To appear in the proceedings of IMA Workshop on Parallel Algorithms, Sept. 1996, Minneapolis, MN

A GENERAL PURPOSE SHARED-MEMORY MODEL FOR PARALLEL COMPUTATION

VIJAYA RAMACHANDRAN*

October 3, 1997

Abstract

We describe a general-purpose shared-memory model for parallel computation, called the QSM [22], which provides a high-level shared-memory abstraction for parallel algorithm design, as well as the ability to be emulated in an effective manner on the BSP, a lower-level, distributed-memory model. We present new emulation results that show that very little generality is lost by not having a 'gap parameter' at memory.

1 Introduction

The design of general-purpose models of parallel computation has been a topic of much importance and study. However, due to the diversity of architectures among parallel machines, this has also proved to be a very challenging task. The challenge here has been to find a model that is general enough to encompass the wide variety of parallel machines available, while retaining enough of the essential features of these diverse machines in order to serve as a reasonably faithful model of them.

Until recently there have been two approaches taken towards modeling parallel machines for the purpose of algorithm design. The more popular of the two approaches has been to design parallel algorithms on the PRAM, which is a synchronous, shared-memory model in which each processor can perform a local computation or access a shared memory location in a unit-time step, and there is global synchronization after each step. As a simple model at a high level of abstraction, the PRAM has served an important role, and most of the basic paradigms for parallel algorithm design as well as the basic ideas underlying the parallel algorithms for many problems have been developed on this model (see, e.g., [28, 24, 41]).

The other approach that has been used to design parallel algorithms has been to consider distributed-memory models, and tailor the parallel algorithm to a specific interconnection network that connects the processors and memory, e.g., mesh, hypercube, shuffle-exchange, cube-connected cycles, etc. There are several results known on embedding one of these networks, the *source network*, on to another, the *target network*(see, e.g., [31]), so that an efficient algorithm on the source network results in an efficient algorithm on the target network.

Neither of the above approaches has been very satisfactory. On the one hand, the PRAM is too high-level a model, and it ignores completely the latency and bandwidth limitations of real parallel machines. On the other hand, algorithms developed for a specific interconnection network are tailored to certain standard, regular networks, and hence are not truly general-purpose.

*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.

Thus is not surprising that a variety of other models have been proposed in the literature, (e.g., [2, 5, 6, 7, 9, 13, 15, 18, 23, 29, 32, 34, 35, 38, 40, 45, 46]) to address specific drawbacks of the PRAM although none of these are general-purpose models.

In recent years, distributed-memory models that characterize the interconnection network abstractly by parameters that capture its performance have gained much attention. An early work along these lines is the CTA [42]. More recently, the BSP model [43, 44] and the LogP model [14] have gained wide acceptance as general-purpose models of parallel computation. In these models the parallel machine is abstracted as a collection of processors-memory units with no global shared memory. The processors are interconnected by a network whose performance is characterized by a *latency* parameter L and a *gap* parameter g. The latency of the network is the time needed to transmit a message from one processor to another. The gap parameter g indicates that a processor can send no more than one message every g steps. This parameter reflects the bandwidth of the network – the higher the bandwidth, the lower is the value of g. The models may have some additional parameters, such as the overhead in sending messages, and the time for synchronization (in a model that is not asynchronous). In contrast to earlier fixed interconnection network models, the BSP and LogP models do not take into consideration the exact topology of the interconnection network.

The BSP and LogP models have become very popular in recent years, and many algorithms have been designed and analyzed on these models and their extensions (see, e.g., [4, 8, 17, 25, 27, 36, 47]). However, algorithms designed for these models tend to have rather complicated performance analyses, because of the number of parameters in the model as well as the need to keep track of the exact memory partition across the processors at each step.

Very recently, in [22] the issue of whether there is merit in developing a general-purpose model of parallel computation, starting with a shared-memory framework was explored. Certainly, shared-memory has been a widely-supported abstraction in parallel programming [30]. Additionally, the architectures of many parallel machines are either intrinsically shared-memory or support it using suitable hardware. The main issues addressed in [22] are the enhancements to be made to a simple shared-memory model such as the PRAM, and the effectiveness of the resulting model in capturing the essential features of parallel machines along the lines of the BSP and the LogP models.

The work reported in [22] builds on the results in [19] where a simple variant of the PRAM model is described in which the read-write steps are required to be *queuing*; this model is called the QRQW PRAM. Prior to this work there were a variety of PRAM models that differed depending on whether read or writes (or both) were *exclusive*, i.e., concurrent accesses to the same memory location in the same step are forbidden, or *concurrent*, i.e., such concurrent accesses are allowed. Thus earlier PRAM models were classified as EREW, CREW, and CRCW (see, e.g., [28]); the ERCW PRAM was studied more recently [33]. The latter two models (CRCW and ERCW PRAM) have several variants depending on how a concurrent write is resolved. In all models a step took unit time. In the QRQW PRAM model, concurrent memory accesses were allowed, but a step no longer took unit time. The *cost* of a step was the maximum number of requests to any single memory location. A randomized work-preserving emulation of the QRQW PRAM on a special type of BSP is given in [19], with slowdown only logarithmic in the number of processors[1].

In [22], the QRQW model was extended to the QSM model, which incorporates a *gap* parameter at processors to capture limitations in bandwidth. It is shown in [22] that the QSM has a random-

[1]An emulation is *work-preserving* if the processor-time bound on the emulated machine is the same as that on the machine being emulated, to within a constant factor. Typically, the emulating machine has a smaller number of processors and takes a proportionately larger amount of time to execute. The ratio of the running time on the emulating machine to the running time on the emulated machine is the *slowdown* of the emulation.

ized work-preserving emulation on the BSP that works with high probability[2] with only a modest slowdown. This is a strong validating point for the QSM as a general-purpose parallel computation model. Additionally, the QSM model has only two parameters – the number of processors p, and the gap parameter g for shared-memory requests by processors. Thus, the QSM is a simpler model than either the BSP or the LogP models.

The QSM has a gap parameter at the processors to capture the limited bandwidth of parallel machines, but it does not have a gap parameter at the memory. This fact is noted in [22], but is not explored further. In this paper we explore this issue by defining a generalization of the QSM that has (different) gap parameters at the processors and at memory locations. We present a work-preserving emulation of this generalized QSM on the BSP, and some related results. These results establish that the gap parameter is not essential at memory locations, thus validating the original QSM model.

The rest of this paper is organized as follows. Section 2 reviews the definition of the QSM model. Section 3 summarizes algorithmic results for the QSM. Section 4 presents the work-preserving emulation result on the QSM on the BSP using the gap parameter at memory locations. Section 5 concludes the paper with a discussion of some of the important features of the QSM.

Since we will make several comparisons of the QSM model to the BSP model, we conclude this section by presenting the definition of the Bulk-Synchronous Parallel (BSP) model [43, 44]. The BSP model consists of p processor/memory components that communicate by sending point-to-point messages. The interconnection network supporting this communication is characterized by a bandwidth parameter g and a latency parameter L. A BSP computation consists of a sequence of "supersteps" separated by bulk synchronizations. In each superstep the processors can perform local computations and send and receive a set of messages. Messages are sent in a pipelined fashion, and messages sent in one superstep will arrive prior to the start of the next superstep. The time charged for a superstep is calculated as follows. Let w_i be the amount of local work performed by processor i in a given superstep. Let s_i (r_i) be the number of messages sent (received) by processor i, and let $w = \max_{i=1}^{p} w_i$. Let $h = \max_{i=1}^{p}(\max(s_i, r_i))$; h is the maximum number of message sent or received by any processor, and the BSP is said to route an *h-relation* in this step. The *cost*, T, of a superstep is defined to be $T = \max(w,\ g \cdot h,\ L)$. The time taken by a BSP algorithm is the sum of the costs of the individual supersteps in the algorithm.

2 The Queuing Shared Memory Model (QSM)

In this section, we present the definition of the Queuing Shared Memory model.

Definition 2.1 *[22] 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]A probabilistic event occurs *with high probability (w.h.p.),* if, for any prespecified constant $\delta > 0$, it occurs with probability $1 - 1/n^{\delta}$, where n is the size of the input. Thus, we say a randomized algorithm runs in $O(f(n))$ time w.h.p. if for every prespecified constant $\delta > 0$, there is a constant c such that for all $n \geq 1$, the algorithm runs in $c \cdot f(n)$ steps or less with probability at least $1 - 1/n^{\delta}$.

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 BSP. In this emulation, the shared memory reads and writes at a processor are issued in a pipelined manner, to amortize against the delay (latency) in accessing the shared memory, and are not guaranteed to complete until the end of the phase. On the other hand, each of the local compute operations are assumed to take unit time in the intended emulation, and hence the values they compute can be used within the same phase.

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

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

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

Definition 2.3 *Consider a* QSM *phase with maximum contention κ. Let $m_{op} = \max_i\{c_i\}$ for the phase, i.e. the maximum over all processors i of its number of local operations, and let $m_{rw} = \max\{1, \max_i\{r_i, w_i\}\}$ for the phase. Then the time cost for the phase is $\max(m_{op}, g \cdot m_{rw}, \kappa)$. (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 we choose to use the former definition.) 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.*

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 in [19].

We note a couple of special features about the QSM model.

- There is an asymmetry in the use of the gap parameter: The model charges g per shared-memory request at a given processor (the $g \cdot m_{rw}$ term in the cost metric), but it only charges 1 per shared-memory request at a given memory location (the κ term in the cost metric). This

Summary of Algorithmic Results		
problem (n = size of input)	QSM **result**[3]	source
prefix sums, list ranking, etc.[4]	$O(g \lg n)$ time, $\Theta(gn)$[5] work	EREW
linear compaction	$O(\sqrt{g \lg n} + g \lg \lg n)$ time, $O(gn)$ work w.h.p.	QRQW [19]
random permutation	$O(g \lg n)$ time, $\Theta(gn)$ work w.h.p.	QRQW [20]
multiple compaction	$O(g \lg n)$ time, $\Theta(gn)$ work w.h.p.	QRQW [20]
parallel hashing	$O(g \lg n)$ time, $\Theta(gn)$ work w.h.p.	QRQW [20]
load balancing, max. load L	$O(g\sqrt{\lg n} \lg \lg L + \lg L)$ time, $\Theta(gn)$ work w.h.p.	QRQW [20]
broadcast to n mem. locations	$\Theta(g \lg n/(\lg g))$ time, $\Theta(gn)$ work	QSM [1]
sorting	$O(g \lg n)$ time, $O(gn \lg n)$ work	EREW [3, 12]
simple fast sorting (sample sort)	$O(g \lg n + \lg^2 n/(\lg \lg n))$ time, $O(gn \lg n)$ work w.h.p.	QSM [22]
work-optimal sorting (sample sort)	$O(n^\epsilon \cdot (g + \lg n))$ time, $\epsilon > 0$, $\Theta(gn + n \lg n)$ work w.h.p.	BSP [17]

Table 1: *Efficient* QSM *algorithms for several fundamental problems.*

appears to make the QSM model more powerful than real parallel machines, since bandwidth limitations would normally dictate that there should be a gap parameter at memory as well as at processor (the two gap parameters need not necessarily be the same).

- The model considers contention only at individual memory locations, not at memory modules. In most machines, memory locations are organized in *memory banks* and access to each bank is queuing. Here again it appears that there is a mis-match between the QSM model and real machines.

Both of the features of the QSM highlighted above give more power to the QSM than would appear to be warranted by current technology. However, in Section 4 we show that we can obtain a work-preserving emulation of the QSM on the BSP with only a modest slowdown. Since the BSP is considered to be a fairly good model of current parallel machines, this is a validation of the QSM as a general-purpose parallel computation model. It is also established in Section 4 that there is not much loss in generality in having the gap parameter only at processors, and not at memory locations.

3 Algorithmic Results

Table 1 summarizes the time and work bounds for QSM algorithms for several basic problems. Most of these results are the consequence of the following four Observations, all of which are from [22].

[3]The time bound stated is the fastest for the given work bound; by Observation 3.1, any slower time is possible within the same work bound.

[4]By Observation 3.2 any EREW result maps on to the QSM with the work and time both increasing by a factor of g. The two problems cited in this line are representatives of the large class of problems for which logarithmic time, linear work EREW PRAM algorithms are known (see, e.g., [28, 24, 41]).

[5]The use of Θ in the work or time bound implies that the result is the best possible, to within a constant factor.

Observation 3.1 *(Self-simulation) 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.*

In view of Observation 3.1 we will state the performance of a QSM algorithm as running in time t and work w (i.e., with $\Theta(w/t)$ processors); by the above Observation the same algorithm will run on any smaller number of processors in proportionately larger time so that the work remains the same, to within a constant factor.

Observation 3.2 (EREW *and* QRQW *algorithms on* QSM) *Consider a* QSM *with gap parameter g.*

1. *An* EREW *or* 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. *An* EREW *or* QRQW PRAM *algorithm in the work-time framework that runs in time t while performing work w implies a* QSM *algorithm that runs in time at most $t \cdot g$ with w/t processors.*

Observation 3.3 *(Simple lower bounds for* QSM*) 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 $\Omega(n \cdot g)$.*

There is a large collection of logarithmic time, linear work EREW and QRQW PRAM algorithms available in the literature. By Observation 3.2 these algorithms map on to the QSM with the time and work both increased by a factor of g. By Observation 3.3 the resulting QSM algorithms are work-optimal (to within a constant factor).

Observation 3.4 (BSP *algorithms on* QSM) *Let* **A** *be an oblivious* BSP *algorithm, i.e., an algorithm in which the pattern of memory locations accessed by the algorithm is determined by the length of the input, and does not depend on the actual value(s) of the input. Then algorithm* **A** *can be mapped on to a* QSM *with the same gap parameter to run in the time and work bound corresponding to the case when the latency $L = 1$ on the* BSP.

Since the BSP is a more low-level model than the QSM, it may seem surprising that not all BSP algorithms are amenable to being adapted on the QSM with the performance stated in Observation 3.4. However, it turns out that the BSP model has some additional power over the QSM which is seen as follows. A BSP processor π could write a value into the local memory of another processor π' without π' having explicitly requested that value. Then, at a later step, π' could access this value as a local unit-time computation. On a QSM the corresponding QSM processor π'_Q would need to perform a read on global memory at the later step to access the value, thereby incurring a time cost of g. In [22] an explicit computation is given that runs faster on the BSP than on the QSM.

One point to note regarding the fact that the BSP is in some ways more powerful than the QSM, is that it is not clear that we want a general-purpose bridging model to incorporate these features of the BSP. For instance, 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 using this additional power of the BSP over the QSM may not be that widely applicable.

The paper [22] also presents a randomized work-preserving emulation of the BSP on the QSM that incurs a slow-down that is only logarithmic in the number of processors. Thus, if a modest slow-down is acceptable, then in fact, any BSP algorithm can be mapped on to the QSM in a work-preserving manner. For completeness, we state here the result regarding the emulation of the BSP on the QSM. The emulation algorithm and the proof of the following theorem can be found in full version of [22].

Theorem 3.5 *An algorithm that runs in time $t(n)$ on an n-component* BSP *with gap parameter g and latency parameter L, where $t(n)$ is bounded by a polynomial in n, can be emulated with high probability on a* QSM *with the same gap parameter g to run in time $O(t(n) \cdot \lg n)$ with $n/\lg n$ processors.*

In summary, by Theorem 3.5, any BSP algorithm can be mapped on to the QSM in a work-preserving manner (w.h.p.) with only a modest slowdown. Additionally, by Observation 3.4, for oblivious BSP algorithms there is a very simple optimal step-by-step mapping of the oblivious BSP algorithm on to the QSM.

4 QSM Emulation Results

Recall that we defined the Bulk Synchronous Parallel (BSP) model of [43, 44] in Section 1. In this section we present a work-preserving emulation of the QSM on the BSP.

One unusual feature of the QSM model that we pointed out in Section 2 is the absence of a gap parameter at the memory: Recall that the QSM model has a gap parameter g at each processor attempting to access global memory, but accesses at individual global memory locations are processed in unit time per access. In the following, we assume a more general model for the QSM, namely the QSM(g, d), where g is the gap parameter at the processors and d is the gap parameter at memory locations. We present a work-preserving emulation of the QSM(g, d) on the BSP, and then demonstrate work-preserving emulations between QSM(g, d) and QSM(g, d'), for any $d, d' > 0$. Thus, one can move freely between models of the QSM with different gap parameters at the memory locations. In particular this means that one can transform an algorithm for the QSM$(g, 1)$, which is the standard QSM, into an algorithm for QSM(g, d) in a work-preserving manner (and with only a small increase in slowdown). Given this flexibility, it is only appropriate that the standard QSM is defined as the 'minimal' model with respect to the gap parameter at memory locations, i.e., the model that sets the gap parameter at memory locations to 1.

We compare the cost metrics of the BSP and the QSM(g, d) as follows. We can equate the g parameters in the two models, and the local computation w_i on the ith BSP processor with the local computation c_i on the ith QSM processor (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 BSP processor, and let $h_r = \max_{i=1}^{p} r_i$, the maximum number of read/write requests to any one BSP processor. The BSP charges the maximum of w, $g \cdot h_s$, $g \cdot h_r$, and L. The QSM(g, d), on the other hand, charges the maximum of w, $g \cdot h_s$, and $d \cdot \kappa$, where $\kappa \in [1..h_r]$ is the maximum number of read/write requests to any one memory *location*. Despite the apparent mis-match between some of the parameters, we present below, a work-preserving emulation of the QSM(g, d) on the BSP.

The proof of the emulation result requires the following result by Raghavan and Spencer.

Theorem 4.1 *[39] Let $a_1, \ldots, a_r$ be reals in $(0,1]$. Let $x_1, \ldots, x_r$ be independent Bernoulli trials with $\mathbf{E}\,(x_j) = \rho_j$. Let $S = \sum_{j=1}^{r} a_j x_j$. If $\mathbf{E}\,(S) > 0$, then for any $\nu > 0$*

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

We now state and prove the work-preserving emulation result. A similar theorem is proved in [22], which presents an emulation of the QSM on a $(d, \mathbf{x})$-BSP. The $(d, \mathbf{x})$-BSP is a variant of the BSP that has different gap parameters for requesting messages and for sending out the responses to the requests (this models the situation where the distributed memory is in a separate cluster of memory banks, rather than within the processors). In the emulation below, the BSP is the standard model, but the QSM has been generalized as a QSM(g,d), with a gap parameter d at the memory locations.

The emulation algorithm in the following theorem assumes that the shared memory of the QSM(g,d) is distributed across the BSP components in such a way that each shared memory location of the QSM(g,d) is equally likely to be assigned to any of the BSP components, independent of the other memory locations, and independent of the QSM(g,d) algorithm. In practice one would distribute the shared memory across the BSP processors using a random hash function from a class of universal hash functions that can be evaluated quickly (see, e.g., [11, 37, 26]).

Theorem 4.2 *A p'-processor* QSM(g,d) *algorithm that runs in time t' can be emulated on a p-processor* BSP *in time $t = t' \cdot \frac{p'}{p}$ w.h.p. provided*

$$p \leq \frac{p'}{(L/g) + (g/d)\lg p}$$

and t' is bounded by a polynomial in p.

Proof. The emulation algorithm is quite simple. The shared memory of the QSM(g,d) is hashed onto the p processors of the BSP so that any given memory location is equally likely to be mapped onto any one of the BSP processors. The p' QSM processors are mapped on to the p BSP processors in some arbitrary way so that each BSP processor has at most $\lceil p'/p \rceil$ QSM processors mapped on to it. In each step, each BSP processor emulates the computation of the QSM processors that are mapped on to it.

In the following we show that the above algorithm provides a work-preserving emulation of the QSM(g,d) on the BSP with the performance bounds stated in the theorem. In particular, if the ith step of the QSM(g,d) algorithm has time cost t_i, we show that this step can be emulated on the BSP in time $O((p'/p)t_i)$ w.h.p.

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 p-processor BSP in time $(p'/p) \cdot t_i$, since each BSP processor emulates p'/p QSM processors.

By the QSM(g,d) cost metric, we have that κ, the maximum number of requests to the same location, is at most t_i/d, and h, the maximum number of requests by any one QSM processor, is at most t_i/g. For the sake of simplicity in the analysis, we add dummy memory requests to each QSM 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 each dummy location receiving up to κ requests. In this way, the maximum number of requests to the same location remains κ, and the total number of requests is $Z = p't_i/g$.

Let $i_1, i_2, \ldots, i_r$ 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, $1 \le j \le r$. Note that $\sum_{j=1}^{r} \kappa_j = Z$.

Consider a BSP processor π. For $j = 1, \ldots, r$, let x_j be an indicator binary random variable which is 1 if memory location i_j is mapped onto processor π, and is 0 otherwise. Thus, $\mathbf{Prob}\,(x_j = 1)$ is $1/p$.

Let $a_j = \kappa_j d/t_i$; we view a_j as the *normalized contention* to memory location i_j. Since $\kappa_j \cdot d \le t_i$, we have that $a_j \in (0, 1]$.

Let $S^\pi = \sum_{j=1}^{r} a_j x_j$; S^π, the *normalized request load to processor* π, is the weighted sum of Bernoulli trials. The expected value of S^π is

$$\mathbf{E}\,(S^\pi) = \sum_{j=1}^{r} \frac{a_j}{p} = \frac{d}{p \cdot t_i} \sum_{j=1}^{r} \kappa_j = \frac{d}{p \cdot t_i} \cdot Z = \frac{d}{p} \cdot \frac{p'}{g} = \frac{d}{g} \cdot \frac{p'}{p} \quad .$$

We now use Theorem 4.1 to show that it is highly unlikely that $S^\pi > 2e \cdot \mathbf{E}\,(S^\pi)$.

We apply Theorem 4.1 with $\nu = 2e - 1$. Then,

$$(1+\nu)\mathbf{E}\,(S^\pi) = 2e \cdot \frac{d}{g} \cdot \frac{p'}{p} \quad . \tag{1}$$

Therefore,

$$\mathbf{Prob}\left(S^\pi > 2e \cdot \frac{d}{g} \cdot \frac{p'}{p}\right) < \left(\frac{e}{2e}\right)^{2e \cdot \mathbf{E}(S^\pi)} = \left(\frac{1}{2}\right)^{2e \cdot \frac{d}{g} \cdot \frac{p'}{p}} < \left(\frac{1}{2}\right)^{2e \lg p} = p^{-2e}$$

since $p'/p > (g/d) \lg p$.

Let h^π be the number of requests to memory locations mapped to processor π. Then,

$$h^\pi = \sum_{j=1}^{r} \kappa_j x_j = \frac{t_i}{d} \sum_{j=1}^{r} a_j x_j = \frac{t_i}{d} \cdot S^\pi \quad .$$

Thus $\mathbf{Prob}\,(h^\pi > 2e \cdot (t_i/g) \cdot (p'/p))$ is $O(1/p^{2e})$. Hence the probability that, at any one of the processors, the number of requests to memory locations mapped to that processor exceeds $2e \cdot (t_i/g) \cdot (p'/p)$ is $O(1/p^{2e-1})$. Hence w.h.p. the number of memory requests to any processor is $O((t_i/g) \cdot (p'/p))$.

By definition, the time taken by the BSP to complete the emulation of the ith step is $T_i = \max(w,\ g \cdot h,\ L)$, where w is the maximum number of local computation steps at each processor, and h is the maximum number of messages sent or received by any processor. As discussed at the beginning of this proof, $w \le t_i \cdot (p'/p)$. Since the maximum number of messages sent by any processor is no more than $(t_i/g) \cdot (p'/p)$ and the maximum number of requests to memory locations mapped on to any given processor is no more than $2e \cdot (t_i/g) \cdot (p'/p)$ w.h.p, it follows that $g \cdot h = O(t_i \cdot (p'/p))$ w.h.p. Finally, since $t_i \ge g$ and $p'/p \ge L/g$, it follows that $t_i \cdot (p'/p) \ge L$.

Thus, w.h.p., the time taken by the BSP to execute step i is

$$T_i = O(t_i \cdot (p'/p))$$

This completes the proof of the theorem. ■

Note that the emulation given above is work-preserving since $p \cdot t = p' \cdot t'$. Informally the proof of the theorem shows that an algorithm running in time t' on a p'-processor QSM(g, d) can be executed in time $t = (p'/p) \cdot t'$ on a p-processor BSP (where p has to be smaller than p' by a factor of at least

$((L/g)+(g/d)\lg p))$ by assigning the memory locations and the QSM(g,d) processors randomly and equally among the p BSP processors, and then having each BSP processor execute the code for the QSM(g,d) processors assigned to it. (Actually the assignment of the QSM processors on the BSP need not be random – any fixed assignment that distributes the QSM processors equally among the BSP processors will do. The memory locations, however, should be distributed randomly.) The fastest running time achievable on the BSP is somewhat smaller than the fastest time achievable on the QSM(g,d) – smaller by the factor $((L/g)+(g/d)\lg p)$. The L/g term in the factor arises because the BSP has to spend at least L units of time per superstep to send the first message, and in order to execute this step in a work-preserving manner, it should send at least the number of messages it can send in L units of time, namely L/g messages. The $(g/d)\lg p$ term comes from the probabilistic analysis on the distribution of requested messages across the processors; the probabilistic analysis in the proof shows that the number of memory requests per processor (taking contention into consideration) is within a factor of $2\cdot e$ times the expected number of requests w.h.p. when the memory locations are distributed randomly across the p BSP processors, and p is smaller than p' by a factor of $(g/d)\cdot\lg p$.

We now give a deterministic work-preserving emulation of QSM(g,d') on QSM(g,d), for any $d,d'>0$.

Observation 4.3 *There is a deterministic work-preserving emulation of* QSM(g,d') *on* QSM(g,d) *with slowdown* $O(\lceil\frac{d}{d'}\rceil)$.

Proof. If $d\le d'$ then clearly, each step on QSM(g,d') will map on to QSM(g,d) without any increase in time (there could be a decrease in the running time through this mapping, but that does not concern us here).

If $d>d'$, let $r=\lceil\frac{d}{d'}\rceil$. Given a p'-processor algorithm on QSM(g,d') we map it on to a $p=\frac{p'}{r}$ processor QSM(g,d) by mapping the p' processors of QSM(g,d') uniformly on to the p processors of QSM(g,d). Now consider the ith step of the QSM(g,d') algorithm. Let it have time cost t_i'. On QSM(g,d) the increase in time cost of this step arising from local computations and requests from processors is no more than $r\cdot t_i'$ since each processor in QSM(g,d) will have to emulate at most r processors of QSM(g,d'). The delay at the memory locations in QSM(g,d) is increased by a factor of exactly r over the delay in QSM(g,d'), since the memory map is identical in both machines. Thus the increase in time cost on the QSM(g,d) is no more than $r\cdot t_i'$, and hence this is a work-preserving emulation of QSM(g,d') on QSM(g,d) with a slowdown of $\frac{p'}{p}=\frac{d}{d'}$. ∎

Observation 4.3 validates the choice made in the QSM model not to have a gap parameter at the memory. Since the proof of this observation gives a simple method of moving between QSM(g,d) models with different gap parameters at memory, it is only appropriate to choose the 'minimal' one as the canonical model, namely, the one with no gap parameter at memory locations.

Note that there could be a slight increase in slowdown when one designs an algorithm on QSM(g,d') which does not use the d parameter that most accurately models the machine under consideration. In situations where this is an important consideration, one should tailor one's algorithm to the correct d parameter.

5 Discussion

In this paper, we have described the QSM model of [22], reviewed algorithmic results for the model, and presented a randomized work-preserving emulation for a generalization of the QSM on the BSP. The emulation results validate the QSM as a general-purpose model of parallel computation, and they also validate the choice made in the definition of the QSM not to have a gap parameter at memory locations.

We conclude this paper by highlighting some important features of the QSM model.

- The QSM model is very simple – it has only two parameters, p, the number of processors, and g, the gap parameter at processors.
- Section 3 summarizes algorithmic results for the QSM derived from a variety of sources – EREW PRAM, QRQW PRAM, BSP – as well as algorithms tailored for the QSM. This is an indication that the QSM model is quite versatile, and that tools developed for other important parallel models map on to the QSM in an effective way.
- The randomized work-preserving emulation of the QSM on the BSP presented in Section 4 validates it as a general-purpose parallel computation model.
- The QSM is a shared-memory model. Given the wide-spread use and popularity of the shared-memory abstraction, this makes the QSM a more attractive model than the distributed-memory BSP and LogP models.
- It can be argued that the QSM models a wider variety of parallel architectures than the BSP or LogP models. The distributed-memory feature of the latter two models causes a mis-match to machines that have the shared-memory organized in a separate cluster of memory banks (e.g., the Cray C90 and J90, the SGI Power Challenge and the Tera MTA). In such cases there would be no reason for the number of memory banks to equal the number of processors, which is the situation modeled by the BSP and LogP models. This point is elaborated in some detail in [22].
- The queuing rule for concurrent memory accesses in the QSM is crucial in matching it to real machines. In addition to the work-preserving emulation of the QSM on BSP given in Section 4, in Section 3 we stated a theorem that gives a randomized work-preserving emulation of the BSP on the QSM. Thus, there is a tight correspondence between the power of the QSM and the power of the BSP. Such a correspondence is not available for any of the other memory access rules for shared-memory (e.g., for exclusive memory access or for unit-cost concurrent memory access). Further, on a QSM with memory accesses required to be exclusive rather than queuing, no linear-work, polylog-time algorithm is known for generating a random permutation or for performing multiple compaction; in contrast, randomized logarithmic time, linear-work algorithms that work correctly with high probability are known for the QSM. Thus the queuing rule appears to allows one to design more efficient algorithms than those known for exclusive memory access. On the other hand, if the QSM is enhanced to have unit-cost concurrent memory accesses, this appears to give the model more power than is warranted by the performance of currently available machines. For more detailed discussions on the appropriateness of the queue metric, see [19, 22].
- The QSM is a bulk-synchronous model, i.e., a step consists of a sequence of pipe-lined requests to memory, together with a sequence of local operations, and there is global synchronization between successive steps. For a completely asynchronous general-purpose shared-memory model, a promising candidate is the QRQW ASYNCHRONOUS PRAM [21], augmented with the gap parameter.

Acknowledgement I would like to thank Phil Gibbons and Yossi Matias for innumerable discussions on queuing shared memory models; this collaboration led to the results in [19, 20, 21, 22].

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*, June 1997. To appear.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] J. L. Carter and M.N. Wegman. Universal classes of hash functions. *J. Comput. Syst.Sci.* 18:143–154, 1979.

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.*

[21] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write Asynchronous PRAM model. *Theoretical Computer Science: Special Issue on Parallel Processing.* To appear. Preliminary version in *Euro-Par'96, Lecture Notes in Computer Science, Vol. 1124*, pages 279–292. Springer, Berlin, August 1996.

[22] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, June 1997. To appear.

[23] 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.

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

[25] 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.

[26] A. Karlin and E. Upfal. Parallel hashing – An efficient implementation of shared memory. *J. ACM*, 35:4, pages 876–892, 1988.

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

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

[29] 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.

[30] 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.

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

[32] 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.

[33] P.D. MacKenzie and V. Ramachandran. ERCW PRAMs and optical communication. *Theoretical Computer Science: Special Issue on Parallel Processing.* To appear. Preliminary version in *Euro-Par'96, Lecture Notes in Computer Science, Vol. 1124*, pages 293-303. Springer, Berlin, August 1996.

[34] 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.

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

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

[37] 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.

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

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

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

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

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

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

[44] 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.

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

[46] 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.

[47] 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.