Each processor applies a certain function to its local inputs, and thereby generates some pairs (i, v), where v is the new value for $A[i]$. The new values generated have the property that each processor generates no more than c such values, and there are no more than c new updates generated for each block of inputs assigned to a processor, where $c = o(n/p)$; other than these two restrictions, the indices i of the locations in the array A whose values are changed are arbitrary. These new values are updated on the BSP by sending a c-relation in cg time units. Then in additional n/p time each BSP processor determines the new values of all of its local inputs by reading the corresponding local memory locations. This computation takes time $O(cg + n/p)$ on the BSP. If we implement this algorithm step-by-step on a QSM, the updated values will be written into a copy of the array $A[1..n]$ in shared memory, and each QSM processor then needs to read these updated values. Since it is not known ahead of time which values were updated, each QSM processor would need to read from global memory, the current value of each of the n/p elements of $A[i]$ that it has in local memory. This will take $\Theta(gn/p)$ time, which is larger than the running time on the BSP since $c = o(n/p)$.

While the above example indicates that the BSP is in some ways more powerful than the QSM, it is not clear that we want a general-purpose bridging model to incorporate these features. In general, there will be features such as these arising due to contrasts between message-passing and shared memory, between coherent and non-coherent caches, between update and invalidation-based coherence protocols, etc. Any choice of these features may not be representative of a wide range of parallel machines. Moreover, current designers of parallel processors often hide the memory partitioning information from the processors since this can be changed dynamically at run-time. As a result an algorithm that is designed using this additional power of the BSP over the QSM may not be that widely applicable. Fortunately, many of the BSP algorithms reported in the literature have simple communication patterns that map onto the QSM by the simple strategy described above. Also, we have a randomized strategy that can map any BSP algorithm onto the QSM in a work-preserving manner, provided a logarithmic slowdown is acceptable. The algorithm is described in [19]. We state here a lemma and a theorem that describe the performance of this algorithm.

Lemma 5.4 *Consider a step of an n-component BSP with gap g and latency L that involves routing an h-relation. On a QSM with gap parameter g this step can be emulated with high probability in n in a work-preserving manner with a slowdown of $O(1 + \lg n/(h + L/g))$.*

The probability that the emulation will fail to perform according to the stated bounds is less than $1/n^{\delta}$, for some $\delta > 0$, whose value depends on parameters of the algorithm. Thus, if a BSP algorithm takes no more than n^{ϵ} steps, for any $\epsilon, 0 < \epsilon < \delta$, then the probability that the emulation of any one of its steps on a QSM fails is polynomially small in n. This leads to the following theorem.

Theorem 5.5 *An algorithm that runs in time $t(n)$ on an n-component* BSP *with gap parameter g and periodicity factor L, where $t(n) \le 1/n^{\epsilon}$, for a suitable $\epsilon > 0$, can be emulated with high probability on a* QSM *with the same gap parameter g to run in time $O(t(n) \cdot \lceil g \lg n/L \rceil)$ with $n/\lceil g \lg n/L \rceil$ processors when $L \ge g$, and otherwise in time $O(t(n) \cdot \lg n)$ with $n/\lg n$ processors.*

5.5 On the queuing memory contention rule for the QSM

We note that a work-preserving emulation of a BSP with $g = 1$ is not known on the EREW PRAM if the slowdown is to be bounded by $polylog(n)$. If such an emulation is discovered, it will give rise to randomized linear work polylog time algorithms on the EREW PRAM for certain problems, such as computing a random permutation, for which such an algorithm is not known currently. Therefore, even though the EREW PRAM is often referred as stronger model than the BSP, its expressive power may actually be inferior, in some cases.

On the other hand, for the more powerful CRCW PRAM there appears to be a mismatch in the reverse direction since no work-preserving emulation of a CRCW PRAM on a BSP with $g = 1$ is known if the slowdown is to be bounded by $polylog(n)$. Thus, if either the EREW PRAM or the CRCW PRAM is augmented with the gap parameter, the resulting model is not known to have as strong a correspondence to the BSP as we have shown for the QSM. In other words, the queuing memory contention rule for the QSM, in contrast to the exclusive or concurrent rules, is crucial in order for it to serve as a bridging shared-memory model.

6 Conclusion

Developing effective models for parallel computation, at suitable levels of abstraction, remains a fundamental challenge in parallel processing. This paper has provided evidence that a shared-memory model, with its high level of abstraction, can nevertheless serve as a bridging model for parallel computation. Models at lower levels of abstraction are important in providing increased accuracy at the cost of increased complexity in the model.

We have described a new model, the Queuing Shared Memory (QSM) model, and discussed its possible advantages over previous shared-memory models, and over current bridging message-passing models. The model has a simple queuing metric for shared-memory access, and only two parameters– p, the number of processors and g, the bandwidth gap– yet it can be efficiently emulated on both the BSP and $(d, \mathbf{x})$-BSP models, using an arguably practical emulation. Thus the QSM can be effectively realized on machines that can effectively realize the BSP, as well as on machines that are better modeled by the $(d, \mathbf{x})$-BSP. We have presented evidence that both the queuing metric and the bandwidth parameter are essential to the QSM's effectiveness as a bridging model. In addition, we have described several algorithms for the QSM, as well as general strategies for mapping EREW PRAM, QRQW PRAM and BSP algorithms onto the QSM.

We conclude that a model such as the QSM can serve the role of a bridging model for parallel computation while preserving the high-level abstraction of a shared-memory model.

Future research should consider further algorithmic techniques that may be useful for this model, as well as experimental validation of the model. Such validation may reveal the primary importance of features not present in either the QSM, BSP or LogP. For example, each of these models defines a single bandwidth parameter that reflects a per-processor bandwidth limitation; other recent work has considered variants of these models with an aggregate bandwidth limitation [1] or a hierarchical bandwidth limitation that accounts for network proximity [30, 11, 12, 26, 41]. Per-processor bandwidth limitations better model machines in which each processor has access to its "share" of the network bandwidth and no more, as well as machines for which the primary network bottleneck, in the absence of hot-spots, is in the processor-network interface. As a second example, each of these models ignores the memory hierarchy at a processor, assuming a unit-time charge for local operations regardless of the local working set size. A possible feature to consider is to limit the size of the private memories on the QSM, or to have two levels of memory hierarchy on the BSP or LogP. Finally, as discussed in Section 3, each of these models disregards spatial locality. Variants of the BSP and LogP that account for spatial locality include [24, 3, 26, 5]. In machines supporting a single address space, the unit of data transfer between components is typically either a cache line or a page, and hence opportunities to exploit spatial locality are restricted to that level of granularity. A possible enhancement for the QSM would be to have the shared-memory partitioned into small, fixed-sized blocks of locations that could be accessed efficiently; the realization of such a QSM on a distributed-memory machine would map these blocks pseudo-randomly onto the memory banks.

Should it become necessary to include additional features as part of a bridging model, the QSM may be more suited for augmentation than the BSP or LogP, since it is simpler, with fewer parameters.

References

[1] M. Adler, P. B. Gibbons, Y. Matias, and V. Ramachandran. Modeling parallel bandwidth: Local vs. global restrictions. *These proceedings*.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, 1990.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Sheiman. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 95–105, July 1995.

[4] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 13–22, June-July 1992.

[5] A. Baumker and W. Dittrich. Fully dynamic search trees for an extension of the BSP model. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 233–242, June 1996.

[6] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. Technical report, University of Paderborn, 1996.

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

[8] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

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

[10] 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.

[11] P. de la Torre and C. P. Kruskal. Towards a single model of efficient computation in real parallel machines. *Future Generation Computer Systems*, 8:395–408, 1992.

[12] P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proc. Euro-Par'96*, pages 352–358, August 1996.

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

[14] A. V. Gerbessiotis and L. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[15] 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.

[16] P. B. Gibbons. What good are shared-memory models? In *Proc. 1996 ICPP Workshop on Challenges for Parallel Processing*, pages 103–114, August 1996. Invited position paper.

[17] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. *Journal of Computer and System Sciences*, 53(3):417–442, 1996. Special issue devoted to the best papers in the *1994 ACM Symp. on Parallel Algorithms and Architectures*.

[18] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write Asynchronous PRAM model. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing, Lecture Notes in Computer Science, Vol. 1124*, pages 279–292. Springer, Berlin, August 1996. Proc. 2nd International Euro-Par Conference, Lyon, France, Volume II.

[19] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? Technical report, Bell Laboratories, April 1997.

[20] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 1997. To appear. Preliminary version appears in *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 638-648, January 1994.

[21] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 16–25, June 1994.

[22] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

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

[24] J. JáJá and K. W. Ryu. The Block Distributed Memory model. Technical Report UMIACS-TR-94-5, University of Maryland Institute for Advanced Computer Studies, College Park, MD, January 1994.

[25] B H. H. Juurlink. Ph.D. Thesis, Leiden University, 1996.

[26] B. H. H. Juurlink and H. A. G. Wijshoff. The E-BSP Model: Incorporating general locality and unbalanced communication into the BSP Model. In *Proc. Euro-Par'96*, pages 339–347, August 1996.

[27] R. Karp, A. Sahay, E. Santos, and K.E. Schauser. Optimal broadcast and summation in the LogP model. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 142–153, June-July 1993.

[28] 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.

[29] K. Kennedy. A research agenda for high performance computing software. In *Developing a Computer Science Agenda for High-Performance Computing*, pages 106–109. ACM Press, 1994.

[30] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3(1):53–77, 1988.

[31] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. 28th Hawaii International Conf. on System Sciences*, pages II: 61–70, January 1995.

[32] Y. Mansour, N. Nisan, and U. Vishkin. Trade-offs between communication throughput and parallel time. In *Proc. 26th ACM Symp. on Theory of Computing*, pages 372–381, 1994.

[33] W. F. McColl. A BSP realization of Strassen's algorithm. Technical report, Oxford University Computing Laboratory, May 1995.

[34] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[35] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

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

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

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

[39] 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.

[40] U. Vishkin. A parallel-design distributed-implementation (PDDI) general purpose computer. *Theoretical Computer Science*, 32:157–172, 1984.

[41] H. A. G. Wijshoff and B. H. H. Juurlink. A quantitative comparison of parallel computation models. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 13–24, June 1996.