Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

J. Parallel Distrib. Comput. 63 (2003) 1175–1192

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Emulations between QSM, BSP and LogP: a framework for general-purpose parallel algorithm design ☆

Vijaya Ramachandran,[a,*,1] Brian Grayson,[b,2] and Michael Dahlin[a,3]

[a] *Department of Computer Sciences, University of Texas, Austin, TX 78712, USA*
[b] *Motorola Somerset Design Center, Austin, TX 78729, USA*

Received 27 May 1999; revised 29 March 2003; accepted 2 April 2003

Abstract

We present work-preserving emulations with small slowdown between LogP and two other parallel models: BSP and QSM. In conjunction with earlier work-preserving emulations between QSM and BSP, these results establish a close correspondence between these three general-purpose parallel models. Our results also correct and improve on results reported earlier on emulations between BSP and LogP. In particular we shed new light on the relative power of stalling and non-stalling LogP models.

The QSM is a shared-memory model with only two parameters—p, the number of processors, and g, a bandwidth parameter. The simplicity of the QSM parameters makes QSM a convenient model for parallel algorithm design, and simple work-preserving emulations of QSM on BSP and QSM on LogP show that algorithms designed for the QSM will also map quite well to these other models. The simplicity and generality of QSM present a strong case for the use of QSM as the model of choice for parallel algorithm design.

We present QSM algorithms for three basic problems—prefix sums, sample sort and list ranking. We show that these algorithms are optimal in terms of both the total work performed and the number of 'phases' for input sizes of practical interest. For prefix sums, we present a matching lower bound that shows our algorithm to be optimal over the complete range of these parameters. We then examine the predicted and simulated performance of these algorithms. These results suggest that QSM analysis will predict algorithm performance quite accurately for problem sizes that arise in practice.

Keywords: Emulations; General-purpose parallel models; BSP; LogP; QSM; Parallel algorithm design; Prefix sums; Sample sort; List ranking

1. Introduction

There is a vast amount of literature on parallel algorithms for various problems. However, algorithms developed using traditional approaches such as PRAM or fixed-interconnect networks do not map well to real machines. In recent years several *general-purpose parallel models* have been proposed—BSP [24], LogP [6], QSM and s-QSM [12]. These models attempt to capture the key features of real machines while retaining a reasonably high-level programming abstraction. Of these models, the QSM and s-QSM models are the simplest for two reasons: each has only 2 parameters, and each is shared-memory (shared-memory models are generally more convenient than message passing for developing parallel algorithms).

There are both practical and algorithmic reasons for developing a general model for parallel algorithm design.

- On the practical side, the long-term goal is to be able to replace hand tuning with automated methods for larger fractions of programs. The argument here is similar to the argument of hand-tuned assembly versus compiled code, and the goal is to reach a similar point, where automatic methods able to do as well as or better than the average programmer. As parallel programming becomes more and more common—a decade ago parallel supercomputers

☆ This work was supported in part by an NSF CISE Grant (CDA-9624082), and grants from Intel, Novell, and Sun. A summary of this work appears in *Proceedings* of the 1999 ACM-SIAM Symposium on Discrete Algorithms (SODA'99).

* Corresponding author. Fax: +1-512-471-8885.
E-mail addresses: vlr@cs.utexas.edu (Vijaya Ramachandran), brian.grayson@motorola.com (Brian Grayson), dahlin@cs.utexas.edu (Michael Dahlin).

[1] Supported in part by NSF Grant CCR-9988160.
[2] Supported in part by an NSF Graduate Fellowship.
[3] Supported in part by NSF CAREER Grant 9733842.

0743-7315/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2003.04.001

were rare, today Beowulf clusters are within the reach of moderate-sized organizations, tomorrow most chips on individual desktops may have parallel cores—we expect parallel programming to become more mainstream, and for higher-level programming techniques to become increasingly important.

- On the algorithmic side it is important to identify the simplest programming models that expose the right system and algorithmic properties. Giving algorithm designers a simpler model will make it easier for them to focus on the underlying idea without being distracted by less fundamental concerns.

In this paper we first provide two strong justifications for utilizing the QSM models for developing general-purpose parallel algorithms:

1. We present *work-preserving emulations* with only *modest* (*polylog*) *slowdown* between the LogP model and the other 3 models. (Work-preserving emulations between BSP, QSM and s-QSM were presented earlier in [12] (see also [20]).) An emulation is work-preserving if the processor-time bound on the emulating machine is the same as that on the machine being emulated, to within a constant factor. The slowdown of the emulation is the ratio of the number of processors on the emulated machine to the number on the emulating machine. Typically, the emulating machine has a somewhat smaller number of processors and takes proportionately longer to execute. For many situations of practical interest, both the original algorithm and the emulation would be mapped to an even smaller number of physical processors and thus would run within the same time bound to within a constant factor.

   Our results indicate that the four models are more or less interchangeable for the purpose of algorithm design. The only mis-match we have is between the 'stalling' and 'non-stalling' LogP models. Here we show that an earlier result claimed in [3] is erroneous by giving a counterexample to their claim. (The journal version [4] of paper [3] corrects this error, attributing the correction to the first author of our paper, and citing the conference version [21] of our paper.)
2. The emulations of s-QSM and QSM on the other models are quite simple. Conversely, the reverse emulations—of BSP and LogP on shared-memory—are more involved. The difference is mainly due to the 'message-passing' versus 'shared-memory' modes of accessing memory. Although message passing can easily emulate shared memory, the known work-preserving emulations for the reverse require sorting as well as 'multiple compaction.' Hence, although such emulations are efficient since they are work-preserving with only logarithmic slowdown, the algorithms thus derived are fairly complicated.

Since both message-passing and shared-memory are widely used in practice, we suggest that a high-level general-purpose model should be one that can be simply and efficiently implemented on both message-passing and shared-memory systems. The QSM and s-QSM have this feature. Additionally, these two models have a smaller number of parameters than LogP or BSP, and they do not have to keep track of the distributed memory layout.

To facilitate using QSM or s-QSM for designing general-purpose parallel algorithms, we develop a suitable cost metric for such algorithms and evaluate several algorithms both analytically and experimentally against this metric. The metric asks algorithms to (1) minimize work (where 'work' is defined in the next section), (2) minimize the number of 'phases' (defined in the next section), and (3) maximize parallelism, subject to the above requirements. In the rest of the paper we present QSM algorithms for three basic problems: *prefix sums*, *sample sort*, and *list ranking*, and we show that they have provably good behavior under this metric. Finally we describe simulation results for these algorithms that indicate that the difference between the QSM and BSP cost metrics is small for these algorithms for reasonable problem sizes.

A popular model for parallel algorithm design is the PRAM (see, e.g., [17]). We do not discuss the PRAM in this paper since it does not fall within the frame-work of a 'general-purpose model' for parallel algorithm design in view of the fact that it ignores all communication costs. However, the QSM and s-QSM can be viewed as realistic versions of the PRAM. Extensive discussions on the relation between the PRAM model and the QSM model can be found in [11,9,12,20].

The rest of this paper is organized as follows. Section 2 provides background on the models examined in this paper and Section 3 presents our emulation results. Section 4 presents a cost metric for QSM and describes and analyzes some basic algorithms under this metric. Section 5 describes experimental results for the three algorithms and Section 6 summarizes our conclusions.

2. General-purpose parallel models

In this section, we briefly review the BSP, LogP, and QSM models.

BSP model. The Bulk-Synchronous Parallel (BSP) model [24] 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. It is assumed that in each superstep messages are sent by a processor based on its state at the start of the 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 and let s_i (r_i) be the number of messages sent (received) in the superstep by processor i. Let $h_s = \max_{i=1}^{p} s_i$, $h_r = \max_{i=1}^{p} r_i$, and $w = \max_{i=1}^{p} w_i$. Let $h = \max(h_s, h_r)$; h is the maximum number of messages sent or received by any processor in the superstep, and the BSP is said to route an *h-relation* in this superstep. The *cost*, T, of the superstep is defined to be $T = \max(w, gh, L)$. The time taken by a BSP algorithm is the sum of the costs of the individual supersteps in the algorithm.

The *work* performed by the computation is the processor-time product.

LogP model. The LogP model [6] consists of p processor/memory components communicating with point-to-point messages. It has the following parameters.

- *Latency* l: Time taken by network to transmit a message from one processor to another is at most l.
- *Gap* g: A processor can send or receive a message no faster than once every g units of time.
- *Capacity constraint*: A receiving processor can have no more than $\lceil l/g \rceil$ messages in transit to it.
- *Overhead* o: To send or receive a message, a processor spends o units of time to transfer the message to or from the network interface; during this period of time the processor cannot perform any other operation.

If the number of messages in transit to a destination processor π is $\lceil l/g \rceil$, then a processor that needs to send a message to processor π *stalls* and does not perform any operation until it can send the message.

The *time* taken by a LogP algorithm is the amount of time needed for the computation and communication to terminate at all processors, assuming each message takes maximum time (l units) in transit.

The *work* performed by the computation is the processor-time product.

QSM and s-QSM models. The Queuing Shared Memory (QSM) model [12] consists of p processors, each with its own private memory, that communicate by reading and writing shared memory. Processors execute a sequence of synchronized phases, each consisting of an arbitrary interleaving of shared memory reads, shared memory writes, and local computation. QSM implements a *bulk-synchronous* programming abstraction in that (i) each processor can execute several instructions within a phase but the values returned by shared-memory reads issued in a phase cannot be used in the same phase and (ii) the same shared-memory location cannot be both read and written in the same phase.

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.

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.

Consider a QSM phase with maximum contention κ. Let m_{op} be the maximum number of local operations performed by any processor in this phase, and let m_{rw} be the maximum number of read and write requests to shared memory issued by any processor. Then the *time cost* for the phase is $\max(m_{\text{op}}, gm_{\text{rw}}, \kappa)$. 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 s-QSM (*Symmetric QSM*) is a QSM in which the time cost for a phase is $\max(m_{\text{op}}, gm_{\text{rw}}, g\kappa)$, i.e., the gap parameter is applied to the accesses at memory as well as to memory requests issued at processors.

The particular instance of the QSM model in which the gap parameter, g, equals 1 is the Queue-Read Queue-Write (QRQW) PRAM model defined in [11].

Note that although the QSM models are shared-memory they explicitly reward careful data placement since local memory is cheap but it is expensive to access global memory. The results we present in this paper indicate that once one has accounted for local memory in the algorithm design, it is not necessary to burden the programmer with more detailed global memory layout.

3. Emulation results

The results on work-preserving emulations between models are shown in Table 1 with new results printed within boxes. In this section we focus on three aspects of these emulations. First, we develop new, work-preserving emulations of QSM or BSP on LogP; previously known emulations [3] required sorting and increased both time and work by a logarithmic factor. Second, we provide new analysis of the known emulation of LogP on BSP [3]; we provide a counter-example to the claim that this emulation holds for the stalling LogP model, and we observe that the original non-work-preserving emulation may be trivially extended to be work-preserving for non-stalling LogP. Third, we discuss the fact that known emulations of message passing on shared memory require sorting and multiple-compaction, complicating emulations of BSP or LogP algorithms on shared memory.

Table 1
All results are randomized and hold whp except those marked as 'det.', which are deterministic emulations. Results in which the LogP model is either the emulated or the emulating machine are new results that appear boxed in the table and are reported in this paper. (For exact expressions, including sub-logarithmic terms, please see the text of the paper.) The remaining results are in [12,20].

Slowdown of Work-Preserving Emulations (sublogarithmic factors have been rounded up for ease of display)				
Emulated model (p processors)	Emulating model			
	BSP	LogP (stalling)	s-QSM	QSM
BSP		$O(\log^4 p + (l/g)\log^2 p)$	$O(\lceil \frac{g \log p}{L} \rceil)$	$O(\lceil \frac{g \log p}{L} \rceil)$
LogP (non-stalling)	$O(L/l)$ (det.)[a]	1 (det.)	$O(\lceil \frac{g \log p}{l} \rceil)$	$O(\lceil \frac{g \log p}{l} \rceil)$
s-QSM	$O((L/g) + \log p)$	$O(\log^4 p + (l/g)\log^2 p)$		1 (det.)
QSM	$O((L/g) + g \log p)$	$O(\log^4 p + (l/g)\log^2 p + g \log p)$	$O(g)$ (det.)	

[a] This result is presented in [3], but it is stated erroneously that it holds for stating Log programs. We provide a countersample in Claim 3.8 and Theorem 3.9 here.

We focus on *work-preserving* emulations. An emulation is work-preserving if the processor-time bound on the emulating machine is the same as that on the machine being emulated, to within a constant factor. The ratio of the running time on the emulating machine to the running time on the emulated machine is the *slowdown* of the emulation. Typically, the emulating machine has a smaller number of processors and takes proportionately longer to execute. For instance, consider the entry in Table 1 for the emulation of s-QSM on BSP. It states that there is a randomized work-preserving emulation of s-QSM on BSP with a slowdown of $O(L/g + \log p)$. This means that, given a p-processor s-QSM algorithm that runs in time t (and hence with work $w = pt$), the emulation algorithm will map the p-processor s-QSM algorithm on to a p'-processor BSP, for any $p' \leqslant p/((L/g) + \log p)$, to run on the BSP in time $t' = O(t(p/p'))$ whp in p. Note that if sufficient parallelism exists, for a machine with p physical processors, one would typically design the BSP algorithm on $\Theta((L/g) + \log p)p$ or more processors, and then emulate the processors in this BSP algorithm on the p physical processors. In such a case, the performance of the BSP algorithm on p processors and the performance of the QSM emulation on p processors would be within a constant factor of each other. Since large problems are often the ones worth parallelizing, we expect this situation to be quite common in practice.

Many of our algorithms are randomized. We will say that an algorithm *runs in time t whp in n* if the probability that the time exceeds t is less than $1/n^c$, for some constant $c > 0$.

3.1. Work-preserving emulations of QSM and BSP on LogP

We now sketch our results for emulating BSP, QSM and s-QSM on LogP. Our emulation is randomized, and is work-preserving with polylog slowdown. In the next subsection, we describe a slightly more complex randomized emulation that uses sorting (with sampling) and which reduces the slowdown by slightly less than a logarithmic factor.

Fact 3.1 (Karp et al. [18]). *The following two problems can be computed in time $O\left(l\left\lceil \frac{\log p}{\log(l/g)} \right\rceil\right)$ on p processors* under the LogP model.

1. *Barrier synchronization on the p processors.*
2. *The sum of p values, stored one per processor.*

We will denote the above time to compute barrier synchronization and the sum of p values on the p-processor LogP by $B(p)$.

Theorem 3.2. *Suppose we are given an algorithm to route an h-relation on a p-processor LogP while satisfying the capacity constraint in time $O(g(h + H(p)) + l)$, when the value of h is known in advance. (Here, $H(p)$ is some given function of p.) Then,*

1. *There is a work-preserving emulation of a p-processor QSM on LogP with slowdown $O\big(g \log p + \log^2 p + (H(p) + B(p))\frac{\log p}{\log \log p}\big)$ whp in p.*
2. *There is a work-preserving emulation of a p-processor s-QSM and BSP on LogP with slowdown $O\big(\log^2 p + (H(p) + B(p))\frac{\log p}{\log \log p}\big)$ whp in p.*

Proof. We first describe the emulation algorithm, and then prove that it has the stated performance.

Algorithm for Emulation on LogP:

I. For the QSM emulation we map the QSM (or s-QSM) processors uniformly among the LogP processors, and we hash the QSM (or s-QSM) memory on the LogP processors so that each shared-memory location is equally likely to be assigned to any of the LogP components. For the BSP emulation we map the BSP processors uniformly among the LogP processors and the associated portions of the distributed memory to the LogP processors.

II. We route the messages to destination LogP processors for each phase or superstep while satisfying the capacity constraint as follows:
   1. Determine a good upper bound on the value of h.
   2. Route the h relation while satisfying the capacity constraint in $O(g(h+H(p))+l)$ time.
   3. Execute a barrier synchronization on the LogP processors in $O(B(p))$ time.

To complete the description of the algorithm, we provide in Fig. 1 a method for performing step II.1 in the above algorithm. To estimate h, the maximum number of messages sent or received by any processor, the algorithm must estimate the maximum number of messages received by any processor (note that the maximum number of sent messages by any processor, *maxsend*, is already known). The algorithm does this by selecting a small random subset of the messages to be sent and determining their destinations. The size of this subset is gradually increased until either a good upper bound on the maximum number of messages to be received by any processor is obtained or this value is determined to be less than *maxsend*.

Claim 3.3. *The algorithm for Step* II.1 *runs in time* $O(g\log^2 p + (H(p)+B(p))(\log p)/\log\log p)$ *whp, and whp it returns a value for* h *that is* (i) *an upper bound on the correct value of* h, *and* (ii) *within a factor of* 2 *of the correct value of* h.

Proof. The correctness of the algorithm follows from the following observations, which can be derived using Chernoff bounds:

1. If $\mu \geqslant \log p$ after some iteration of the **repeat** loop, then whp, the LogP processor that receives μ messages in that iteration has at least $\mu/(2q)$ messages being sent to it in that phase/superstep, and no LogP processor has more than $2\mu/q$ messages sent to it in that phase/superstep.
2. If $\mu < \log p$ at the end of an iteration in which $q \geqslant (2\log p)/maxsend$ then whp the maximum number of messages received by any LogP processor in this phase/superstep is less than *maxsend*.
3. In each iteration, whp the total number of messages sent does not exceed the value used for h in that iteration, hence the number of messages sent or received by any processor in that iteration does not exceed the value used for h.

For the time taken by the algorithm we note that $maxsend \geqslant m/p$, hence the **while** loop is executed $O(\log p/\log\log p)$ times. Each iteration takes time $O(g\max(\mu, maxsend\, q) + gH(p) + l)$ whp to route the h-relation, and time $O(B(p))$ to compute μ and perform a barrier synchronization. Hence each iteration takes time $O(g(\mu + maxsend\, q + H(p) + B(p)))$ since $l < B(p)$. Since the **while** loop terminates when $\mu \geqslant \log p$ or $maxsend\, q \geqslant 2\log p$, and q is increased by a factor of $\log p$ in each iteration, the overall time taken by the algorithm is $O(g\log^2 p + g(\log p/\log\log p)(H(p)+B(p)))$. □

Finally, to complete the proof of Theorem 3.2 we need to show that the emulation algorithm is work-preserving for each of the three models. Let $\tau = \log^2 p + (H(p)+B(p))(\log p)/\log\log p$.

```
maxsend := maximum number of messages to be sent by any LogP processor
m := total number of messages to be sent by all LogP processors
q := 1/m
μ := 1
repeat
    pfor each processor do
        q := q · log p;
        Select each message with probability q and send selected messages to
            destination with h = μ · log p;
        μ := max. number of messages received by any processor;
    rofp
until q ≥ (2 log p)/maxsend or μ ≥ log p
h := max(2μ/q, maxsend)
```

Fig. 1. Algorithm for Step II.1 of the algorithm for emulation on LogP.

If $p' \leqslant p/\tau$ then the time taken by the emulation algorithm to execute steps II.1 and II.3 is $O(g\tau)$, and hence the work performed in executing these two steps is $O(g\tau p') = O(gp)$. Since any phase or superstep of the emulated machine must perform work $\geqslant gp$, steps II.1 and II.3 of the emulation algorithm are executed in a work-preserving manner on a LogP with p' or fewer processors.

For step II.2, we consider each emulated model in turn. For the BSP we note that if we map the p BSP processors evenly among p' LogP processors, where $p' \leqslant p/\tau$, then a BSP superstep that takes time $c + gh + L$ will be emulated in time $O((p/p')(c + gh) + l)$ on a LogP with p' processors and hence is work-preserving. (We assume that $l \leqslant L$ since L includes the cost of synchronization.)

Next consider a phase on a p processor s-QSM in which h is the larger of (a) the maximum number of reads/writes by a processor and (b) the maximum queue-length at a memory location. If we hash the shared memory of the QSM on the distributed memory of a p'-processor LogP, and map the p s-QSM processors evenly among the p' LogP processors, then by the probabilistic analysis in [12], the number of messages sent or received by any of the p' LogP processors is $O(h(p/p'))$ whp in p, if $p' \leqslant p/\log p$. Hence the memory accesses can be performed in time $T = O(gh(p/p'))$ whp in p, once the value of h is determined. This is work-preserving since $Tp' = O(ghp)$.

Similarly, we can obtain the desired result for QSM by using the result in [12] that the mapping of QSM on a distributed memory machine results in the number of messages sent or received by any of the p' LogP processors being $O(h(p/p'))$ whp in p, if $p' \leqslant p/g \log p$. □

Corollary 3.4 (to Theorem 3.2). 1. *There is a work-preserving emulation of a p-processor QSM on LogP with slowdown* $O\left(g \log p + \log^4 p + \frac{l/g}{\log(l/g)} \frac{\log^2 p}{\log \log p}\right)$ *whp in p.*

2. *There is a work-preserving emulation of a p-processor s-QSM and BSP on LogP with slowdown* $O\left(\log^4 p + \frac{l/g}{\log(l/g)} \frac{\log^2 p}{\log \log p}\right)$ *whp in p.*

Proof. The corollary follows from Theorem 3.2 using the algorithm in [18] for barrier synchronization on p-processor LogP that runs in time $O\left(l \left\lceil \frac{\log p}{\log(l/g)} \right\rceil\right)$, and the algorithm in [1] for routing an h-relation on a p-processor LogP in $O(g(h + \log^3 p \log \log p) + l)$ whp in p. □

3.1.1. A faster emulation of BSP and QSM on LogP

For completeness, we describe a faster method for Step II.1 of the emulation algorithm given in the previous section. Since the algorithm given in this section uses sorting, it is not quite as simple to implement as the algorithm for Step II.1 given in Fig. 1, although it is simpler to describe and analyze.

Claim 3.5. *The algorithm given in Fig. 2 for Step* II.1 *determines an upper bound on the value of h whp in time* $O(gh + l \log p)$. *If* $h \geqslant \log^2 p$ *then the algorithm determines the correct value of h to within a constant factor whp.*

Proof. The result follows from the $O((gr + l)\log p)$ running time of the AKS sorting algorithm on the LogP [2,3], when rp keys in the range $[1..p]$ are distributed evenly across the p processors. (If the keys are not evenly distributed across the processors, they can be distributed evenly at an additional cost of $O(gh + l)$ time, where h is the maximum number of keys at any processor.)

The number of elements selected in step 3 is $m/\log p$ whp, where m is the total number of messages to be sent. Hence the number of elements to be sorted is $(m/(p \log p))p$, which is $O((s/\log p)p)$. Hence the time needed to execute step 4 is $O(gs + l \log p)$ whp. The remaining steps can be performed within this time bound in a straightforward manner.

Let m_i be the number of messages to be received by processor P_i. In step 3 of the algorithm in Fig. 2, for each processor P_i for which $m_i = \Omega(\log^2 p)$, $\theta(m_i/\log p)$ messages are selected whp (by a Chernoff bound). Hence (again by a Chernoff bound) it follows that the upper bound computed in step 6 for processor P_i is equal to m_i to within a constant factor, and hence the overall upper bound computed in step 7 is correct to within a constant factor. If no processor is the destination of more than $\log^2 p$ messages, then clearly the upper bound computed in step 7 is correct (although it may not be tight). □

Theorem 3.6. 1. *There is a work-preserving emulation of a p-processor QSM on LogP with slowdown* $O(\log^3 p \log \log p + (g + (l/g))\log p)$ *whp in p.*

2. *There is a work-preserving emulation of p-processor s-QSM and BSP on LogP with slowdown* $O(\log^3 p \log \log p + (l/g)\log p)$ *whp in P.*

3.2. Emulation of LogP on BSP

If a LogP program is *non-stalling* then it can be emulated in a work-preserving manner on BSP with slowdown $O(L/l)$ by dividing the LogP computation into blocks of computations of length l, and emulating each block in two BSP supersteps of time L each. This emulation is presented in [3] as an emulation where both the time and work increases by a factor of L/l. In the following theorem we pin down some of the details of

1. Compute $s :=$ maximum number of messages to be sent by any processor.
2. $q := 1/(\log p)$
3. **pfor** each processor **do** select each message with probability q **rofp**
4. Sort the selected messages by destination processor ID (in $O(g \cdot s + l \log p)$ time).
5. Compute the number of samples n_i destined for the ith LogP processor, for each i, by computing prefix sums on the sorted array (in time $O(l\lceil \frac{\log p}{\log(l/g)} \rceil)$).
6. **pfor** each processor i **do** compute an upper bound on the number of messages to be received as $r_i := (n_i + 1) \cdot \log p$ **rofp**
7. $h := \max(\log^2 p, s, \max_i r_i)$

Fig. 2. Faster algorithm for Step II.1 of algorithm for emulation on LogP.

the emulation not specified in [3], and we also make the emulation work-preserving.

Theorem 3.7. *There is a deterministic work-preserving emulation of a p-processor non-stalling LogP on BSP with slowdown $O(L/l)$.*

Proof. We map the LogP processors evenly among the BSP processors. Each BSP processor then emulates the L/l LogP processors assigned to it as follows.

- Divide the LogP computation into blocks of computation of length l.
- Emulate the steps performed by each LogP processor in this block of computation.

In the LogP cost measure the time taken is computed assuming that each message takes exactly l units of time to reach its destination. We show that we can perform the BSP computation in accordance with this measure. Since each block of LogP computation is of length l, messages sent within a block will arrive at their destination in the next block. In the BSP emulation each BSP processor tags each message sent with the LogP step in which it was sent. At the start of emulating a LogP block the BSP processor examines the messages received from the previous block, and sorts them by their tags in $O(L)$ time using integer sort. It then processes the received messages in the sorted order. Hence the BSP emulation executes the LogP steps at each processor in the same order as the execution under which LogP running time was measured.

Let us compute the time cost of emulating one block of LogP computation on a $(l/L)p$-processor BSP. The total amount of local computation at each BSP processor is $\leqslant (L/l)l = L$. Each BSP processor sends $\leqslant (L/l)(l/g) = L/g$ messages to other processors. Since the LogP computation is non-stalling, each BSP processor receives at most $(L/l)(l/g) = L/g$ messages. Hence the time cost of this computation on the BSP is $O(L)$ and the work is $O(pL(l/L)) = O(pl)$. Hence each block of LogP computation is emulated on the BSP in a work-preserving manner with slowdown $O(L/l)$. □

The analysis in [3] erroneously states that the L/l performance bound holds for stalling LogP computations. We now show a simple example of a stalling LogP computation whose execution time squares when emulated in the above manner on the BSP.

The LogP computation is shown in Fig. 3.

The following claim shows that this computation cannot be mapped on to the BSP with constant slowdown.

Claim 3.8. *The LogP computation shown in Fig.* 3 *takes time $O(rl + gq)$. When mapped on to the BSP this computation takes time $\Omega(r(L + gq))$.*

Proof. We note the following about the computation in Fig. 3:

(i) At time $(i-1)l + g$, all processors in the ith group send a message to processor P_i, $1 \leqslant i \leqslant r$. This is a stalling send if $q > l/g$. Processor P_i then receives all messages at time $il + gq$.
(ii) The computation terminates at time $rl + gq$ when P_r receives all messages sent to it.

On a BSP we note that the computation in Fig. 3 must be executed in r phases (or supersteps) since a processor in groups 2 to r can send its message(s) only after it has received a message from a processor in group $(i-1)$. In a BSP computation any send based on a message received in the current phase cannot be executed in the same phase. Hence the computation requires r phases. In each phase there are q messages received by some processor (by processor P_i in phase i). Hence this computation takes time $\Omega(r(L + gq))$, which is $\Omega(rL + rgq)$ time. Thus the slowdown of this emulation is $\Omega\left(\frac{rL+rgq}{rl+gq}\right)$.

Note that the parameter o does not appear in the cost of the LogP computation since there is no local computation in this program. □

The above claim leads to the following theorem.

Configuration. LogP with $p = r \cdot (q+1)$ processors, grouped into
r groups of q processors, and one group of r processors.
For $1 \le i \le r$, the jth processor in the ith group is denoted by $p_{i,j}$.
The processors in the group with r processors are labeled P_j, $1 \le j \le r$.
// initial step:
pfor $1 \le j \le r$ processor $p_{1,j}$ executes the following two steps in sequence:
a. send a message to processor $p_{2,j}$
b. send a message to processor P_1.
rofp
pfor $2 \le i \le r$
pfor $1 \le j \le q$ **do**
if processor $p_{i,j}$ receives a message from processor $p_{(i-1),j}$ **then** it executes the following two steps in sequence:
a. sends a message to processor $p_{(i+1),j}$ (if $i \ne r$)
b. sends a message to processor P_i.
rofp
rofp

Fig. 3. A stalling LogP computation whose execution time can increase by more than L/l when emulated on a BSP with same number of processors.

Theorem 3.9. *Consider the deterministic emulation of non-stalling LogP on BSP.*

a. *Any deterministic step-by-step emulation of LogP on BSP can have arbitrarily large slowdown.*

b. *There is no deterministic step-by-step emulation of stalling LogP on BSP that is work-preserving.*

Proof. For part a, consider the computation given in the proof of Claim 3.8. If r is any non-constant function with $rl = o(gq)$ and $l \leqslant L$, then the slowdown of this emulation is $\Theta(r)$ and is not dependent on the ratio L/l. We can assume that $l \leqslant L$ since l accounts only for latency while L accounts for both latency and global synchronization cost. Thus to obtain a slowdown of S, we choose, e.g., $r = S$ and $q = Sl^2/g$. Since all that was assumed of the emulation is that it is step-by-step, i.e., each step of a LogP processor is executed by some BSP processor in the same manner as prescribed by the LogP computation, the result follows.

For part b, suppose there is a work-preserving emulation of stalling LogP on BSP with slowdown τ. Then consider the emulation on BSP of the LogP computation in Fig. 3 with $r = \omega(\tau)$ and with $rl = o(gq)$ and $l \leqslant L$. Then the work performed by the LogP computation is $\Theta(gqp)$ while the work performed by the emulating BSP computation is $\Theta(rgqp/\tau)$, which is $\omega(gqp)$. Hence the emulation is not work-preserving. □

3.3. Emulation of LogP on QSM

In this section we consider the emulation of LogP on QSM. For this emulation we assume that the input is distributed across the local memories of the QSM processors in order to conform to the input distribution for the LogP computation. 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. We prefer the former method, since it is meaningful to evaluate the computation time on a QSM in which the input is distributed across the local processors of the QSM—as, for instance, in an intermediate stage of the large computation, where values already reside within the local memories of the QSM, and where the result of a computation executed on these values will be used locally by these processors later in the computation.

As in the case of the emulations seen earlier, we map the LogP processors uniformly among the QSM processors in the emulating machine, and we assign to the local memory of each QSM processor the input values that were assigned to the LogP processors emulated by it. We can then emulate LogP on a QSM or s-QSM with slowdown $O\left(\left\lceil \frac{g \log p}{l} \right\rceil\right)$ whp as follows:

I. Divide the LogP computation into blocks of size l.

II. Emulate each block in $O\left(\left\lceil \frac{g \log p}{l} \right\rceil\right)$ time in two QSM phases as follows, using the shared memory of the QSM (or s-QSM) only to realize the h-relation routing performed by the LogP in each block of computation.

Each QSM (or s-QSM) processor copies into its private memory the messages that were sent in the current superstep to the local memory of the LogP processors mapped to it using the method of [12] to emulate BSP on QSM, which we summarize below.

1. Compute M, the total number of messages to be sent by all processors in this phase. Use the shared memory to estimate the number of messages being sent to each group of $\log^3 M$ destination processors as follows:

Sample the messages with probability $1/\log^3 M$, *sort* the sample, thereby obtaining the counts of the number of sample elements being sent to each group of $\log^3 M$ destination processors; then estimate an upper bound on the number being sent to the ith group as $c \max(k_i, 1)\log^3 M$, where k_i is the number of sample elements being sent to the ith group, and c is a suitable constant.
2. Processors that need to send a message to a processor in a given group use a *queue-read* to determine the estimate on the number of messages being sent to the ith group and then place their messages in an array of this size using a *multiple compaction* algorithm.
3. Perform a stable sort (by destination processor ID) on the elements being sent to a given group, thereby grouping together the elements being sent to each processor.
4. Finally each processor reads the elements being sent to it from the grouping performed in the above step.

Theorem 3.10. *A non-stalling LogP computation can be emulated on the QSM or s-QSM in a work-preserving manner whp with slowdown* $O\left(\left\lceil \frac{g \log p}{l} \right\rceil\right)$, *assuming that the input to the LogP computation is distributed uniformly among the local memories of the QSM processors.*

3.4. Discussion

We have presented work-preserving emulations between LogP and the other three models—QSM, s-QSM and BSP. (Work-preserving emulations between QSM, s-QSM and BSP were presented earlier in [12], see also [20].) Overall these results indicate that these models are essentially interchangeable for the purpose of algorithm design since each can emulate the others in a work-preserving manner with only a small slowdown.

A couple of features about our emulations are worth further discussion.

1. *Stalling versus non-stalling LogP.* The one mis-match we have in our emulations is between stalling and non-stalling LogP. Here we showed that there is a simple, deterministic, work-preserving emulation of non-stalling LogP on BSP, but there is no deterministic step-by-step emulation of stalling LogP on BSP that is work-preserving. This is in contrast to an incorrect inference made in [3] that LogP is essentially equivalent to BSP.

   Our counterexample that shows the negative result on emulating stalling LogP on BSP indicates that the stalling LogP gives processors an automatic scheduling feature. This does not appear to mirror the behavior of real parallel systems, and seems to indicate that the modeling of stalling should be done more carefully in order to be reflective of real machines.
2. *Emulating message-passing on shared-memory and vice versa.* The algorithms we have given for emulating a distributed memory model, LogP or BSP, on shared-memory are rather involved due to the use of sorting and multiple compaction. On the other hand the shared-memory models, QSM and s-QSM, have simple emulations on BSP and LogP.

   The reason for the complications in our BSP/LogP emulation on shared-memory is the need to map a message-passing interface on to a shared-memory environment. Since both message-passing and shared-memory are widely used in practice, we suggest that a high-level general-purpose model should be one that maps on to both in a simple way. We have shown that QSM and s-QSM give us this feature. Additionally, they have a smaller number of parameters, and do not have to keep track of the layout of data across shared memory.

Since the QSM and s-QSM have fewer parameters than the BSP or LogP, and they are shared-memory, for the rest of this paper we use these two models as our basic models. We analyze algorithms using the s-QSM cost metric, as the symmetry between processor requests and memory accesses in the s-QSM model leads to simpler analyses, and also helps achieve a clean separation between the cost for local computation and cost for communication. Since any s-QSM algorithm runs within the same time and work bounds on the QSM, our upper bounds are valid on both models.

4. Basic QSM algorithms

To support using QSM or s-QSM for designing general-purpose parallel algorithms, we develop a suitable cost metric for such algorithms. We then present simple QSM algorithms for prefix sums, sample sort and list ranking; all three algorithms are adaptations of well-known PRAM algorithms suitably modified to optimize for our cost measure. In the next section we present some experimental analysis and data on simulations using parallel code we wrote for these algorithms.

4.1. Cost measures for a QSM computation

Our cost metric for a QSM algorithm seeks to

1. minimize the work performed by the algorithm,
2. minimize the number of phases in the algorithm, and
3. maximize parallelism, subject to the requirements (1) and (2).

The *work* $w(n)$ of a parallel algorithm for a given problem is the processor-time product for inputs of size

n. There are two general lower bounds for the work performed by a QSM algorithm: First, the work is at least as large as the best sequential running time of any algorithm for the problem; and second, if the input is in shared-memory and the output is to be written into shared-memory, the work is at least gn, where n is the size of the input [12].

The *maximum parallelism* of an algorithm with work $w(n)$ is the smallest running time $t(n)$ achievable by the algorithm while performing $w(n)$ work. As for a PRAM algorithm, maximum parallelism is a meaningful measure for a QSM or s-QSM algorithm, since these algorithms can always be slowed down (by using a smaller number of processors) while performing the same work [12].

The motivation for the new second metric on minimizing number of phases is the following. One major simplification made by QSM is that it does not incorporate an explicit charge for latency or the synchronization cost at the end of each phase. The total time spent on synchronizations is proportional to the number of phases in the QSM algorithm. Hence minimizing the number of phases in a QSM algorithm minimizes the hidden overhead due to synchronization. In particular it is desirable to obtain an algorithm for which the number of phases is independent of the input size n as n becomes large. All of the algorithms we present have this feature.

Related work on minimizing the number of phases (or supersteps) using the notion of *rounds* is reported in [13] for sorting and in [5] for graph problems. Several lower bounds for the number of rounds needed for basic problems on the QSM and BSP are presented in [19].

A 'round' is a phase or superstep that performs linear work ($O(gn/p)$ time on s-QSM, and $O(gn/p + L)$ time on BSP). Any linear-work algorithm must compute in rounds, hence this is a useful measure for lower bounds on the number of phases (or supersteps) needed for a given problem. On the other hand, a computation that proceeds in rounds need not lead to a linear work algorithm if the number of rounds in the algorithm is non-constant. In fact, all of the algorithms presented in [5] perform superlinear work. The algorithm in [13] performs superlinear communication when the number of processors is large.

In contrast to the cost metric that uses the notion of rounds, in this paper we seek algorithms that perform optimal work and communication and additionally compute in a small number of phases.

Our metric does not consider providing good performance for tiny problem sizes to be a primary consideration. This is because our emphasis is on simple algorithms that can be used in practice. This encourages us to focus on algorithms for the case when the input size is, say, at least quadratic in the number of processors, since the input sizes for which we would use a parallel machine for the problems we study would normally be at least as large, if not larger. The pay-off we get for considering this moderate level of parallelism is that our algorithms are quite simple. Our algorithm for sample sort is inspired by, and fits this frame-work. However, our algorithms for prefix sums and list ranking achieve a much higher level of parallelism. In fact, we prove that our prefix sums algorithm is optimal for the complete range of values for parameters that lead to linear-work algorithms, and it differs from earlier prefix sums algorithms for the case when p is very close to $n/\log n$, i.e., for the highly parallel case. But it should be noted that this is achieved with a simple algorithm that is easily implementable in practice. In short, our goal in developing all three algorithms was to obtain effective algorithms for moderate levels of parallelism. Discussion of simulation results in the next section support our belief that we can simplify QSM algorithms without hurting performance for practical problems.

As noted in the section describing our emulation of LogP on QSM, it is meaningful to consider computations in which the input and output remain distributed uniformly across the local memories of the QSM processors. This would correspond, for instance, to a situation where the computation under consideration is part of a more complex computation. In such a case a QSM processor would not need to write back the computed values into shared-memory if these values will be used only by this processor in later computations. Our simple prefix sums algorithm (given in Fig. 5) has an improved performance under this assumption of distributed input and output. In the other algorithms we present, the savings gained by this configuration is no more than a constant factor. However, we will come back to this point in the next section where we present experimental results. There we pin down the constant factors for the running time, based on the distributed input environment that we used to run our algorithms.

4.2. Prefix sums algorithm

The prefix sums algorithm is given in Fig. 4. The processors perform a corresponding sequence of 'expansion' steps in which the correct prefix sum value is computed for each position once the correct offset is supplied to it.

We analyze its performance in the following claim, and in the next claim, we show that its performance is optimal whenever $p \leqslant n/\log n$. (Note the parameter ϕ in this algorithm, which distinguishes it from all other known algorithms for prefix sums. This parameter becomes relevant only when the value of p is close to $n/\log n$.)

Claim 4.1. *The algorithm in Fig.* 4 *computes the prefix sums of array* $A[1..n]$, *and runs in* $O(gn/p)$ *time* (*and*

Input. Array $A[1..n]$ to a p-processor QSM.
// Preprocess to reduce size to p:
pfor $1 \le i \le p$ **do**
  processor p_i reads the ith block of n/p elements from array
    A, computes local prefix sums, and stores the sum in $B[i]$.
rofp
// Main loop
Let $f(n) = \frac{n}{p \log n}$
$\phi := \frac{\log n}{\log f(n)}$
$r := (n/p) \cdot (1/\phi)$
$k := p$
repeat
  pfor $1 \le i \le \lceil k/r \rceil$ **do**
    processor i reads the ith block of $\lceil r \rceil$ elements from array B,
      computes local prefix sums, and stores the sum in $B[i]$
    $k := \lceil k/r \rceil$
  rofp
until $k = 1$
The processors perform a corresponding sequence of 'expansion' steps in which the correct
  prefix sum value is computed for each position once the correct offset is supplied to it.

Fig. 4. Prefix sums algorithm.

hence $O(gn)$ work) and $O(\phi)$ phases when $p \leqslant n/\log n$ on QSM and s-QSM.

Proof. Let t be the number of iterations of the **repeat** loop. Then $t = \log p/\log r$, i.e., $t = \frac{\log p}{\log(n/p) - \log \phi}$.

We have $\log(n/p) = \log\log n + \log f(n)$ and $\log \phi = \log\log n - \log\log f(n)$, hence $t = \Theta\left(\frac{\log p}{\log\log n + \log f(n) - \log\log n + \log\log f(n)}\right) = \Theta\left(\frac{\log p}{\log f(n)}\right) = O(\phi)$.

The algorithm performs each iteration of the **repeat** loop in one phase, hence the number of phases in the algorithm is $2t + 1$, which is $O(\phi)$.

The time taken by each iteration of the **repeat** loop is $O(gr)$, hence the overall running time of the **repeat** loop is $O(tgr)$, which is $O(g\phi(n/p)(1/\phi)) = O(g(n/p))$. The first **pfor** loop takes $O(gn/p)$ time, and hence the overall running time of the algorithm is $O(gn/p)$, and the work performed by the algorithm is $O(gn)$. When $r = O(1)$, the time taken by the algorithm is $O(g \log n)$, hence the algorithm performs $O(gn)$ work as long as $p = O(n/\log n)$. □

Note that this algorithm runs in a constant number of rounds if $p = O(n^c)$, for some constant $c < 1$. In the following claim we show that this algorithm uses an optimal number of phases over the entire range of values for p (i.e., $1 \leqslant p \leqslant n/\log n$) for which a prefix-sums algorithm with $O(gn)$ work is possible.

Claim 4.2. *The algorithm in Fig. 4 computes the prefix sums of array $A[1..n]$ on s-QSM with optimal $O(gn)$ work and in optimal number of phases whenever $p \leqslant n/\log n$.*

Proof. The bound on the work performed is $O(gn)$ when $p \leqslant n/\log n$. This is seen to be the best possible (for either the QSM or the s-QSM) through a simple lower bound given in [12].

We need to show that the upper bound given in Claim 4.1 for the number of phases is optimal for s-QSM. If $n/p \geqslant \log^{1+\varepsilon} n$ then the optimality follows from a lower bound of $\Omega(\log n/\log(n/p))$ given in [19] for the number of phases needed to compute the OR of n bits on s-QSM when constrained to perform $O(g(n/p))$ communication per phase. Note that $\log n/\log f(n) = \Theta(\log n/\log(n/p))$ when $n/p \geqslant \log^{1+\varepsilon} n$.

We now strengthen the above lower bound to show that for computing parity and prefix sums, the number of phases needed is $\Omega\left(\frac{\log n}{\log f(n)}\right)$, where $f(n) = n/(p \log n)$. In [19] it is shown that if an s-QSM computes the parity of n bits in l phases while performing gT communication, then

$$\prod_{j=1}^{l} (6\tau_j) \geqslant n,$$

where $g\tau_j$ is the time taken for communication in the jth phase. Since $\sum_{j=1}^{l} \tau_j = T$, the above product is maximized when the τ_j's are all equal, and equal to T/l. Hence we have $(6T/l)^l \geqslant n$, i.e., $l = \Omega\left(\frac{\log n}{\log T - \log l}\right)$.

We have $T = n/p = \log n f(n)$, hence $l = \Omega\left(\frac{\log n}{\log\log n + \log f(n) - \log l}\right)$. Solving for l we find that $l = \Omega\left(\frac{\log n}{\log f(n)}\right)$, giving us the desired matching lower bound (since computing prefix sums on an n-array is at least as hard as computing parity of n bits). □

Broadcasting. We note that the above algorithm can be run in reverse to broadcast a value to p processors to

obtain the same bounds if $O(gn/p)$ time is allowed per phase.

Finally we note that the QSM algorithm for prefix sums is extremely simple when $p \leqslant \sqrt{n}$, which is the situation that typically arises in practice. This algorithm is shown in Fig. 5. It is straightforward to see that this algorithm computes the result in $O(gn/p)$ time and two phases. The process of writing and then reading locations in the array $S[i,j]$ is a simple method of broadcasting p values to all processors.

Theorem 4.3. *The simple prefix sums algorithm runs in $O(gn/p)$ time and in two phases when $p \leqslant \sqrt{n}$.*

If the input and output are to be distributed uniformly among the local memories of the processors, then the simple prefix sums algorithm runs in $O(gp)$ time when $p \leqslant \sqrt{n}$.

4.3. Sample sort algorithm

Fig. 6 shows the QSM sample sort algorithm. We assume that $p \leqslant \sqrt{\frac{n}{\log n}}$; in other words, there is a significant amount of work for each processor to do.

This algorithm is based on the standard sample sort algorithm that uses 'over-sampling' and then picks pivots evenly from the chosen samples arranged in sorted order [7,8,10,11,16,22,23].

Theorem 4.4. *The algorithm in Fig. 6 sorts the input array while performing optimal work $(O(gn + n\log n))$, optimal communication $(O(gn))$, in $O(1)$ phases whp when the number of processors $p = O\left(\sqrt{\frac{n}{\log n}}\right)$.*

Input. Array $A[1..n]$ to a p-processor QSM, $p \leq \sqrt{n}$.
pfor $1 \leq i \leq p$ **do**
  processor p_i reads the ith block of n/p elements from array A,
    computes local prefix sums, and stores the sum in locations $S[i,j],\ i+1 \leq j \leq p$
rofp
pfor $1 \leq i \leq p$ **do**
  processor p_i reads all entries in subarray $S[1..i-1,i]$, computes
    the sum of the elements in the subarray, adds this offset to its
    local prefix sums, and stores the computed prefix sums in locations
    $(i-1)\cdot(n/p)+1$ through $i \cdot n/p$ in output array B
rofp

Fig. 5. Simple prefix sums algorithm for $p \leqslant \sqrt{n}$.

Input. Array $A[1..n]$ to a p-processor QSM, $p \leq \sqrt{n/\log n}$.

1. **pfor** $1 \leq i \leq p$ **do**
   a. The ith processor p_i reads the ith block of n/p elements from the input array;
   b. p_i selects $c\log n$ random elements from its block of elements and writes p copies of these selected elements in locations $S[1..c \cdot p\log n, i]$

   rofp
2. **pfor** $1 \leq i \leq p$ processor p_i performs the following steps
   a. Processor p_i reads the values of the samples from locations $S[i, j \cdot c\log n + i]$, $0 \leq j \leq (p-1)$
   b. p_i sorts the $cp\log n$ samples, and picks every $c\log n$th element as a pivot;
   c. p_i groups its local n/p elements from the input array into groups depending on the bucket into which they fall with respect to the pivots.
   d. For $1 \leq j \leq p$
      write back the elements in the jth bucket into a block in an array meant for all elements in the jth bucket. (This requires a global prefix sums calculation to determine the location of the block within the array in which to write the elements in bucket j from the ith processor.
      The same computation gives the locations needed for the writes in step 3.)

   rofp
3. **pfor** $1 \leq i \leq p$ **do**
   Processor p_i reads the elements in the ith bucket, sorts them
     and writes the sorted values in the corresponding positions in the output array.

   rofp

Fig. 6. Sample sort algorithm.

Proof. The algorithm selects $cp \log n$ random samples in step 1. In step 2 these samples are read by each processor, then sorted, and $p-1$ evenly spaced samples are chosen as the 'pivots'. The pivots divide the input values into p *buckets*, where the ith bucket consists of elements whose values lie between the $(i-1)$st pivot and the ith pivot in sorted order (assuming the 0th pivot has value $-\infty$ and the pth pivot has value ∞. The elements in the ith bucket are locally sorted by the processor p_i and then written in sorted order in the output array. Hence the algorithm correctly sorts the input array.

We now analyze the running time of the algorithm with p processors, $p \leqslant \sqrt{n/\log n}$. Steps 1a and 2d take $O(gn/p)$ time, and steps 1b and 2a take time $O(gp \log(n/p)) = O(gn/p)$, since $p \leqslant \sqrt{n/\log n}$. Step 2b takes time $O(p \log n \log(p \log n)) = O((n/p)\log(n/p))$, and step 2c takes time $O((n/p)\log p)$ if binary search on the pivots is used to assign each element to its bucket. Step 3 takes time $O(B \log B + gB)$, where B is the size of the largest bucket.

We now obtain a bound on the size of the largest bucket B.

Consider the input elements arranged in sorted order in a sequence S. Consider an interval I of size $s = \alpha n/p$ on S, for a suitable constant $\alpha > 1$. In the following we obtain a high probability bound on the number of samples in any interval of size s.

Let $Y_{i,j}$, $1 \leqslant i \leqslant c \log n, 1 \leqslant j \leqslant p$, be a random variable that is 1 if the ith sample of the jth processor lies in I, and is zero otherwise.

$Pr[Y_{i,j} = 1] = s_j p/n$, for $1 \leqslant i \leqslant c \log n$, where s_j is the number of elements in I that are from processor p_j's block of n/p elements.

Let $Y = \sum_{i=1}^{c \log n} \sum_{j=1}^{p} Y_{i,j}$. Note that Y is the number of samples in I.

$E[Y] = c \log n \sum_{j=1}^{p} s_j(p/n) = (scp \log n)/n$

Hence $E[Y] = \alpha c \log n$.

By Hoeffding's inequality, $Pr[Y \leqslant k] \leqslant Pr[X \leqslant k]$, for $k < \alpha c \log n$, where X is the sum of $pc \log n$ 0-1 independent random variables, with probability of success equal to s/n for all of these random variables.

$E[X] = c\alpha \log n$.

By a Chernoff bound, $Pr(X \leqslant c \log n) \leqslant e^{-\frac{c(\alpha-1)^2 \ln n}{2\alpha \ln 2}} = n^{-c(\alpha-1)^2/(2\alpha \ln 2)}$, i.e., $Pr(Y \leqslant c \log n) \leqslant n^{-c(\alpha-1)^2/(2\alpha \ln 2)}$.

Let a_i be the position of the $ci \log n$th sample in the sorted sequence S, $1 \leqslant i \leqslant p-2$. Let B_i be the interval of size $\alpha n/p$ on sequence S starting at a_i, $1 \leqslant i \leqslant p-2$. Let B_0 be the interval of size $\alpha n/p$ starting at the first element of S and let B_{p-1} be the interval of size $\alpha n/p$ ending at the last element of S.

The probability that any of the intervals B_i, $0 \leqslant i \leqslant p-1$ has less than $c \log n$ samples is no more than $pn^{-c(\alpha-1)^2/(2\alpha \ln 2)}$, which is $O(1/n^r)$, $r > 0$, for a suitable choice of α and c. Hence whp every bucket has no more than $\alpha n/p$ elements.

Thus whp, step 3 takes time $O((n/p)\log n + gn/p)$, and thus the overall running time of the algorithm is $O(g(n/p) + (n \log n)/p)$, which is optimal.

There are 6 phases in the algorithm—one each for steps 1a, 1b, 2a, 2d, 3, and 4. □

4.4. List ranking algorithm

Fig. 7 summarizes the list ranking algorithm.

Theorem 4.5. *The List Ranking algorithm runs with optimal work and optimal communication ($O(gn/p)$ for both), and in $O(\log p)$ phases whp when the number of processors $p = O(n/\log n)$.*

Proof. We first consider the case when $p = o(n^{\varepsilon})$. Consider a given iteration of the **pfor** loop. Let r be the number of elements in a given processor P, and let $r = r_e + r_o$, where r_e denotes the number of elements at even distance, and r_o denotes the number at odd distance from the end of the current linked list.

Let X_e be a random variable denoting the number of elements at even distance from the end of the list in processor P that are eliminated in this iteration of the **pfor** loop. Let X_o be the corresponding random variable for elements at odd distance from the end of the linked list. The random variables X_e and X_o are binomially distributed r.v.'s with $E[X_e] = r_e/4$ and $E[X_o] = r_o/4$.

By a Chernoff bound,

$Pr(X_e \leqslant (1-\beta)r_e/4) \leqslant e^{-\beta^2 r_e/8}$ and $Pr(X_o \leqslant (1-\beta)r_o/4) \leqslant e^{-\beta^2 r_o/8}$.

Hence, since either r_e or r_o is at least $r/2$, with exponentially high probability in r, at least $(1-\beta)/8$ of the elements in P are eliminated in this iteration.

If $p = o(n^{\varepsilon})$, for any $\varepsilon > 0$, then $n/p^2 = \Omega(n^b)$, for some constant $b > 0$. Hence, in every iteration of the **pfor** loop, either at least $(1-\beta)/8$ of the elements are eliminated at each processor with exponentially high probability, or the number of elements remaining at the processor is $o(n/p)$. Hence, after $c \log p$ iterations, the number of elements remaining in the linked list is $\leqslant (1-\beta)/4)^{c \log p} n$ with exponentially high probability. With a suitable choice of c this number of elements remaining can be made $\leqslant n/p$.

By the above analysis the number of elements eliminated at any given processor is geometrically decreasing from iteration to iteration. Hence the total time for step 2 (and hence for step 4) is $O(gn/p)$. At the end of step 2 the number of elements is reduced to $O(n/p)$ (with exponentially high probability), hence the time for step 3 is $O(gn/p)$. Hence the overall running time of the algorithm is $O(gn/p)$. The number of phases is $O(\log p)$, since there is a constant number of phases in each iteration of step 2.

If $p = \Omega(n^{\varepsilon})$, we can use a standard analysis of randomized list ranking to show that all elements at a

Input. Successor array $S[1..n]$ to a p-processor QSM, $p \leq \sqrt{n/\log n}$.

1. Each processor reads a block of n/p of the input successor array.
2. **for** $c \log p$ iterations **do**
   pfor $1 \leq i \leq p$ **do**
   a. Processor p_i generates a random bit for each element in its local sublist.
   b. p_i 'eliminates' each local active element for which its random bit is a 0 and its successor random bit is a 1.
   c. p_i compacts its local sublist by removing the eliminated elements using an 'indirection' array.
   rofp
   rof
3. All processors send their current sublist to processor 0, which then ranks the current elements sequentially.
4. All processors perform a sequence of 'expansion' steps corresponding to step 2 in which the correct list rank is computed for each element once the correct offset is supplied to it.

Fig. 7. List ranking algorithm.

processor are eliminated in $O(\log n) = O(\log p)$ time whp. In this case, for a suitable choice of c, the length of the list is reduced to 1 at the end of step 2, and step 3 is not required (although one might still use step 3 for improved performance). □

5. Experimental results

We investigated the performance of the prefix sums, sample sort and list ranking algorithms on *Armadillo* [14], which is a simulated architecture with parameterizable configurations and cycle-by-cycle profiling and accuracy. In this section we describe the results we obtained by simulating our three algorithms on an eight-processor machine. Results for a 16-processor machine, as well as results of other experiments and conclusions derived from them can be found in [15]. A detailed description of the experimental set-up can be found there as well. (Those experiments were performed on a simulator in order to evaluate the effect of varying parameters of the parallel machine such as latency and overhead and the effectiveness of the QSM model and the BSP model in predicting performance of algorithms.)

The results of the experiments indicate that the QSM predictions come close to the observed values for fairly small problem sizes and that they become more accurate as problem sizes increase. We also found that the looseness of bounds obtained using standard techniques of algorithm analysis for non-oblivious algorithms and standard tools for analyzing randomized algorithms are often larger than the errors introduced by QSM's simplified network model. This was certainly the case for both sample sort and list ranking.

5.1. General comments

The graphs for the three algorithms are given in Figs. 8–10. Each of our graphs shows the measured results of running one of the three algorithms, and compares the measured communication time to the communication time predicted by QSM. As a comparison, we also plot the communication time for the same algorithm as would be predicted by the more detailed BSP model. We do not include predictions of the LogP model since they would be almost identical to the predictions of the BSP model for the three algorithms we consider.

Our analysis focuses on communication performance—excluding CPU time—for two reasons. First, all models examined here model CPU performance in the same way, so comparisons of predictions of CPU performance are not interesting. Second, exact CPU time calculations depend on low level parameters that are beyond the scope of the QSM and BSP models. However, for completeness the graphs also show the total measured time taken by the computation.

The architecture we simulated was that of a distributed-memory multiprocessor, and thus the input and the output was distributed uniformly across the processors. Hence in analyzing the algorithms we excluded the initial cost of reading the input from shared-memory and the final cost of writing the output into shared-memory. As discussed earlier, such an analysis is meaningful in the context of a shared-memory model since it would correspond, for instance, to a situation where the computation under consideration is part of a more complex computation and the input/output is available at the local memories of the appropriate processors. The algorithms were simulated on 4, 8 and 16 processors.

We plotted several computed and measured costs as listed below:

1. 'Communication' is the measured cost of the communication performed by the algorithm, measured in cycles.
2. 'QSM best-case' represents the ideal performance of each of the randomized algorithms. It uses the QSM

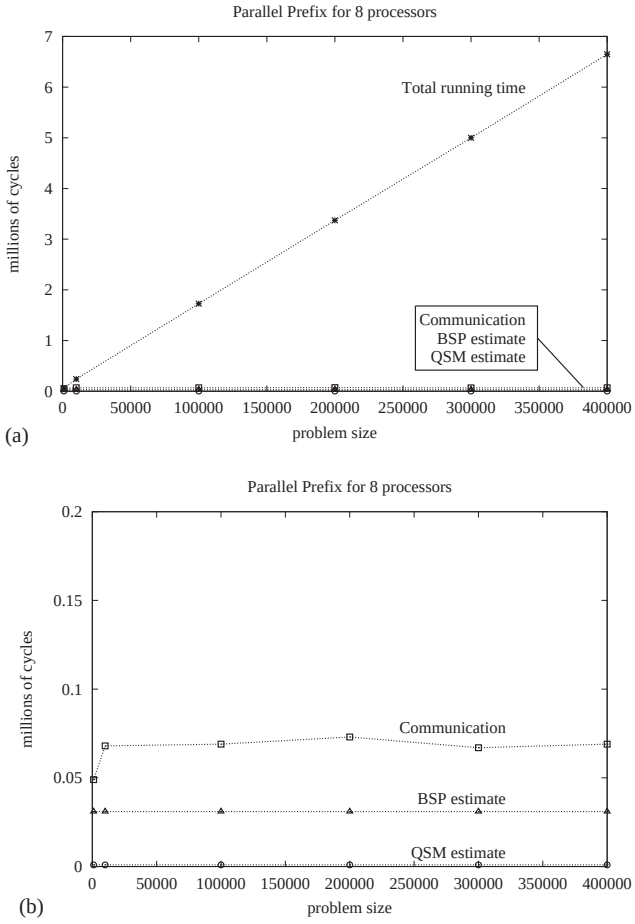

Fig. 8. Measured and predicted performance for the prefix sums algorithm on 8 processors. (a) Total running time and communication time. (b) Communication time.

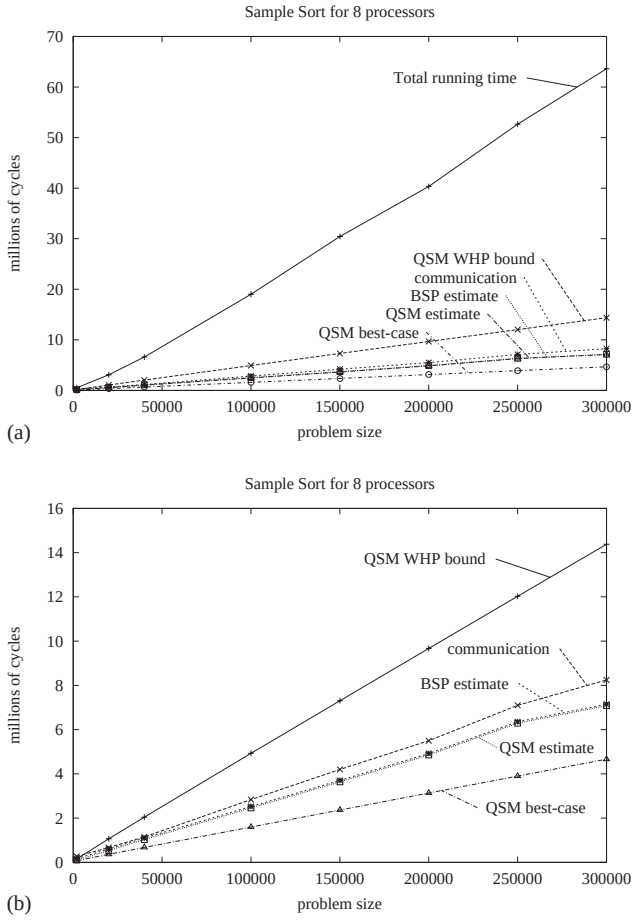

Fig. 9. Measured and predicted performance for the sample sort algorithm on 8 processors. (a) Total running time and communication time. (b) Communication time.

analysis but assumes no skew in the performance of the randomized steps.

3. 'QSM WHP bound' represents the performance of each of the randomized algorithms that we can guarantee with probability at least 0.9 using Chernoff bound analysis.
4. The 'QSM estimate' line is a plot of the measured maximum number of communication operations at any processor multiplied by the gap parameter. (Since none of the algorithms we implemented had queue contention at memory locations, this correctly measures the communication cost as modeled by QSM.) It accounts for the actual skew encountered during the runs. For the prefix sums algorithm the 'QSM estimate' line also gives 'QSM best case' since the algorithm is deterministic and oblivious. For the randomized algorithms, this line plots the QSM prediction without the inaccuracy that is incurred when working with loose analytical bounds on the amount of communication.
5. The 'BSP estimate' line is similar to 'QSM estimate', except that there is an additional term to account for the latency parameter.
6. 'Total running time' is the measured cost of the total running time of the algorithm, measured in cycles. We include this for completeness.

5.2. Discussion

For all three algorithms, we found that 'QSM estimate' tracks communication performance well when the input size is reasonable large. The input sizes for which we simulated the algorithms are fairly small due to the CPU-intensive computation of the step-by-step simulation performed by Armadillo. Modern parallel architectures typically give each processor many megabytes of memory, so problems of practical interest are likely to be even larger than those presented here.

As expected, the communication cost for the prefix sums algorithm is negligible compared to the total computation cost as n becomes large. QSM (and to a lesser extent BSP) both underestimate the communication cost by a large amount, but since the communication cost is very small anyway, this does not appear to be a significant factor. The possible cause for this

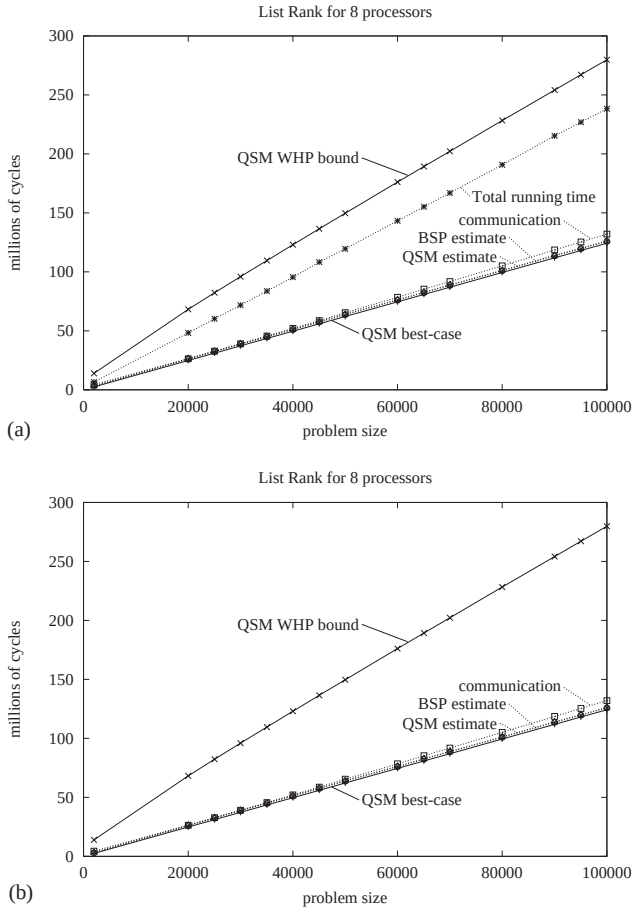

Fig. 10. Measured and predicted performance for the list ranking algorithm on 8 processors. (a) Total running time and communication time. (b) Communication time.

discrepancy between the predicted and measured communication costs is discussed in [15].

As expected, for both sample sort and list ranking the lines for 'QSM best-case' and 'QSM WHP bound' envelope the line for actual measured communication except for tiny problem sizes (when latency dominates the computation cost). For both algorithms the 'QSM estimate' line is quite close to the 'communication' line, indicating that QSM models communication quite effectively when an accurate bound is available for the number of memory accesses performed by the processors. For instance with 8 processors, 'QSM estimate' is within 20% of 'communication' for sample sort when the input size is larger than 40,000, and is within 5% of 'communication' for list ranking when input size is larger than 20,000. The 'BSP estimate' lines are very close to the 'QSM estimate' lines for both algorithms.

For both sample sort and list ranking the 'QSM WHP' line gives a very conservative bound, and lies significantly above the line for 'communication.' This is to be expected, since the 'communication' line represents the average of ten runs while the 'QSM WHP' line guarantees the bound for at least 90% of the runs. Further, the bounds were computed using Chernoff bounds, and hence are not tight. It should be noted that the fairly large gap between the 'communication' and the 'QSM WHP bound' lines is mainly due to the looseness of the bounds we obtained on the number of memory accesses performed by the randomized algorithms, and not due to inaccuracy in the QSM communication model. As noted above, the 'QSM estimate' line which gives the QSM prediction based on the measured number of memory accesses is quite close to the 'communication' line.

Overall these graphs show that QSM models communication quite effectively for these algorithms, for the range of input sizes that one would expect to see in practice. We also note that the additional level of detail in the BSP model has little impact on the ability to predict communication costs for the algorithms we studied, as compared to the QSM.

6. Conclusion

This paper has examined the use of QSM as a general-purpose model for parallel algorithm design. QSM is especially suited to be such a model because of the following.

1. It is shared-memory, which makes it convenient for the algorithm designer to use.
2. It has a small number of parameters (namely, p, the number of processors, and g the gap parameter).
3. We have presented simple work-preserving emulations of QSM on other popular models for parallel computation. Thus an algorithm designed on the QSM will map on to these other models effectively.

To facilitate using QSM for designing general-purpose parallel algorithms, we have developed a suitable cost metric for such algorithms and we have evaluated algorithms for some fundamental problems both analytically and experimentally against this metric. These results indicate that the QSM metric is quite accurate for problem sizes that arise in practice.

Appendix. Description of the experimental setup

The Armadillo multiprocessor simulator [14] was used for the simulation of a distributed memory multiprocessor. The primary advantage of using a simulator is that it allows us to easily vary hardware parameters such as network latency and overhead. The core of the simulator is the processor module, which models a modern superscalar processor with dynamic branch prediction, rename registers, a large instruction window, and out-of-order execution and retirement. For this set

of experiments, the processor and memory configuration parameters were set as shown in Table 2.

The simulator supports a message-passing multiprocessor model. The simulator does not include network contention, but it does include a configurable network latency parameter. In addition, the overhead of sending and receiving messages is included in the simulation, since the application must interact with the network interface device's buffers. Also, the simulator provides a hardware gap parameter to limit network bandwidth and a per-message network controller overhead parameter.

We implemented our algorithms using a library that provides a shared memory interface in which access to remote memory is accomplished with explicit `get()` and `put()` library calls. The library implements these operations using a bulk-synchronous style in which `get()` and `put()` calls merely enqueue requests on the local node. Communication among nodes happens when the library's `sync()` function is called. During a `sync()`, the system first builds and distributes a communications plan that indicates how many `gets` and `puts` will occur between each pair of nodes. Based on this plan, nodes exchange data in an order designed to reduce contention and avoid deadlock. This library runs on top of Armadillo's high-performance message-passing library (`libmvpplus`).

Our system allows us to set the network's bandwidth, latency, and per-message overhead. Table 3 summarizes the default settings for these hardware parameters as well as the observed performance when we access the network hardware through our shared memory library software. Note that the bulk-synchronous software interface does not allow us to measure the software o and l values directly. The hardware primitives' performance correspond to values that could be achieved on a network of workstations (NOW) using a high-performance communications interface such as 'Active Messages' and high-performance network hardware such as 'Myrinet®'. Note that the software overheads are significantly higher because our implementation copies data through buffers and because significant numbers of bytes sent over the network represent control information in addition to data payload.

Table 2
Architectural parameters for each node in multiprocessor

Parameter	Setting
Functional units	4 int/4 FPU/2 load-store
Functional unit latency	1/1/1 cycle
Architectural registers	32
Rename registers	Unlimited
Instruction issue window	64
Max. instructions issued per cycle	4
L1 cache size	8KB 2-way
L1 hit time	1 cycle
L2 cache size	256KB 8-way
L2 hit time	3 cycles
L2 miss time	3 + 7 cycles
Branch prediction table	64K entries, 8-bit history
Subroutine link register stack	Unlimited
Clock frequency	400 MHz

Table 3
Raw hardware performance and measured network performance (including hardware and software) for simulated system

Parameter	Hardware setting	Observed performance (HW + SW)
Gap g (Bandwidth)	3 cycles/byte (133 MB/s)	35 cycles/byte (put), 287 cycles/byte (get)
Per-message overhead o	400 cycles (1 μs)	N/A
Latency l	1600 cycles (4 μs)	N/A
Synchronization barrier L	N/A	25500 cycles (16-processors) (64 μs)

References

[1] M. Adler, J. Byer, R.M. Karp, Scheduling parallel communication: the h-relation problem, in: Proceedings of MFCS, 1995.

[2] M. Ajtai, J. Komlos, E. Szemeredi, An $O(n \log n)$ sorting network, in: Proceedings of the ACM STOC, 1983, pp. 1–9.

[3] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, P. Spirakis, BSP vs LogP, in: Proceedings of the ACM SPAA, 1996, pp. 25–32.

[4] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, P. Spirakis, BSP vs LogP, Algorithmica 24 (1999) 405–422.

[5] E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, S.W. Song, Efficient parallel graph algorithms for coarse grained multicomputers and BSP, in: Proceedings of the ICALP, Lecture Notes in Computer Science, Vol. 1256, Springer, Berlin, 1997, pp. 390–400.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: towards a realistic model of parallel computation, in: Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, May 1993, pp. 1–12.

[7] W.D. Frazer, A.C. McKellar, Samplesort: a sampling approach to minimal storage tree sorting, J. ACM 17 (3) (1970) 496–507.

[8] A.V. Gerbessiotis, L.G. Valiant, Direct bulk-synchronous algorithms, J. Parallel and Distributed Computing 22 (1994) 251–267.

[9] P.B. Gibbons, Y. Matias, V. Ramachandran, Efficient low-contention parallel algorithms, J. Comput. System Sci. 53 (3) (1996) 417–442 (Special issue of papers from 1994 ACM SPAA.).

[10] P.B. Gibbons, Y. Matias, V. Ramachandran, The queue-read queue-write asynchronous PRAM model, Theoret. Comput. Sci. 196 (1998) 3–29.

[11] P.B. Gibbons, Y. Matias, V. Ramachandran, The queue-read queue-write PRAM model: accounting for contention in parallel algorithms, SIAM J. Comput. 28 (2) (1999) 733–769.

[12] P.B. Gibbons, Y. Matias, V. Ramachandran, Can a shared-memory model serve as a bridging model for parallel computa-

tion?, Theory Comput. Systems 32 (1999) 327–359 (Special issue of papers from *1997 ACM SPAA*.).
[13] M. Goodrich, Communication-efficient parallel sorting, in: Proceedings of the ACM STOC, 1996, pp. 247–256.
[14] B. Grayson, Armadillo: a high-performance processor simulator, Masters Thesis, ECE, UT-Austin, 1996.
[15] B. Grayson, M. Dahlin, V. Ramachandran, Experimental evaluation of QSM: a simple shared-memory model, in: Proceedings of IPPS-SPDP'99, pp. 130–137. (see also TR98-21, Dept. of Computer Science, UT-Austin, 1998).
[16] J.S. Huang, Y.C. Chow, Parallel sorting and data partitioning by sampling. Proceedings of the 7th IEEE International Computer Software and Applications Conference, 1983, pp. 627–631.
[17] R.M. Karp, V. Ramachandran, Parallel algorithms for shared-memory machines, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. A, Elsevier, Amsterdam, The Netherlands, 1990, pp. 869–941.
[18] R. Karp, A. Sahay, E. Santos, K.E. Schauser, Optimal broadcast and summation in the LogP model, in: Proceedings of the 5th ACM SPAA, June–July 1993, pp. 142–153.
[19] P.D. MacKenzie, V. Ramachandran, Computational bounds for fundamental problems on general-purpose parallel models, in: Proceedings of the 10th ACM SPAA, June–July 1998, pp. 152–163.
[20] V. Ramachandran, A general purpose shared-memory model for parallel computation, in: Robert S. Schrieber, Michael T. Heath, Abhiram Ranadae (Eds.), Algorithms for Parallel Processing, Vol. 105, IMA Volumes in Mathematics and its Applications, Springer, Berlin, 1999, pp. 1–17.
[21] V. Ramachandran, B. Grayson, M. Dahlin, Emulations between QSM, BSP and LogP: a framework for general-purpose parallel algorithm design. Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), 1999.
[22] R. Reischuk, Probabilistic parallel algorithms for sorting and selection, SIAM J. Computing 14 (1985) 396–409.
[23] H. Shi, J. Schaeffer, Parallel sorting by regular sampling, J. Parallel and Distributed Computing 14 (1992) 372–382.
[24] L.G. Valiant, A bridging model for parallel computation, Comm. ACM 33 (8) (1990) 103–111.

Vijaya Ramachandran is Professor of Computer Sciences at the University of Texas at Austin. She received her Ph.D. from Princeton University in 1983. Her primary research interests are in the theory and evaluation of algorithms and parallel computation. She is Area Editor for Parallel Algorithms for the *Journal of the ACM* and serves on the Editorial Boards of *SIAM Journal on Computing*, *SIAM Journal on Discrete Mathematics*, *Journal of Algorithms* and *Parallel Processing Letters*.

Brian Grayson received his Ph.D. in Electrical and Computer Engineering from The University of Texas at Austin in 1999. He is currently employed by Motorola in Austin, Texas. His interests include performance modeling and analysis of microprocessor microarchitecture.

Mike Dahlin is an Associate Professor in the Department of Computer Sciences at the University of Texas at Austin. His work focuses on large-scale distributed systems. Dr. Dahlin received his Ph.D. from the University of California at Berkeley in 1996, the NSF CAREER award in 1998, and the Sloan Research Fellowship in 2000.