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

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

September 21, 1998

Abstract

There has been a great deal of interest recently in the development of general-purpose bridging models for parallel computation. Models such as the BSP and LogP have been proposed as more realistic alternatives to the widely-used PRAM model. The BSP and LogP models imply a rather different style for designing algorithms when compared to the PRAM model. Indeed, while many consider data parallelism as a convenient style, and the shared-memory abstraction as an easy-to-use platform, the bandwidth limitations of current machines have diverted much attention to message-passing and distributed-memory models (such as the BSP and LogP) that account more properly for these limitations.

In this paper we consider the question of whether a shared-memory model can 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. 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 more accurately models 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.

[*]A preliminary version of this paper appeared in *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 72–83, June 1997.

[†]Information Sciences Research Center, Bell Laboratories (Lucent Technologies), 600 Mountain Avenue, Murray Hill NJ 07974. email: gibbons@research.bell-labs.com

[‡]Information Sciences Research Center, Bell Laboratories (Lucent Technologies), 600 Mountain Avenue, Murray Hill NJ 07974. Current address is Tel-Aviv University, Ramat Aviv, Tel-Aviv 69978 Israel. Email: matias@math.tau.ac.il.

[§]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.

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. A number of models for parallel computation have been proposed and studied in the last twenty years. Primary among them are the parallel random access machine (PRAM) model [29, 48, 43, 65], in which processors execute in lock-step and communicate by reading and writing locations in a shared memory, and network-based models (hypercube, butterfly, arrays, etc. [51]), in which processors communicate by sending messages to their neighbors in the given network. The PRAM model, although simple and well-suited for developing parallel algorithms, is considered by many to be too high level, failing to accurately model parallel machines. Network-based models are considered by many to be too low level, failing to be broadly applicable, and not reflective of the current generation of parallel machines. Thus, a number of alternative, intermediate models have been proposed and studied in recent years. These abstract models differ in what aspects of parallel machines are exposed. Some focus on dealing with asynchrony in a shared-memory context (e.g., [8, 20, 21, 28, 32, 35, 49, 57, 61]). Others focus on accounting for the overheads in accessing the shared memory ([2, 3, 25, 32, 41, 44, 52, 56]) or in sending messages ([5, 9, 10, 22, 23, 39, 53, 55, 69]). Several models are primarily concerned with the memory hierarchy, especially disk I/O ([6, 62, 72]). Others focus on contention at the memory location ([28, 36]) or memory module ([60, 7, 27]). Finally, a few models incorporate powerful aggregate communication primitives ([14, 18]).

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 [71], Kennedy [50], Smith [67], and Blelloch [15] 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 has been faulted for completely failing to model 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., [43, 48, 65]). However, in the last few years, new alternatives such as the BSP [69] and LogP [22] 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 parameter, $g \geq 1$, that reflects the *gap* between the local instruction rate and the communication rate. 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, as characterized by a gap parameter, g, 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 [36]. 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 [69, 71] 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 [16], 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, where g is the bandwidth (gap) parameter of the QSM. 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. Some advantages of shared-memory models as bridging models are discussed in Section 2. In Section 3, we describe the Queuing Shared Memory model, and qualitatively compare it with previous models, and in particular, with the BSP. In Section 4, 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 5, we provide a few scenarios where the QSM is a more accurate model than the more detailed BSP and LogP. Section 6 presents algorithmic results and issues related to algorithm design on the QSM. Section 7 explores the merits of incorporating into the QSM model distinct bandwidth gaps at the processors and the memories.

Finally, we refer the reader to the position paper [33], 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 Advantages of shared-memory models as bridging models

A bridging model should provide an abstraction that is on the one hand easy to use by algorithm designers and programmers, and on the other hand can be realized by hardware and system software at a variety of price vs. performance points. In this section, we describe several contexts under which the shared-memory abstraction is an attractive choice for a bridging model in this regard.

We consider a (pure) shared memory model to be one in which the processors communicate by reading and writing locations in a shared memory that is equally accessible by all processors. The shared memory is viewed as a collection of independent cells: the contention encountered in accessing a shared memory cell is a function only of the number of processors also accessing the same cell. There is no visible partitioning of the memory, and no sources of contention due to such partitioning. The PRAM is a classic example of a shared memory model.

The shared memory abstraction refers to the interprocessor communication. As part of its local private state, a processor may have additional memory such as registers, buffers, cache, and local memory banks. A shared memory model may be asynchronous. It may also have explicit charges for communication, modeling various overheads in reading or writing a shared memory that is not local to, and may be physically quite remote from, the processor requesting the read or write. Thus it is a mistake to view "shared memory model" as a synonym for PRAM.

The shared-memory abstraction is arguably easier to use than a message-passing or distributed-memory abstraction, and in certain important contexts, can be realized by a wider range of machines. In what follows, we elaborate on three of the advantages of the shared-memory abstraction over the message-passing and distributed-memory abstractions.

Smooth transition from sequential to SMP to MPP. 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). Code can be debugged on a smaller, simpler and cheaper machine, before running it on a larger, more expensive machine; this will often significantly reduce the overall debugging time. In short, the shared-memory abstraction offers ease of use in designing algorithms and programs that span a variety of machine sizes, and it has also been realized by machines that span a variety of machine sizes. This contrasts with message-passing and explicit distributed-memory, which are not directly supported by any sequential machine or SMP.

Portability across memory architectures. The shared-memory abstraction is also attractive for developing algorithms that span a variety of memory architectures. Since the layout of memory is hidden in the model, the target machine can support the model in a variety of ways beyond that made visible in message-passing or distributed-memory machines. For example, the target machine may choose to dynamically map memory locations to processors as the computation proceeds, as in a cache-only memory architecture (COMA) [68]. In general, the target machine is free to implement a variety of cache and memory consistency protocols (e.g., [31]), since the model does not presuppose a particular memory layout. The shared-memory abstraction is more relevant to parallel machines, such as the Cray C90, Cray J90, SGI Power Challenge, and Tera MTA, that have many more memory banks than processors in order to compensate for the slow cycle times of memories. This point is addressed further later in the paper in Section 5.

Important platform for algorithmic ideas. Finally, it is evident that a simple model with a shared-memory abstraction provides a useful platform for studying fundamental algorithmic issues. Many algorithms for more complex models are adaptations of algorithms first developed for a simple shared-memory model. There are numerous examples, covering a wide range of problem domains, including sorting [18, 30, 44, 37], connected components [38, 42], computational geometry [66], FFT [22], and string matching [24]. Designing an algorithm directly for the more complex model is typically a more daunting task than first developing the algorithmic insights on a simple shared-memory model and only then adapting them to the more complex model. Note that for any algorithm designed for a high-level bridging model (whether shared-memory, message-passing or distributed-memory), it may be desirable to consider a more complex, lower-level model when making important performance-enhancing refinements. The shared-memory abstraction is desirable when such refinements are not necessary (i.e., whenever the algorithm performance is acceptable) since it is easier to use, and, as discussed above, it is still useful even if such refinements are necessary.

3 The Queuing Shared Memory model

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

Definition 3.1 *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 could complete any time during the phase, although they are not guaranteed to complete until the end of the phase. Thus, we do not allow a value read from shared-memory to be used during the phase since the value may not be available until the end of the phase. Also if we allow a shared-memory location to be both read and written in the same phase then the value read could be either the initial value or the updated value since we make no assumption about when a read or write completes within 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 3.2 *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 3.3 *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 let $m_{rw} = \max\{1, \max_i\{r_i, w_i\}\}$ 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)[2]. 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 4 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 4 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 [36]. Previous work on the QRQW PRAM [36, 34, 16] has been focused primarily on contention issues, unlike this paper, which is primarily concerned with bridging models and bandwidth issues.

[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 [16] show that at least for certain machines, taking the maximum is more accurate than taking their sum.

[2]This issue is explored further in Section 7.

Comparison of Models of Parallel Computation			
model	**synchrony**	**communication**	**parameters**
PRAM [29]	lock-step	shared memory	p
Module Parallel Computer (MPC) [60]	lock-step	distributed memory	p
LPRAM [3]	lock-step	shared memory	p, ℓ
Phase LPRAM [32]	bulk-synchrony	shared memory	p, ℓ, s
XPRAM [70]	bulk-synchrony	message-passing	p, g, L
Bulk-Synchronous Parallel (BSP) [69]	bulk-synchrony	message-passing	p, g, L
Postal model [10]	asynchronous	message-passing	p, ℓ
LogP model [22]	asynchronous	message-passing	p, g, ℓ, o
QRQW Asynchronous PRAM [35]	asynchronous	shared memory	p
QRQW PRAM [36]	bulk-synchrony	shared memory	p
Block Distributed Memory (BDM) [44]	bulk-synchrony	distributed memory	p, g, L, B
PRAM(m) model [56]	lock-step	shared memory	p, m
Interval model [55]	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.* The fourth column indicates the parameters of the model, where p is the number of processors, ℓ is the communication latency (i.e., the time to deliver a message point-to-point or to access the shared memory), s is the cost for a barrier synchronization among all the processors, L is a single parameter that accounts for the sum of ℓ and s, g is the bandwidth gap (i.e., the rate at which processors can perform local operations divided by the rate at which the processors can sustain interprocessor or processor-memory communication), 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.

3.1 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. A model is considered to be *shared memory* only if it meets the standards for a pure shared-memory abstraction outlined in Section 2, i.e., that the memory is viewed as a collection of independent cells that are equally accessible by all processors. If the processors communicate by reading and writing locations in a memory that is partitioned, the model is considered to be a *distributed memory* model. For example, the BDM model [44] is distributed memory since the contention encountered by a read request depends on the number of other requests to the same memory module. The *message-passing* models shown in this table deliver messages point-to-point: this abstraction hides the details of how the message is routed through the interprocessor communication network, and hence is similar to the distributed-memory abstraction.

The fourth column indicates the parameters in the model. The description of these parameters is given in the table caption. Some models, such as the LPRAM model, account separately for computation steps and communication steps. This can be viewed as having a separate latency parameter, as indicated in the table.

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. **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 advantages of a shared-memory abstraction were discussed in Section 2.

2. **Bulk-synchrony.** The QSM supports bulk-synchronous operation, in which processors operate asynchronously between barrier synchronizations. As in models such as the PHASE LPRAM [32], the algorithm dictates the points at which barriers occur. 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. The model does not allow for general asynchronous algorithms. Permitting general asynchrony can lead to algorithms that run faster on MIMD machines. However, any asynchronous model that reasonably reflects real machines is considerably more difficult to use.

3. **Few parameters.** For simplicity, it is desirable for bridging models to have only a few parameters. As evidenced by [22, 30, 47] 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. In the intended emulation of the model on MIMD machines, the latency of communication is hidden by having each physical processor emulate a number of QSM processors. Formally, we consider the emulation of higher-level models on lower-level models (such as the BSP), in order to make claims about the cost, or lack thereof, of ignoring certain parameters in the higher-level model. 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 [36], the EREW rule is too strict, and the CRCW rule ignores the large performance penalty of high contention steps. Indeed, for most existing machines, including the Cray T3E, Cray C90, Cray J90, IBM SP2, Intel Paragon, MasPar MP-2 (global router), Tera MTA, and Thinking Machines CM-5 (data network), 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. This key feature will be discussed in Section 3.2.

6. **Work-preserving emulation of BSP.** In addition to the work-preserving emulation of QSM on BSP we observe 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 slow-down. This emulation holds for all BSP algorithms. This is discussed in more detail in Section 3.2 and in Section 6.

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. First, having only $m < p$ shared-memory locations is a large burden on the algorithm designer; no machines provide this restriction. Second, the model assumes that input is in a read-only memory that can be accessed by all processors without any bandwidth limitations; this undercharges the cost of such accesses for existing machines. Third, the model provides unlimited contention to the m shared-memory locations at no extra charge; this too is unrealistic for existing machines. Due to these features, the model does not seem to have an efficient emulation on lower-level models such as the BSP. The model is intended for lower bounds, and indeed lower bounds proved for the PRAM(m) model imply lower bounds for a large number of other models.

Mapping parameters to machines. There have been several papers reporting values for various model parameters on existing parallel machines. For example, Martin *et al* [58] reported values for the g, ℓ, and o parameters from the LogP model on three platforms: the Berkeley NOW cluster, the Intel Paragon and the Meiko CS-2. On the Berkeley NOW cluster, $g = 5.8$ microseconds (μs), $\ell = 5.0$ μs, and $o = 2.9$ μs. On the Intel Paragon, $g = 7.6$ μs, $\ell = 6.5$ μs, and $o = 1.8$ μs. On the Meiko CS-2, $g = 13.6$ μs, $\ell = 7.5$ μs, and $o = 1.7$ μs. Since the local instruction rate at a processor is tens of nanoseconds per instruction or faster, the normalized values for these parameters are in the hundreds to a few thousand. In contrast, Blelloch *et al* [17] considered two shared-memory vector multiprocessors, reporting (normalized) gap parameter values of $g = 1.2$ for the Cray C90 and $g = 1.8$ for the Cray J90.

3.2 Comparison to BSP

In this section we compare the QSM and the BSP in terms of their effectiveness as a bridging model for parallel computation. We choose to compare the QSM with the BSP rather than the LogP model since the QSM is a bulk-synchronous model like the BSP (and unlike the LogP) model.

The Bulk-Synchronous Parallel (BSP) model [69, 70] 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

Emulations on BSP			
model emulated	**on model**	**with slackness**	**work-preserving?**
EREW PRAM	BSP(g, L)	$\geq \max(\lg p, L/g)$	inefficient by a factor of g
QRQW PRAM	BSP(g, L)	$\geq \max(\lg p, L/g)$	inefficient by a factor of g
CRCW PRAM	BSP(g, L)	$\geq \max(p^{1+\epsilon}, L/g)$	inefficient by a factor of g
QSM(g)	BSP(g, L)	$\geq \max(g \lg p, L/g)$	yes

Table 2: *Some emulations of higher-level models on the* BSP *model.* The result for QSM is new. The emulations are randomized and the bounds are obtained with high probability in p.

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 $T = \max(w,\ g \cdot h,\ L)$.[3] Although the BSP is a message-passing model, it can also be viewed as a distributed-memory model where each memory component serves as a memory bank.

To compare the cost metrics of the BSP and the QSM, we consider the distributed-memory view of the BSP and a superstep comprised of local work, read requests and write requests. We can equate the two g parameters, and w_i with c_i (and hence w with m_{op}). Let $h_s = \max_{i=1}^{p} s_i$, the maximum number of read/write requests by any one processor, and let $h_r = \max_{i=1}^{p} r_i$, the maximum number of read/write requests to any one memory *bank*. The BSP charges the maximum of w, $g \cdot h_s$, $g \cdot h_r$, and L. The QSM, on the other hand, charges the maximum of w, $g \cdot h_s$, and κ, where $\kappa \in [1..h_r]$ is the maximum number of read/write requests to any one memory *location* and is not multiplied by g.

One important measure of a bridging model is its ability to be emulated by important lower-level models. Table 2 presents some known emulation results of higher-level models on the BSP. The *parallel slackness* in an emulation is the number of processors in the higher-level model per processor in the BSP model. An emulation is *work-preserving* if both models perform the same amount of work, to within constant factors. The first three rows show emulation results on the BSP of the EREW PRAM [69], the QRQW PRAM [36] and the CRCW PRAM [69]; note that none of these three models have a work-preserving emulation on the BSP if g is not a constant. In the case of the CRCW PRAM, even for a BSP with gap parameter that is a constant, a work-preserving emulation on the BSP is known only with a parallel slackness that is very large, i.e., polynomial in p. In contrast, the QSM does have a work-preserving emulation on a BSP with the same gap parameter, for any g, using only modest slackness and small constants. This result will be shown in the next section.

The emulation result implies that any algorithm designed on the QSM can be mapped onto the BSP in a work-preserving manner with only a modest slowdown. Since the QSM has fewer parameters than the BSP, and does not deal with memory partitioning details, for most problems it should be easier to design algorithms on the QSM than on the BSP. Moreover, the emulation result implies that any machine that can realize the BSP model can also realize the QSM model, given the additional system software needed for the (simple) emulation algorithm.

Many algorithms designed for the BSP have as their goal to minimize the number of supersteps

[3]Alternatively, the time is $w + g \cdot h + L$; this affects the bounds by at most a factor of 3, and the results in [16] show that at least for certain machines, taking the maximum of the three terms is more accurate than taking their sum.

(e.g., [37]). In contrast, the QSM does not account for the number of supersteps (e.g., there is no L parameter in the QSM model). Ignoring the number of supersteps simplifies the model, and it can be somewhat formally justified by the emulation result, which shows that any two QSM algorithms with the same QSM time bound will have the same BSP time bound when emulated on the BSP, regardless of the number of supersteps in the respective algorithms.

One can also consider the mapping of BSP algorithms onto the QSM. Many of the BSP algorithms reported in the literature have a simple version on the QSM corresponding to the case when the latency $L = 1$. As shown in Section 6 it is possible, in principle, to have BSP algorithms that do not map back to the QSM in a work- and time-preserving manner. Such algorithms would exploit the fact that a BSP processor

(i) could receive a piece of information that it did not specifically request, or

(ii) could access, as a unit-time local computation, a value (not requested by it) that was written into its local memory bank by another processor in an earlier step.

These features are appropriate in contexts where a processor can send a message directly to a processor at any time, or can write remotely into a processor's local portion of the shared memory. On the QSM a processor would need to initiate a read for any piece of information that it receives; further that access will be charged a cost of g at the time the processor reads it in addition to a cost of g being applied at the time the value was written into the shared-memory location.

While the features listed above could indicate that the BSP is in some ways more powerful than the QSM, it may not be desirable for 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, as discussed in Section 2, current designers of parallel processors often hide the memory partitioning information from the processors since this can be changed dynamically at runtime. As a result an algorithm that is designed, say, using this additional power of the BSP over the QSM may not be that widely applicable.

In Section 6, we show that a BSP that does not exploit features (i) and (ii) can be emulated on a QSM using a simple, deterministic, time- and work-preserving algorithm. We also show that any n-component BSP, even one that exploits these features, has a work-preserving emulation on a QSM with the same gap parameter, with a modest slowdown of $O(\lg n/(1 + L/g))$, with high probability in n; this emulation uses a fairly involved algorithm.

Thus, overall, a case can be made that the QSM is effective in modeling the essential features of the BSP while remaining at a higher level of abstraction.

4 Emulations of QSM on BSP models

The $(d, \mathbf{x})$-BSP [16] 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 [16], it is argued that for shared-memory machines with a high-bandwidth communication network and more memory banks than processors, the $(d, \mathbf{x})$-BSP is a more accurate model than the BSP. 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.

4.1 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).

The mapping of the QSM shared memory among the machine's memory banks assumes the machine supports a single address space. Many recent machines (e.g., Cray T3E) provide hardware support for a single address space; for other machines (e.g., IBM SP-2), it can be emulated in software with some overhead.

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 [16] 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 [16], 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 [44], LogGP [5], and BSP* [12, 11], 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 [22].

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. (In fact, this emulation is a fundamental component in the design of the Tera MTA.)

4.2 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 4.1 (work-preserving QSM emulation) *Consider a p-processor $(d, \mathbf{x})$-BSP with gap pa-*

rameter 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

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

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 [36, 16], 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.

Before we present the proof of this theorem, we note that in the original BSP, $d_g = x = 1$, so from the above theorem we obtain the following corollary:

Corollary 4.2 (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 4.1 We now prove the theorem. The proof is similar to that in [16], 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 $\mathbf{x} \geq 2d_g$. Thus,

$$\delta \geq d \frac{\lg p}{\lg(\mathbf{x}/d_g)} \quad . \tag{1}$$

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, $t_i \geq g$, and the maximum number of local operations at a processor in this step is t_i. The local computation of the

QSM processors can be performed on the $(d, \mathbf{x})$-BSP in time $(p'/p)t_i$, since each $(d, \mathbf{x})$-BSP processor emulates p'/p QSM processors.

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 the following theorem by Raghavan and Spencer, which provides a tail inequality for the weighted sum of Bernoulli trials:

Theorem 4.3 ([63]) *Let $a_1, \ldots, a_m$ be reals in $(0, 1]$. Let $x_1, \ldots, x_m$ be independent Bernoulli trials with $\mathbf{E}\,(x_j) = \rho_j$. Let $\Psi_\beta = \sum_{j=1}^{m} a_j x_j$. If $\mathbf{E}\,(\Psi_\beta) > 0$, then for any $\nu > 0$*

$$\mathbf{Prob}\,(\Psi_\beta > (1+\nu)\mathbf{E}\,(\Psi_\beta)) < \left(\frac{e^\nu}{(1+\nu)^{(1+\nu)}}\right)^{\mathbf{E}(\Psi_\beta)}\,. \tag{2}$$

We apply Theorem 4.3 with $\rho_j = 1/B$, and set

$$\nu = \alpha \frac{\mathbf{x}}{d_g} - 1\ ,$$

implying

$$(1+\nu)\mathbf{E}\,(\Psi_\beta) = \alpha \frac{\mathbf{x}}{d_g} \cdot \frac{p'}{\mathbf{x}pg} = \frac{\alpha p'}{dp}\ . \tag{3}$$

Therefore,

$$\begin{aligned}\mathbf{Prob}\left(\Psi_\beta > \frac{\alpha p'}{dp}\right) &\overset{[(2),(3)]}{<} \left(\frac{e}{(1+\nu)}\right)^{(1+\nu)\mathbf{E}(\Psi_\beta)} \overset{[(3)]}{=} \left(\frac{\alpha \mathbf{x}}{e d_g}\right)^{-\frac{\alpha p'}{dp}} \\ &\overset{[\alpha \geq e]}{\leq} \left(\frac{\mathbf{x}}{d_g}\right)^{\frac{-\alpha p'}{dp}} \overset{[\mathbf{x} > d_g]}{\leq} \left(\frac{\mathbf{x}}{d_g}\right)^{\frac{-\alpha}{d}\max(\delta, L/g)}\end{aligned}$$

$$\overset{[\mathbf{x}>d_g]}{\leq} \left(\frac{\mathbf{x}}{d_g}\right)^{\frac{-\alpha}{d}\delta} \overset{[(1)]}{\leq} \left(\frac{\mathbf{x}}{d_g}\right)^{-\alpha\frac{\lg p}{\lg(\mathbf{x}/d_g)}} = p^{-\alpha} \leq p^{-(c+\bar{c}+1)}$$

$$= \frac{p^{-(c+1)}}{p^{\bar{c}}} \overset{[\mathbf{x}\leq p^{\bar{c}}]}{\leq} \frac{p^{-(c+1)}}{\mathbf{x}} .$$

Note that

$$h_r^\beta = \sum_{j=1}^{m} x_j k_j = \Psi_\beta \cdot t_i .$$

Therefore,

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

Let $h_r = \max_\beta h_r^\beta$. Then

$$\mathbf{Prob}\left(h_r > \frac{\alpha p' t_i}{dp}\right) \leq B \cdot \mathbf{Prob}\left(h_r^\beta > \frac{\alpha p' t_i}{dp}\right) < B \cdot \frac{p^{-(c+1)}}{\mathbf{x}} = p^{-c} .$$

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

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

This completes the proof of Theorem 4.1. ■

4.3 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 4.4 (QSM on unbalanced (d,x)-BSP) *Consider a p-processor $(d, \mathbf{x})$-*BSP *with gap parameter g and periodicity factor L, such that $1 \leq \mathbf{x} < \min\{d_g, p^{\bar{c}}\}$, 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. As in the proof of Theorem 4.1, 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 4.1: 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 (Theorem 4.3), but with $\nu = \alpha - 1$. This yields

$$\begin{aligned}\textbf{Prob}\left(\Psi_\beta > \frac{\alpha p'}{\mathbf{x}p}\right) &< \left(\frac{\alpha}{e}\right)^{-\frac{\alpha p'}{\mathbf{x}pg}} \overset{[\alpha \geq 2e]}{\leq} 2^{-\frac{\alpha}{\mathbf{x}g}\max(\mathbf{x}g\lg p, d, L/g)} \\ &\leq p^{-\alpha} \leq p^{-(c+\bar{c}+1)} \overset{[\mathbf{x} \leq p^{\bar{c}}]}{<} \frac{p^{-(c+1)}}{\mathbf{x}} .\end{aligned}$$

It follows as in the previous proof that

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

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

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

The theorem follows. ∎

4.4 A lower bound

The following lower bound shows that the work bound in Theorem 4.4 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 4.5 *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. ∎

5 Improved accuracy through the QSM abstraction

The shared-memory abstraction of the QSM hides the details of the partitioning of memory into memory modules/components on existing machines. This partitioning is explicit in message-passing or distributed-memory models such as the BSP or LogP. Thus the QSM provides a higher-level of abstraction, while the BSP and LogP seemingly provide more accurate modeling of memory module contention.

In this section, we draw attention to machines for which the BSP and LogP models fail to accurately model memory module contention, whereas the QSM can lead to a more accurate accounting. For the former, we will refer to results in Blelloch *et al.* [16], whereas for the latter we will leverage Theorem 4.1 and experimental results in [16]. We also present a simple illustrative example.

5.1 The problem of memory layout mismatch

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 computer manufacturers, motivated by the increasing divergence between processor speeds and memory speeds, have designed 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. For these machines, the $(d, \mathbf{x})$-BSP [16] (described in Section 4) is a more accurate model than the BSP or LogP since it explicitly accounts for both (i) the *bank delay*, d, which is the bandwidth gap parameter at a memory bank, and (ii) the *bank expansion*, $\mathbf{x}$, which is the ratio of memory banks to processors.

Blelloch *et al.* showed experimentally that the $(d, \mathbf{x})$-BSP models the Cray C90 and Cray J90 quite accurately, even though the model ignores many details about these machines. They also showed 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 the LogP provide a poor prediction of an algorithm's performance, while the $(d, \mathbf{x})$-BSP provides 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 [38]. 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, BSP, and LogP. The contention is given on a logarithmic scale indicating the ratio between the maximum contention, k, and the total number of requests, $p \cdot S$ (p is the number of processors and S is the number of requests sent by each processor).

5.2 Suitability of QSM to Cray-like machines

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, we argue that 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, Theorem 4.1 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 (via hashing) to a memory layout of any granularity. Thus

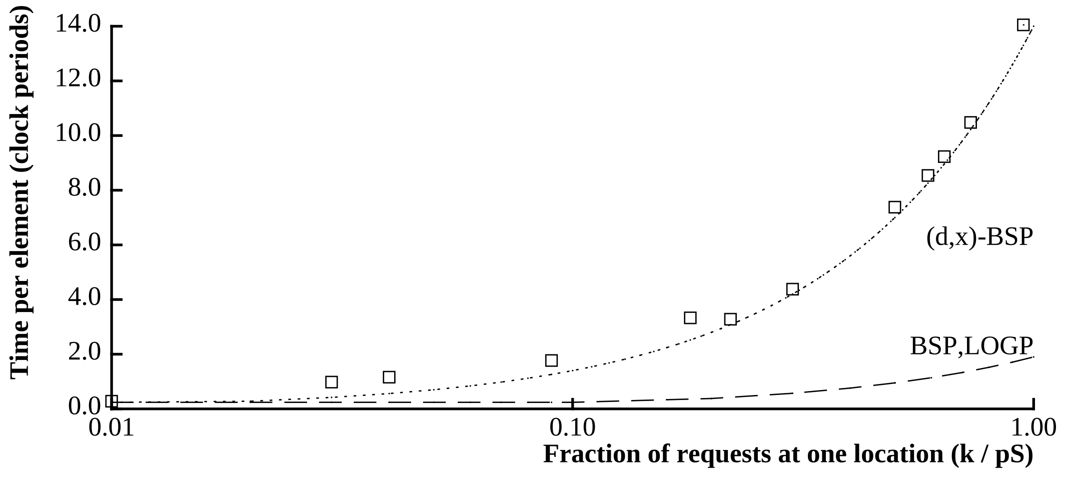

Figure 1: *Inaccuracies in the* BSP *and the* LogP *predictions, due to assuming the wrong memory layout and underestimating the cost of memory bank contention.* This figure is from [16].

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.

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 $\geq \mathbf{x} \geq d/g$. (When the slackness is $\geq \mathbf{x} \geq d/g$, then the p-processor distributed-memory model is emulated on a $(d, \mathbf{x})$-BSP with at most p memory banks.)

As a simple example illustrating the above discussion, consider the following two memory access patterns, A and B, occurring in an algorithm designed for the BSP. Suppose k processors send one message each to a BSP component C, for some 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. The cost on the BSP of each access pattern is the same, namely, $g \cdot k$, as in each case the requests are aggregated. Now suppose that the algorithm is run on a machine well-modeled by the $(d, \mathbf{x})$-BSP, with $\mathbf{x} \geq k$. On the $(d, \mathbf{x})$-BSP, the requests in A are always mapped to the same memory bank, but the requests in B could be mapped to different banks, depending on the mapping of $(d, \mathbf{x})$-BSP banks to BSP components. This results in a cost on the $(d, \mathbf{x})$-BSP of $\max(g, d \cdot k)$ for A but a cost of anywhere from $\max(g, d \cdot k)$ to $\max(g, d)$ for B, a potentially large distinction.

An algorithm designed for the QSM would distinguish between k requests to the same location versus k requests to different locations, charging $\max(g, k)$ for A and g for B. Moreover, Theorem 4.1 implies that using a random mapping of QSM memory locations to the $(d, \mathbf{x})$-BSP memory banks guarantees that, with high probability, there are no surprises in terms of memory bank contention when the algorithm is run on a machine well-modeled by the $(d, \mathbf{x})$-BSP. In any such mapping, the requests in A will be mapped to the same $(d, \mathbf{x})$-BSP memory bank, and hence are rightfully aggregated, whereas the requests in B will likely be mapped to different memory banks, and hence are rightfully *not* aggregated. Thus while the metric of the BSP may not be consistent with that of the $(d, \mathbf{x})$-BSP, the QSM maintains close consistency with the $(d, \mathbf{x})$-BSP.

6 Algorithmic issues

As a shared-memory model, the QSM offers a simple high-level medium for the design of parallel algorithms that can take into consideration effective use of limited bandwidth. In this section we present some algorithmic results and techniques for the QSM as well as general strategies to map algorithms developed on some other models onto the QSM.

In general the QSM is to be used for direct algorithm design that makes effective use of limited bandwidth. However, since we would like to leverage on the extensive literature on PRAM algorithms, in Section 6.1 we discuss the mapping of QRQW PRAM and EREW PRAM algorithms onto the QSM. In Section 6.2 we present some lower bounds, and in Section 6.3 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 6.4 and Section 6.5. In view of a simple lower bound of $\Omega(n \cdot g)$ that we prove in Section 6.2 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 6.4 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. In that section we also argue that BSP algorithms that are not 'well-behaved' use certain features of the BSP that are not quite representative of a large class of parallel machines. For completeness on the issue of expressive power, in Section 6.5 we show 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 6.6 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 6.1 *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 6.1, 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.

6.1 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 6.2 *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 [36, 34] 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 6.2 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.

There are two other avenues through which we can hope to obtain useful results for the QSM over those obtained through the mapping of QRQW PRAM algorithms. First, we can consider tailoring QSM algorithms to its cost metric for the gap parameter, thereby obtaining an improved running time for the algorithm. Second, we can relax the requirement that the input be placed in global memory, and allow the input to be distributed across the local memories of the processors in a suitable way. This would conform to the initial state for BSP algorithms, and in fact most BSP algorithms map back to the QSM in a natural way in this case.

We address each of these in turn in Section 6.3 and Section 6.4, respectively. But first, in the next section we mention some lower bounds for the QSM model.

6.2 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 6.3 *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 6.2 and Observation 6.3, 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 [13] 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 [45] 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 6.3 $\Omega(ng)$ work is necessary since writes to n distinct global memory locations are required.

6.3 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. Again, the QRQW PRAM algorithm of choice for this problem is a 'binary tree' broadcasting method that takes $O(\lg n)$ time with $O(n \lg n)$ work. This algorithm will map on to the QSM as a $O(g \lg n)$ time algorithm with $O(gn \lg n)$ work. By optimizing along the lines of the algorithm for leader election, we can derive an algorithm to broadcast to n processors on the QSM that runs in $O(g \lg n / \lg g)$ while performing $O((gn \lg n)/ \lg g)$ work. By the lower bound cited in Section 6.2, this result is optimal.

We can solve the related problem of *broadcasting to n memory locations* in the above time bound of $O(g \lg n / \lg g)$ but with $O(ng)$ work. For this, we use $p = n \lg g / \lg n$ processors and broadcast to the p processors in time $O(g \lg n / \lg g)$. We then spend an additional $O(g \lg n / \lg g)$ time to have each processor write into $\lg n / \lg g$ locations. As noted in Section 6.2 we have a matching lower bound on both the running time and the work.

We now consider the problem of sorting on the QSM. The problem of designing highly parallel algorithms for sorting n keys from a totally ordered set is a well-studied one. On the EREW PRAM, there are two known $O(\lg n)$ time, $O(n \lg n)$ work algorithms for general sorting [4, 19]; these deterministic algorithms match the asymptotic lower bounds for general sorting on the EREW and CREW PRAM models. Both of these algorithms map onto the QSM to run in $O(g \lg n)$ time and $O(gn \lg n)$ work using Observation 6.2. Unfortunately, these two algorithms are not as simple and practical as one would like. Goodrich [37] gives an algorithm for the BSP based on [19] that performs work $O((L + gn) \lg n / \lg(n/p) + n \lg n)$ with p processors. Since this algorithm is an adaptation of [19] it is again a fairly complicated algorithm.

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., [43]), 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 [34].

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 [30]. 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 6.2 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.

6.4 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 [30] and the matrix multiplication algorithms of [69, 59] 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.

The mapping onto the QSM needed for a well-behaved BSP algorithm may not be possible if, in the BSP algorithm, a BSP processor

(i) could receive a piece of information that it did not specifically request, and its future behavior depends on whether or not it receives this piece of information; or

(ii) could access, as a unit-time local computation, a value (not requested by it) that was written into its local memory bank by another processor in an earlier step.

On the QSM a processor would need to initiate a read for any piece of information that it receives; further that access will be charged a cost of g at the time the processor reads it in addition to a cost of g being applied at the time the value was written into the shared memory location.

We now give an example of a BSP computation that is not well-behaved. 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 may not be desirable for a general-purpose bridging model to incorporate these features of the BSP, as argued in Section 3.2.

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, as shown in the next subsection, there is a randomized strategy that can map any BSP algorithm onto the QSM in a work-preserving manner, provided a logarithmic slowdown is acceptable.

6.5 A work-preserving emulation of BSP on QSM

In this section we describe a randomized work-preserving emulation of an n-component BSP on a QSM with $O(\lg n)$ slowdown that works with high probability in n (i.e., the probability of failure is $1/n^{\delta}$, for some $\delta > 0$). For this emulation we assume that the input is distributed across the local memories of the QSM processors in the same manner as in the BSP algorithm.

In the emulation we use the shared memory of the QSM only for the purpose of realizing the h-relation performed by the BSP in each step, and each QSM processor copies into its private memory any message that was sent to the local memory of the corresponding BSP processor in that step. The algorithm is reminiscent of a randomized CRQW PRAM algorithm for integer sorting given in [34]. It proceeds by using the shared memory to sort the messages being sent in the current step according to their destination. Each processor then reads the messages being sent to it from an appropriate subarray in the shared memory and writes it into the corresponding location in its local memory. The details of the emulation algorithm are given below.

1. Compute the total number of messages, M, to be sent by all processors as follows: Construct an array $A[1..n]$ in shared memory, with $A[i]$ containing the number of messages being sent by processor i, and compute M as the sum of the elements in this array. This step can be performed deterministically in $O(g \lg n)$ time and $O(M + g \cdot n)$ work (note that $M \leq n \cdot h$, where h is the maximum number of messages sent or received by any processor in this BSP step).

If $M \geq n/\lg n$ then execute steps 2 through 9 below.

2. Construct a sample S of the messages to be sent by choosing each message independently with probability $1/\lg^3 M$. The size of the sample will be $O(M/\lg^3 M)$ w.h.p.
3. Sort the sample deterministically according to destination using a standard sorting algorithm, e.g., Cole's merge-sort; this takes $O(g \lg M)$ time and $O(g \cdot M/\lg^2 M)$ work.
4. Group the destinations into groups of size $\lg^3 M$ and determine the number of messages destined for each group. This can be computed by a prefix sums computation that takes $O(g \lg M)$ time and $O(gM)$ work.
5. Let k_i be the number of elements in the sample destined for the ith group. Obtain a high probability bound on the total number of messages to each group as $r_i = O(\max(k_i, 1) \cdot \lg^3 M)$. Make $\lg^3 M$ copies of each r_i, and place the duplicate values of the r_i in an array $R[1..n]$ such that $R[i]$ contains the bound for the group that contains destination $i, 1 \leq i \leq n$. This step can be performed in $O(g(1 + \lg\lg M/\lg g))$ time and $O(ng)$ work using a broadcasting algorithm for each r_i.
6. In parallel, for each i, all processors with a message to a destination i read the value of this bound from $R[i]$; this takes time $\leq gh$ and $O(g \cdot M)$ work.
7. Use an algorithm for multiple compaction to get the messages in each group into a linear-sized array for that group; this takes $O(g \lg M)$ time and $O(g \cdot M)$ work by the adaptation of the

randomized QRQW PRAM algorithm for multiple compaction given in [34] to the QSM using Observation 6.2.

8. Perform a stable sort within each group according to the individual destination; this can be performed in $O(g \lg M)$ time and $O(gM)$ work deterministically using an EREW PRAM radix-sort algorithm within each group.

9. Move the messages into an output array R of size M sorted according to destination in $O(gh)$ time and $O(M)$ work. Create an array B of size n that contains the number of messages to each destination, and the starting point in the output array for messages to that destination; this can be done by computing prefix sums on an appropriate M-array and takes $O(g \lg M)$ time and $O(g \cdot M)$ work. Processor i reads this value from $B[i]$ and then reads the messages destined for it from the output array in time $O(gh)$ and work $O(g \cdot M)$.

If $M < n/\lg n$ then we sort the messages deterministically according to their destination; this takes time $O(g \lg n)$ and $O(gn)$ work. We then perform step 9 above.

Since $M \leq n \cdot h$, the above QSM algorithm runs in $O(g(h + \lg n))$ time while performing $O(ghn)$ work. High-probability bounds for the randomized steps in the above algorithm are shown in [34]. Since a BSP routes an h-relation in $O(gh + L)$ time while performing $O(n(gh + L))$ work, this is a work-preserving emulation of a BSP h-relation, with a slowdown of $O(1 + \lg n/(h + L/g))$.

In summary we have the following result.

Lemma 6.4 *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 such as the constants in the sizes of arrays used in steps 5 and 7. 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 6.5 *An algorithm that runs in time $t(n)$ on an n-component BSP with gap parameter g and periodicity factor L, where $t(n) \leq c \cdot n^{\gamma}$, for some constants $c, \gamma > 0$, 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 $L \geq g$, and otherwise in time $O(t(n) \cdot \lg n)$ with $n/\lg n$ processors.*

6.6 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.

7 Gap parameter at memory

The QSM has a gap parameter g at the processors, but no gap parameter at the memory – each request at memory is serviced in unit time once it reaches the head of its queue. One could argue that another gap parameter d for processing memory accesses would be a desirable feature in a general-purpose model, since many currently available parallel machines have different gap parameters at processors and at memory banks. We refer to this model as QSM(g, d). The following result is shown in [64].

Observation 7.1 *[64] There is a deterministic work-preserving emulation of* QSM(g, d') *on* QSM(g, d) *with slowdown* $O(\lceil \frac{d}{d'} \rceil)$.

The above observation shows that very little generality is lost in assuming that the gap parameter at memory is 1 rather than some other value d. The only potential drawback is that an algorithm designed for the QSM$(g, 1)$ (which is the standard QSM model) may not achieve the full level of speed-up attainable on QSM(g, d), due to the slowdown in the emulation mentioned in the observation. The advantage in not having a gap parameter d at memory is that we have a simpler model with fewer parameters. We believe that the simplicity achieved in not having a gap parameter d at memory far outweighs the drawback of not achieving the best possible speed-up for a specific value of d.

We define the s-QSM (the *symmetric* QSM) to be the model QSM(g, g). This is the special case of QSM(g, d) with the same gap parameter g at both processors and memory. This model has the same number of parameters as the QSM, and could serve as an alternative to the QSM. The main difference between the two models is the asymmetry in the application of the gap parameter at processor and memory in the case of the QSM versus the symmetry in this application in the s-QSM. As a result, the fastest speed-up achievable for a given problem can be slightly different in the two models, e.g., on the s-QSM broadcasting a bit to n memory locations has the tight time bound of $\Theta(g \lg n)$ in contrast to the tight bound of $\Theta(g \lg n / \lg g)$ for the QSM. (Several other lower bounds for QSM and s-QSM are given in [54].) However, except for this difference, the QSM and the s-QSM are essentially interchangeable models. Specifically, the QSM can emulate the s-QSM with no slowdown and, as follows from Observation 7.1, there is a work-preserving emulation of the QSM on the s-QSM with slowdown $O(g)$.

8 Conclusion

Developing effective models for parallel computation, at suitable levels of abstraction, remains a fundamental challenge in parallel processing. The BSP and LogP models have gained considerable popularity as high level "bridging models" for parallel computation, and indeed they have many attractive features and have proven to be effective in many scenarios. We have described a new model, the Queuing Shared Memory (QSM) model, which in many cases may be an attractive alternative as a bridging model for parallel computation. In contrast to the BSP and LogP models, the QSM model provides a shared-memory abstraction. 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. On the other hand, as discussed in this paper, there are trade-offs in any bridging model, and scenarios in which another model (BSP, LogP, etc.) may be preferred. Thus the choice of a best bridging model remains open to debate.

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 [52, 25, 26, 46, 73]. 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. Third, as discussed in Section 4, each of these models disregards spatial locality. Variants of the BSP and LogP that account for spatial locality include [44, 5, 46, 11]. 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. Finally, each of these models ignores the effects of the cache coherence protocol used in most shared-memory multiprocessors to maintain consistency among the various cached copies of shared-memory data. It would be interesting to study a QSM model that incorporates and accounts for a standard invalidation-based cache coherence protocol [40]. 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. In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 94–105, June 1997.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. In *Proc. 1st ACM Symp. on Parallel Algorithms and Architectures*, pages 11–21, June 1989.

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

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

[5] 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.

[6] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 600–608, October 1990.

[7] R. J. Anderson and G. L. Miller. Optical communication for pointer based algorithms. Technical Report CRI 88-14, Computer Science Department, University of Southern California, Los Angeles, CA, 1988.

[8] 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.

[9] A. Bar-Noy, J. Bruck, C. T. Ho, S. Kipnis, and B. Schieber. Computing global combine operations in the multi-port postal model. In *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, pages 336–343, December 1993.

[10] 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.

[11] 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.

[12] 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.

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

[14] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, Cambridge, MA, 1990.

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

[16] G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):943–958, 1997. Preliminary version appears in *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 84–94, July 1995.

[17] G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):943–958, 1997.

[18] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

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

[20] 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.

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

[22] 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.

[23] R. Cypher and S. Konstantinidou. Bounds on the efficiency of message-passing protocols for parallel computers. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 173–181, June-July 1993.

[24] A. Czumaj, Z. Galil, L. Gąsieniec, K. Park, and W. Plandowski. Work-time-optimal parallel algorithms for string problems. In *Proc. 27th ACM Symp. on the Theory of Computing*, pages 713–722, May-June 1995.

[25] 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.

[26] 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.

[27] M. Dietzfelbinger and F. Meyer auf der Heide. Simple, efficient shared memory simulations. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 110–119, June-July 1993.

[28] 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.

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

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

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

[32] 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.

[33] 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.

[34] 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 selected papers from the *1994 ACM Symp. on Parallel Algorithms and Architectures.*

[35] 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.

[36] 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.

[37] M. Goodrich. Communication-efficient parallel sorting. In *Proc. 28th ACM Symp. on the Theory of Computing*, pages 247–256, May 1996.

[38] 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.

[39] S. Hambrusch and A. Khokhar. C^3: An architecture-independent model for coarse-grained parallel machines. In *Proc. 6th IEEE Symp. on Parallel and Distributed Processing*, pages 544–551, 1994.

[40] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, 1996. Second edition.

[41] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation: I. The model. *Journal of Parallel and Distributed Computing*, 16:212–232, 1992.

[42] T.-s. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components in graphs. In *Proc. AMS/DIMACS Parallel Implementation Challenge Workshop III*, 1997. To appear.

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

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 306–317, May 1992.

[50] 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.

[51] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

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

[53] 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.

[54] P. D. MacKenzie and V. Ramachandran. Computational bounds for fundamental problems on general-purpose parallel models. In *Proc. 10th ACM Symp. on Parallel Algorithms and Architectures*, pages 152–163, June-July 1998.

[55] 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.

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

[57] 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.

[58] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proc. 24th International Symp. on Computer Architecture*, pages 85–97, June 1997.

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

[60] 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.

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

[62] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 29–39, July 1991.

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

[64] V. Ramachandran. A general purpose shared memory model for parallel computation. In *Proc. IMA Workshop on Parallel Algorithms*, September 1996.

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

[66] J. H. Reif and S. Sen. Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications. In *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, pages 327–337, July 1990.

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

[68] P. Stenström, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. 19th International Symp. on Computer Architecture*, pages 80–91, May 1992.

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

[70] 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.

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

[72] J. S. Vitter and E. A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 159–169, May 1990.

[73] 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.