Emulations Between QSM, BSP and LogP: A Framework for General-Purpose Parallel Algorithm Design*

Vijaya Ramachandran† Brian Grayson‡ Michael Dahlin§

November 23, 1998

UTCS Technical Report TR98-22

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 nonstalling LogP models.

The QSM is a shared-memory model with only two parameters – p, the number of processors, and g, a bandwidth parameter. These features of the QSM make it a convenient model for parallel algorithm design, and the simple work-preserving emulations of QSM on BSP and LogP show that algorithms designed on the QSM will map well on to these other models. This presents 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. Using appropriate cost measures, we analyze the performance of these algorithms and describe simulation results. These results suggest that QSM analysis will predict algorithm performance quite accurately for problem sizes that arise in practice.

1 Introduction

There is a vast amount of literature on parallel algorithms for various problems. However, algorithms developed using traditional approaches on PRAM and 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 [10]. 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 because each has only 2 parameters and because they are shared-memory, which is generally more convenient than message passing for developing parallel algorithms.

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

*This work was supported in part by an NSF CISE grant (CDA-9624082), and grants from Intel, Novell, and Sun.

†Dept. of Computer Sciences, Univ. of Texas, Austin, TX 78712. Email: vlr@cs.utexas.edu

‡Dept. of ECE, Univ. of Texas, Austin, TX 78712. Email: bgrayson@ece.utexas.edu. Also supported in part by an NSF Graduate Fellowship.

§Dept. of Computer Sciences, Univ. of Texas, Austin, TX 78712. Email: dahlin@cs.utexas.edu. Also supported in part by NSF CAREER grant 9733842.

1. We present work-preserving emulations with only modest (polylog) slowdown between the LogP model and the other 3 models. These results indicate that the four models are more or less interchangeable for the purpose of algorithm design. 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.

   The only mis-match we have is between the 'stalling' and 'nonstalling' LogP models. Here we show that an earlier result claimed in [4] is erroneous by giving a counterexample to their claim. Work-preserving emulations between BSP, QSM and s-QSM were presented earlier in [10, 20].

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 maps on to both in a simple and efficient way. 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, (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 prefix sums, sample sort, and list ranking, and we analyze them under this cost metric. We also describe simulation results for these algorithms that indicate that the difference between the BSP and QSM cost metrics is small for these algorithms for reasonable problem sizes.

Several of the algorithms we present 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 $c > 0$.

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 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. A summary of earlier emulation results can be found in the appendix.

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 $T = \max(w,\ g \cdot h,\ L)$. The time taken by a BSP algorithm is the sum of the costs of the individual supersteps in the algorithm.

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 π *stalls* and does not perform any operation until it can the message.

QSM and s-QSM models. The Queuing Shared Memory (QSM) model [10] consists of a number of identical 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 $\max(m_{op}, g \cdot m_{rw}, \kappa)$. 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 $\max(m_{op}, g \cdot m_{rw}, g \cdot \kappa)$, i.e., the gap parameter is applied to the accesses at memory as well as to memory requests issued at processors.

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		$\boxed{O(\log^4 p\ +\ (l/g)\log^2 p)}$	$O(\lceil \frac{g \log p}{L} \rceil)$	$O(\lceil \frac{g \log p}{L} \rceil)$
LogP (nonstalling)	$\boxed{O(L/l)\ \ (det.)^{\dagger}}$	1 (det.)	$\boxed{O(\lceil \frac{g \log p}{l} \rceil)}$	$\boxed{O(\lceil \frac{g \log p}{l} \rceil)}$
s-QSM	$O((L/g) + \log p)$	$\boxed{O(\log^4 p\ +\ (l/g)\log^2 p)}$		1 (det.)
QSM	$O((L/g) + g \log p)$	$\boxed{O(\log^4 p + (l/g)\log^2 p + g \cdot \log p)}$	$O(g)$ (det.)	

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 [10, 20].*

[1] This result is presented in [4] but it is stated there erroneously that it holds for stalling LogP programs. We provide a counterexample.

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 [7].

3 Emulation Results

The results on work-preserving emulations between models are tabulated 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 [4] 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 [4]; 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. 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.

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 $w = p \cdot t$), the emulation algorithm will map the p-processor s-QSM algorithm on to a p'-processor BSP, for any $p' \leq p/((L/g) + \log p)$, to run on the BSP in time $t' = O(t \cdot (p/p'))$ whp in p. Note that if sufficient parallelism exists, for a machine with p physical processors, one would typically design the BSP algorithm on $\Theta((L/g) + \log p) \cdot p)$ 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.

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 *[17] The following two problems can be computed in time $O(l\lceil\frac{\log p}{\log(l/g)}\rceil)$ on p processors under the LogP model.*
1. Barrier synchronization on the p LogP 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 3.1 *Suppose we are given an algorithm to route an h-relation on a p-processor LogP while satisfying the capacity constraint in time $O(g\cdot(h+H(p))+l)$, when the value of h is known in advance. Then,*
1. There is a work-preserving emulation of a p-processor QSM on LogP with slowdown $O(g\cdot\log p+\log^2 p+(H(p)+B(p))\cdot\frac{\log p}{\log\log p})$ whp in p.
2. There is a work-preserving emulation of a p-processor s-QSM and BSP on LogP with slowdown $O(\log^2 p+(H(p)+B(p))\cdot\frac{\log p}{\log\log p})$ 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 $O(g\cdot(h+H(p))+l)$ 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 Figure 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, since the maximum sent ($maxsend$) is 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 3.1 *The algorithm for Step II.1 runs in time $O(g\log^2 p+(H(p)+B(p))\cdot(\log p)/\log\log p)$ 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.*

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

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

Proof: The correctness of the algorithm follows from the following observations, which can be derived using Chernoff bounds:
1. If $\mu \geq \log p$ 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 $q \geq (2 \log p)/maxsend$ 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 $maxsend \geq m/p$, hence the **while** loop is executed $O(\log p/\log\log p)$ times. Each iteration takes time $O(g(\mu \cdot \log p + H(p)) + l)$ whp to route the h-relation, and time $O(B(p))$ to compute μ and perform a barrier synchronization. Hence each iteration takes time $O(g \cdot (\mu \log p + H(p) + B(p)))$ since $l < B(p)$. Since the **while** loop terminates when $\mu \geq \log p$, the overall time taken by the algorithm is $O(g \log^2 p + g \cdot (\log p/\log\log p)(H(p) + B(p)))$. □

Finally, to complete the proof of Theorem 3.1 we need to show that the emulation algorithm is work-preserving for each of the three models. Let $\tau = \log^2 p + (H(p) + B(p)) \cdot (\log p)/\log\log p$.

If $p' \leq p/\tau$ then the time taken by the emulation algorithm to execute steps II.1 and II.3 is $O(g \cdot \tau)$, and hence the work performed in executing these two steps is $O(g \cdot \tau \cdot p') = O(g \cdot p)$. Since any phase or superstep of the emulated machine must perform work $\geq g \cdot p$, 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 $p' \leq p/\tau$, then a BSP superstep that takes time $c + gh + L$ will be emulated in time $O((p/p') \cdot (c + gh) + l)$ on a LogP with p' processors and hence is work-preserving. (We assume that $l \leq L$ since L includes the cost of synchronization.)

Next consider a phase on a p processor s-QSM in which h is the maximum of the maximum number of reads/writes by a processor and 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 [10], the number of messages sent or received by any of the p' LogP processors is $O(h \cdot (p/p'))$ whp in p, if $p' \leq p/\log p$. Hence the memory accesses can be performed in time $T = O(g \cdot h \cdot (p/p'))$ whp in p, once the value of h is determined.

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. $h := \max(\log^2 p, s, \max_i r_i)$

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

This is work-preserving since $T \cdot p' = O(g \cdot h \cdot p)$.

Similarly, we can obtain the desired result for QSM by using the result in [10] 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 $O(h \cdot (p/p'))$ whp in p, if $p' \leq p/g \log p$. □

Corollary 3.1 *(to Theorem 3.1)*
1. There is a work-preserving emulation of a p-processor QSM on LogP with slowdown $O(g \cdot \log p + \log^4 p + \frac{l/g}{\log(l/g)} \cdot \frac{\log^2 p}{\log \log p})$ *whp in p.*
2. There is a work-preserving emulation of a p-processor s-QSM and BSP on LogP with slowdown $O(\log^4 p + \frac{l/g}{\log(l/g)} \cdot \frac{\log^2 p}{\log \log p})$ *whp in p.*

Proof: The corollary follows from Theorem 3.1 using the algorithm in [17] for barrier synchronization on p-processor LogP that runs in time $O(l \lceil \frac{\log p}{\log(l/g)} \rceil)$, and the algorithm in [1] for routing an h-relation on a p-processor LogP in $O(g(h + \log^3 p \cdot \log \log p) + l)$ 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 Figure 1, although it is simpler to describe and analyze.

Claim 3.2 *The algorithm given in Figure 2 for Step II.1 determines an upper bound on the value of h whp in time* $O(gh + l \log p)$. *If* $h \geq \log^2 p$ *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 [3, 4], when $r \cdot p$ 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 $(m/(p \log p)) \cdot p$, which is $O((s/\log p) \cdot p)$. Hence the time needed to execute step 4 is $O(g \cdot s + l \log p)$ 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 Figure 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 3.2 *1. There is a work-preserving emulation of a p-processor QSM on LogP with slowdown $O(\log^3 p \cdot \log\log p + (g + (l/g)) \cdot \log p)$ whp in p.*
2. There is a work-preserving emulation of p-processor s-QSM and BSP on LogP with slowdown $O(\log^3 p \cdot \log\log p + (l/g) \log p)$ 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 [4] as an emulation where both the time and work increases by a factor of L/l. We observe that this emulation can be made work-preserving by using a BSP with a smaller number of processors and mapping L/l LogP processors onto each BSP processor.

The analysis in [4] 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 Figure 3. The following Claim shows that this computation cannot be mapped on to the BSP with constant slowdown.

Claim 3.3 *The LogP computation shown in Figure 3 takes time $O(r \cdot l + g \cdot q)$. When mapped on to the BSP this computation takes time $\Omega(r \cdot (L + g \cdot q))$.*

Proof: We note the following about the computation in Figure 3:

(i) At time $(i-1) \cdot l + g$, all processors in the ith group send a message to processor P_i, $1 \le i \le r$. This is a stalling send if $q > l/g$. Processor P_i then receives all messages at time $i \cdot l + g \cdot q$.

(ii) The computation terminates at time $r \cdot l + g \cdot q$ when P_r receives all messages sent to it.

On a BSP we note that the computation in Figure 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 $\Omega(r \cdot (L + g \cdot q))$, which is $\Omega(r \cdot L + r \cdot g \cdot q)$ time.

Hence the slowdown of this emulation is $\Omega(\frac{r \cdot L + r \cdot g \cdot q}{r \cdot l + g \cdot q})$.

If r is any non-constant function with $r \cdot l = o(g \cdot q)$ and $l \le L$, then the slowdown of this emulation is $\Theta(r)$ and is not dependent on the ratio L/l. Example values that satisfy the above constraints are $l = \log p$, $L = \log^2 p$, $g = 1$, $r = n^{1/3}$, and $q = n^{2/3}$, where p is the number of processors and $n > p$ is the size of the input. In this case the slowdown of the emulation is $\Omega(n^{1/3})$.

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 ≤ j ≤ 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 ≤ i ≤ r
    pfor 1 ≤ j ≤ 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 ≠ r)
            b. sends a message to processor P_i.
    rofp
rofp
```

Figure 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 3.3 *Consider the deterministic emulation of LogP on BSP.*
1. A nonstalling LogP program can be emulated deterministically in a work-preserving manner with slowdown L/l.
2. If the LogP program is allowed to be stalling then
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: We have already shown 1 and 2a so we only need to show 2b. 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 Figure 3 with $r = \omega(\tau)$ and with $r \cdot l = o(g \cdot q)$ and $l \le L$. Then the work performed by the LogP computation is $\Theta(g \cdot q \cdot p)$ while the work performed by the emulating BSP computation is $\Theta(r \cdot g \cdot q \cdot p/\tau)$, which is $\omega(g \cdot q \cdot p)$. Hence the emulation is not work-preserving. $\Box$

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 output of a program 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 $O(\lceil \frac{g \log p}{l} \rceil)$ whp as follows:

I. Divide the LogP computation into blocks of size l

II. Emulate each block in $O(\lceil \frac{g \log p}{l} \rceil)$ 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 [10] 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 $c \cdot \max(k_i, 1) \cdot \log^3 M)$, 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 3.4 *A non-stalling LogP computation can be emulated on the QSM or s-QSM in a work-preserving manner whp with slowdown $O(\lceil \frac{g \log p}{l} \rceil)$, 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. The one mis-match we have is between stalling and non-stalling LogP, and here we show that there is no deterministic step-by-step emulation of stalling LogP on BSP that is work-preserving. This is in contrast to the inference made in [4] that LogP is essentially equivalent to BSP.

The algorithms 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 the 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. 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.

For the rest of this paper we will use the QSM and s-QSM as our basic models, and we analyze the algorithms using the s-QSM cost metric. We do this since 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. In fact, for the algorithms we present in the rest of the paper, the upper bounds we derive are tight 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 performed 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 $g \cdot n$, where n is the size of the input [10].

The *maximum parallelism* of an algorithm performing $w(n)$ work is the smallest running time $t(n)$ achievable by the algorithm while performing $w(n)$ work. This is a meaningful measure for a QSM or s-QSM algorithm, as for a PRAM algorithm, since these algorithms can always be slowed down (by using a smaller number of processors) while performing the same work [10].

The motivation for the second metric on minimizing number of phases (which is the new one) is the following. One major simplification made by the QSM models 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 an s-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 [12] 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 [12] 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 ask for algorithms that perform optimal work and communication and additionally compute in a small number of phases.

Input. Array $A[1..n]$ to a p-processor QSM.
// Preprocess to reduce size to p:
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 $B[i]$.
rofp
// Main loop
$r := \frac{n \log(n/p)}{p \log n}$
$k := p$
repeat
  pfor $1 \leq i \leq \lceil k/r \rceil$ **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.

Figure 4: Prefix sums algorithm.

By placing the maximization of parallelism as a consideration secondary to minimizing work and number of phases, we are emphasizing our desire for practical algorithms; thus providing good performance for tiny problem sizes is not a primary consideration in our metric. Our emphasis is on simple algorithms that can be used in practice, hence we are mainly interested in 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. Some of our algorithms achieve a higher level of parallelism, but our goal in developing them 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 Figure 5) has an improved performance under this assumption of distributed input and output. In the other algorithms we present, the savings gained by this representation 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 Figure 4.

Theorem 4.1 *The algorithm in Figure 4 computes the prefix sums of array* $A[1..n]$*, and runs in* $O(gn/p)$ *time (and hence* $O(gn)$ *work) and* $O(\frac{\log p}{\log(n/p)})$ *phases when* $p \leq n/\log n$ *on QSM and s-QSM.*

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

Figure 5: Simple prefix sums algorithm for $p \leq \sqrt{n}$.

Proof: Let t be the number of iterations of the **repeat** loop. Then $t = O(\log p/\log r)$, i.e., $t = O(\log p/\log(n/p))$. 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 $O(\log p/\log(n/p))$.

For the algorithm to terminate we need $r > 1$, and the time taken by each iteration of the **repeat** loop is $O(g \cdot r)$, hence the overall running time of the **repeat** loop is $O(t \cdot gr)$, which is $O((gn/p) \cdot (\log n/\log(n/p)) = O(gn/p)$. 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 $O(g \cdot n)$ work as long as $p = O(n/\log n)$. □

This result is optimal for s-QSM since there is a corresponding lower bound for the work [10], time [19] and the number of phases [19]. Note that this algorithm runs in a constant number of rounds if $p = O(n^c)$, for some constant $c < 1$.

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 $p \leq \sqrt{n}$, which is the situation that typically arises in practice. This algorithm is shown in Figure 5. It is straightforward to see that this algorithm computes the result in $O(g \cdot n/p)$ 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 4.2 *The simple prefix sums algorithm runs in $O(gn/p)$ time and in two phases when $p \leq \sqrt{n}$.*

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 $O(g \cdot p)$ time when $p \leq \sqrt{n}$.

4.3 Sample Sort Algorithm

Figure 6 shows the QSM sample sort algorithm. We assume that $p \leq \sqrt{\frac{n}{\log n}}$; 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 [15, 22, 23, 11, 9]. In recent related work, we have investigated a modified sample sort algorithm with a slightly different pivot selection method, and we have shown it to have superior performance. Details of this method can be found in [21].

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

Figure 6: Sample sort algorithm.

Theorem 4.3 *The algorithm in Figure 6 sorts the input array while performing optimal work ($O(g \cdot n + n \log n)$), optimal communication ($O(g \cdot n)$), in $O(1)$ phases whp when the number of processors $p = O(\sqrt{\frac{n}{\log n}})$.*

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, $p \leq \sqrt{n/\log n}$. 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 $p \leq \sqrt{n/\log n}$. Step 2b takes time $O(p \log n \log(p \log n)) = O((n/p) \log(n/p))$, and step 2c takes time $O((n/p) \cdot \log p)$ 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 $Y_{i,j}$, $1 \leq i \leq c \log n, 1 \leq j \leq p$, be a random variable that is 1 if the ith sample of the jth processor lies in I, and is zero otherwise.

$Pr[Y_{i,j} = 1] = s_j \cdot p/n$, for $1 \leq i \leq c \log n$, 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.

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.

Figure 7: List ranking algorithm.

$E[Y] = c \log n \sum_{j=1}^{p} s_j \cdot (p/n) = (s \cdot c \cdot p \cdot \log n)/n$
Hence $E[Y] = \alpha c \log n$.

By Hoeffding's inequality, $Pr[Y \leq k] \leq Pr[X \leq k]$, 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, $Pr(X \leq c \log n) \leq e^{-\frac{c \cdot (\alpha - 1)^2 \ln n}{2\alpha \ln 2}} = n^{-c(\alpha-1)^2/(2\alpha \ln 2)}$,
i.e., $Pr(Y \leq c \log n) \leq n^{-c(\alpha-1)^2/(2\alpha \ln 2)}$.

Let a_i be the position of the $ci \log n$th sample in the sorted sequence S, $1 \leq i \leq p-2$. Let B_i be the interval of size $\alpha n/p$ on sequence S starting at a_i, $1 \leq i \leq p-2$. 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, $0 \leq i \leq p-1$ has less than $c \log n$ samples is no more than $p \cdot n^{-c(\alpha-1)^2/(2\alpha \ln 2)}$, 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 $O(g \cdot (n/p) + (n \log n)/p)$, 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

Figure 7 summarizes the list ranking algorithm.

Theorem 4.4 *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 $p = o(n^{\epsilon})$. 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,

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

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 $p = o(n^\epsilon)$, for any $\epsilon > 0$, 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 $\leq (1-\beta)/4)^{c \log p} \cdot n$ with exponentially high probability. With a suitable choice of c this number of elements remaining can be made $\leq n/p$.

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 $p = \Omega(n^\epsilon)$, we can use a standard analysis of randomized list ranking to show that all elements at a 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* [13], which is a simulated architecture with parameterizable configurations and cycle-by-cycle profiling and accuracy. The simulator was set to parameters for a state-of-the-art machine. A detailed description of this experimental work can be found in [14].

The experiments in [14] 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. In this section we concentrate on presenting the analyses that led to the graphs in the basic experiment performed in [14]. These plots from [14] are attached to the end of this paper. For details of the other experiments and conclusions derived from them, see [14].

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 nonoblivious algorithms and variations introduced by randomization 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.

The architecture of the Armadillo simulator is described in the appendix. In the following subsections, we describe our analysis on bounding the constant factors in the performance bounds of the code we implemented for the three algorithms, and we discuss the experimental results we obtained. The pseudo-code for the experiments is included in the appendix.

5.1 General Comments

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 and by the more detailed BSP model. 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 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.

4. The 'QSM estimate' line is a plot of the measured maximum number of communication steps 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.) 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.

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 presented here.

We now describe in some detail the computations that led to the computed plots in the various graphs.

5.2 Description of the Graphs

In this section we describe the equations we used to plot the computed curves for QSM for each of the three algorithms.

5.2.1 Prefix Sums

We implemented the simple prefix sums algorithm shown in Figure 5. Since the ith block of n/p input elements was distributed to the ith processor ahead of the computation and the output location for the prefix sums for these elements was designated to be on the i processor (the natural distribution), the communication cost for this algorithm is $g(p-1)$, and the total running time is $O(gp + n/p)$.

5.2.2 Sample Sort

We implemented the sample sort algorithm given in Figure 6.

The 'QSM ideal' plot shows the computed communication time assuming that each bucket is of size no more than $\lceil n/p \rceil$. The algorithm chose $4p \log n$ samples, hence the equation plotted is

$$T_I^{SS}(n) = 4(p-1)g \log n + 3(p-1)g + (gn/p) \cdot (p-1)/p$$

Here the first term is for the broadcast performed in steps 1c and 2a of the sample sort algorithm, the second term is for step 2d, and the third term is for the read in step 3. The factor $(p-1)/p$ in the third term accounts for the fact that in the ideal case, $1/p$ of the elements in each bucket will be local to the processor that needs to sort that bucket and hence will not participate in the bucket redistribution step. Since we assume ideal behavior by the algorithm, the write in step 3 has cost 0, since the elements in each bucket will be in their final position after the local sort in step 3.

The 'QSM whp' plot shows the computed communication time that is guaranteed with probability greater than .9. The running time here is

$$T_{whp}^{SS}(n) = 4(p-1)g \log n + 3(p-1)g + gBr + gB,$$

where B is the size of the largest bucket, and r is a bound on the fraction of elements in any bucket that are outside the processor that will sort that bucket. Here the term gBr represents the time to perform the read in step 3 to copy the elements in each bucket into the appropriate processor, and the term gB represents the time to write back the sorted elements into their final location.

To obtain a bound on B we use the analysis in Section 4.3. We solve for a value of α that is guaranteed by Chernoff bound to give a bucket size no larger than $B = \alpha n/p$ with probability at least $1 - q$ (where q was set to 0.05). For this we solve for α in the following equation:

$$\frac{(\alpha-1)^2}{\alpha} \geq ((\ln 2)/2) \frac{\log_{10}(1/q) + \log_{10} p}{\log_{10} n}$$

Solving for α, we obtain $\alpha = (-b + \sqrt{b^2 - 4})/2$, where $b = -(2 + ((\ln 2)/2) \cdot (\log(1/q) + \log p)/\log n)$.

To solve for r we use Chernoff bounds again. Let P be a processor whose bucket size is maximum, and let X be the number of elements in that bucket that are local to processor P before step 3 of the algorithm in Figure 6. Then $E[X] \leq \alpha n/p^2$, and $Pr(x \leq (1-\beta) \cdot B/p) \ \leq \ e^{-\beta^2 B/(2p)}$.

We solve for β in the equation $(p/2) \cdot e^{-\beta^2 B/(2p)} \ \leq \ q'$, where we set $q' = .05$, and the factor of $p/2$ comes from the (generous) observation that no more than $p/2$ of the processors can have the maximum bucket size whp.

Thus, $\beta = \sqrt{(2p/B) \cdot \ln(p/(2q'))}$ and hence $r = (p - 1 + \beta)/p$.

Since we set each of q and q' to 0.05, the overall success probability is at least .9.

5.2.3 List Ranking

We implemented the list ranking algorithm given in Figure 7.

The equation for the time taken by the communication on a p-processor computation is

$$T^{LR}(n) = \pi g \cdot ((A \cdot c_1/2) \ + \ (B \cdot c_2)/4) \sum_{i=1}^{c \log p} x_i) + \pi' \cdot Cgz$$

where the various parameters are the following:

$c = 4$ since we implemented step 2 of the list ranking algorithm to run for $4 \log n$ iterations.

x_i is a bound on the maximum number of elements at any processor in the ith iteration of the **for** loop in step 2 of the list ranking algorithm.

$A = 1$ is the number of communication steps performed by elements that flipped a one bit in an iteration of the **for** loop in step 2 of the list ranking algorithm, and c_1 is a correction factor to compute a bound on the maximum number of elements that flipped a one bit at any processor in the i th iteration as $(c_1/2) \cdot x_i$.

$B = 7$ is the number of communication steps performed for elements that are eliminated in an iteration of the **for** loop in step 2 of the list ranking algorithm, and c_2 is a correction factor to compute a bound on the maximum number of elements that eliminated themselves at any processor in the ith iteration as $(c_2/4) \cdot x_i$.

z is the total number of elements remaining after step 2.

$C = 4$ is the number of communication steps involves these z elements.

π is a bound on the fraction of elements that flipped a bit whose successors/predecessors are not at the same processor, and π' is a bound on the fraction of the elements remaining after step 2 that are not in processor P_0.

For the 'QSM ideal' plot, we assume there is no skew in the performance of the randomized steps. Hence we obtained the following values for the parameters:
$x_i = (n/p) \cdot (3/4)^{i-1}, \ \ z = n \cdot (3/4)^{4 \log p}, \ \ c_1 = c_2 = 1, \ \ \pi = \pi' = (p-1)/p,$
where n is the size of the input. By approximating the summation by the sum of the infinite geometric series, we obtained the following equation for the 'QSM ideal' plot:

$$T_I^{LR}(n) = ((p-1)/p) \cdot 2.25g \cdot 4 \cdot (n/p) + 4gn \cdot (3/4)^{4 \log p}$$

The 'QSM whp' plot gives the bound on communication that we can guarantee with probability at least .9. This bound was obtained using Chernoff bound as described in Section 4.4 and is rather weak, since many approximations were made in the analysis. We describe below the values we used for the various parameters.

We obtained a bound on x_{i+1} using Chernoff bounds, first to obtain a lower bound on x_i', the number of elements in any processor at each of odd and even positions on the linked list, and then to obtain a lower bound on the number of these elements that are eliminated in the ith iteration. To compute this we first computed a bound on x_i' that is guaranteed at any processor with probability at least $1 - q_2$ as $x_i' = (1-\alpha) \cdot x_i/2$, where $\alpha = \sqrt{(4/x_i) \cdot \ln((2pc \log p)/q_2)}$.

We then computed a bound on x_{i+1} that is guaranteed at any processor with probability at least $1 - q_1$ as $x_{i+1} = x_i \cdot (3/4 + \beta'/4)$, where $\beta' = \sqrt{(2/x_i') \cdot \ln((2pc \log p)/q_1)}$

We obtained an upper bound on c_1 that is guaranteed at any processor with probability at least $1 - q_4$ as $c_1 = 1 + \gamma$, where $\gamma = \sqrt{(6/x_i) \cdot \ln((pc \log p)/q_4)}$.

We decided to set $c_2 = c_1$ to simplify the analysis, since it appeared that the benefit obtained from computing an upper bound on c_2 was out-weighed by the fact that we needed to devote some probability to this computation.

We computed a bound on z that is guaranteed with probability at least q_3 using Chernoff bounds to bound the total number of elements remaining at each iteration of the **for** loop, and hence obtained a bound for z as the bound on the number of elements remaining after the last iteration of the **for** loop. For this we set $r = n$ initially, and in the ith iteration we updated r to $r = r \cdot (3/4 + \beta/4)$, where $\beta = \sqrt{(8/r) \cdot \ln((c \log p)/q_3)}$.

We set $\pi = \pi' = 1$ to simplify the analysis.

We set $q_1 = .04$ and $q_2 = q_3 = q_4 = .02$ to obtain an overall probability of success of at least .9.

5.3 Discussion

The graphs for the three algorithms are given at the end of the paper.

As expected the communication cost for the prefix sums algorithm is negligible compared to the total computation cost as n becomes large. Hence we have only shown the plots for 16 processors. 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 discrepancy between the predicted and measured communication costs is discussed in [14].

As expected, for both sample sort and list rank 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 16 processors, 'QSM estimate' is within 10% of 'communication' for sample sort when the input size is larger than 125,000, and is within 15% of 'communication' for list rank when input size is larger than 40,000. The 'BSP estimate' lines are very close to the 'QSM estimate' lines for both algorithms.

For both sample sort and list rank 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, as seen from the equations that led to the 'QSM WHP bound' lines for both algorithms, the bounds computed 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 Conclusions

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.

References

[1] M. Adler, J. Byer, R. M. Karp, Scheduling parallel communication: The h-relation problem. In *Proc. MFCS*, 1995.

[2] M. Adler, W. Dittrich, B. Juurlink, M. Kutylowski, I. Rieping. Communication-optimal parallel minimum spanning tree algorithms. In Proc. ACM SPAA, pp. 27–36, 1998.

[3] M. Ajtai, J. Komlos, E. Szemeredi, An $O(n \log n)$ sorting network. In *Proc. ACM STOC*, pp. 1–9, 1983.

[4] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, P. Spirakis. BSP vs LogP. In *Proc. ACM SPAA*, pp. 25–32, 1996.

[5] E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Proc. ICALP*, LNCS 1256, pp. 390-400, 1997.

[6] 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.

[7] 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 SODA*, pages 638-648, January 1994.

[8] 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 *1994 ACM SPAA*.

[9] P.B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science*, vol. 196, 1998, pp. 3-29.

[10] 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 SPAA*, June 1997, pp. 72-83. To appear.

[11] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous algorithms. *J. Parallel and Distributed Computing*, 22:251-267, 1994.

[12] M. Goodrich. Communication-efficient parallel sorting. In *Proc. ACM STOC*, pp. 247–256, 1996.

[13] B. Grayson. Armadillo: A High-Performance Processor Simulator. Masters thesis, ECE, UT-Austin, 1996.

[14] B. Grayson, M. Dahlin, V. Ramachandran, Experimental evaluation of QSM: A simple shared-memory model. TR98-21, Dept. of Computer Science, UT-Austin, 1998.

[15] J.S. Huang and Y.C. Chow. Parallel sorting and data partitioning by sampling. *Proc. 7th IEEE Intl. Computer Software and Applications Conference*, pp. 627-631, 1983.

[16] 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.

[17] R. Karp, A. Sahay, E. Santos, and K.E. Schauser, Optimal broadcast and summation in the LogP model, In *Proc. 5th ACM SPAA*, 142–153, June-July 1993.

[18] 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.

[19] P. D. MacKenzie and V. Ramachandran. Computational bounds for fundamental problems on general-purpose parallel models. In *Proc. 10th ACM SPAA*, June-July 1998, pp. 152-163.

[20] V. Ramachandran. A general purpose shared-memory model for parallel computation. In Algorithms for Parallel Processing, Volume 105, IMA Volumes in Mathematics and its Applications, Springer-Verlag, to appear.

[21] V. Ramachandran, B. Grayson, M. Dahlin. An improved sample sort algorithm. Manuscript under preparation.

[22] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Jour. Computing*, 14:396-409, 1985.

[23] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *J. Parallel and Distributed Computing*, 14:382-372, 1992.

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

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 2: Architectural parameters for each node in multiprocessor.

APPENDIX

A Description of the Experimental Set-up

The Armadillo multiprocessor simulator [13] 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 are set for an advanced processor in 1998, and are not modified further. Table 2 summarizes these settings.

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'

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)

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

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.

B Summary of Earlier Results on Work-Preserving Emulations

QSM and s-QSM

Since any phase of an s-QSM can be performed with the same time cost or less on the QSM, it follows that the s-QSM can be emulated on the QSM in a work-preserving manner with no slowdown. For the reverse emulation we obtain a work-preserving emulation with slowdown g by mapping g QSM processors onto each s-QSM processor and then have each s-QSM processor emulate the computation of each of the QSM processors mapped to it.

Observation: [20] There is a deterministic work-preserving emulation of QSM on s-QSM with slowdown g.

Emulation of QSM and s-QSM on BSP

Theorem: [10]

There is a randomized work-preserving emulation of a p-processor QSM on BSP with slowdown $O(L/g+g\log p)$ whp in the size of the input.

There is a randomized work-preserving emulation of a p-processor s-QSM on BSP with slowdown $O(L/g+\log p)$ whp in the size of the input.

The emulation algorithm that leads to the above theorem is very simple, and is described below.

- Hash the QSM (or s-QSM) memory onto the BSP components.
- Map the QSM processors uniformly on to the BSP components.
- Have each BSP component emulate the (or s-QSM) QSM processors mapped to it.

A probabilistic analysis [10, 20] shows that the emulation has performance stated in theorem.

Emulation of BSP on QSM and s-QSM

Theorem. [10] A p-component BSP can be emulated in a work-preserving manner on a QSM or s-QSM with slowdown $O(\lceil(g/L)\cdot\log p\rceil)$ whp in p.

This emulation uses the shared memory of the QSM (or s-QSM) only to realize the h-relation routing performed by the BSP in each step. The following is a sketch of the emulation algorithm from [10].

Map the BSP processors uniformly among the QSM (or s-QSM) processors. In each phase, each QSM (or s-QSM) processor emulates the local computation for the current superstep of the BSP processors assigned to it.

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 BSP processors mapped to it as follows. Here n is the number of BSP processors.

1. Compute the total number of messages, M, to be sent by all processors in $O(g \log n)$ time and $O(M + gn)$ work.

2. Construct a sample S of the messages to be sent by choosing each message independently with probability $1/\log^3 M$. The size of the sample will be $O(M/\log^3 M)$ whp.

3. Sort the sample deterministically according to destination using a standard sorting algorithm, e.g., Cole's merge-sort; this takes $O(g \log M)$ time and $O(g \cdot M/\log^2 M)$ work.

4. Group the destinations into groups of size $\log^3 M$ and determine the number of messages destined for each group. This can be computed by a prefix sums computation that takes $O(g \log M)$ time and $O(gM)$ work.

5. Let k_i be the number of elements in the sample destined for the ith group. Obtain a high probability bound on the total number of messages to each group as $r_i = O(\max(k_i, 1) \cdot \log^3 M)$. Make $\log^3 M$ copies of each r_i, and place the duplicate values of the r_i in an array $R[1..n]$ such that $R[i]$ contains the bound for the group that contains destination $i, 1 \leq i \leq n$. This step can be performed in $O(g(1 + \log\log M/\log g))$ time and $O(ng)$ work using a broadcasting algorithm for each r_i.

6. In parallel, for each i, all processors with a message to a destination i read the value of this bound from $R[i]$; this takes time $\leq gh$ and $O(gM)$ work.

7. Use an algorithm for multiple compaction to get the messages in each group into a linear-sized array for that group; this takes $O(g \log M)$ time and $O(gM)$ work by the adaptation of the randomized QRQW algorithm for multiple compaction given in [8].

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

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

C Pseudo-codes

We describe the algorithms for prefix sums, sample sort and list ranking, as they were implemented on the simulator. For all algorithms the input and output was distributed uniformly across the P processors.

parallelprefix(array A, size n)

Step 1: Calculate local prefix sums. Each processor calculates a prefix sum on its local portion of the array.
Step 2: Exchange sums between processors. Each processor broadcasts a copy of its last sum to every other processor.
BARRIER SYNCHRONIZATION
Step 3: Final modification. Each processor adds up the sums from its preceding processors, and adds this offset to each of its previously-calculated prefix sums.

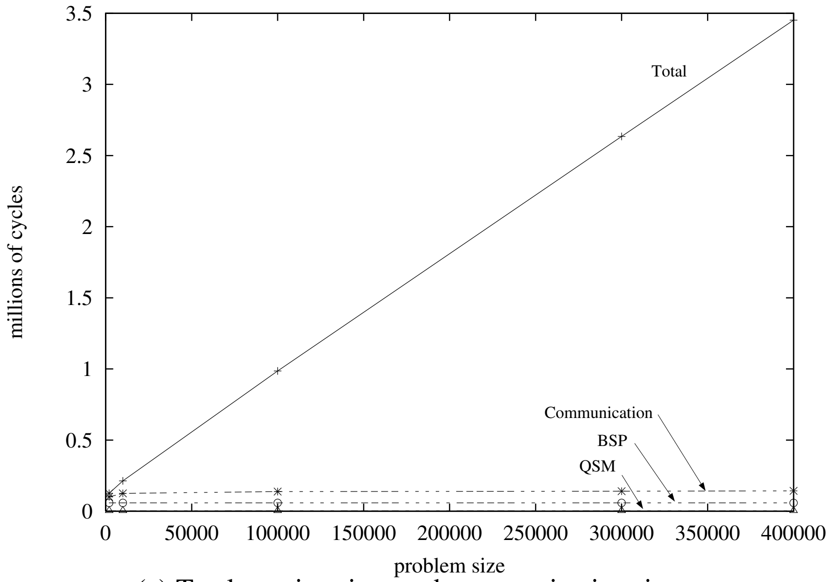
samplesort(array S, size n)

Major step 1: Pivot selection
  Allocate and "register" temporary structures.
  BARRIER SYNCHRONIZATION(to ensure the shared-memory "registrations" have completed)
  Each processor selects $c \log n$ of its elements randomly (with replacement),
    and broadcasts its samples to all other processors.
  BARRIER SYNCHRONIZATION
  Each processor quicksorts all $cP \log n$ samples, and selects every
    $c \log n$th element as a pivot (for a total of $P - 1$ pivots, or P "buckets").
Major step 2: Redistribution
  Assign each local element to one of of the P buckets, based on the chosen pivots,
    and reorder the elements locally so that all elements for the ith bucket are contiguous.
  For $1 \leq i \leq P$, every processor sends its count of elements for bucket i, along
    with a pointer to the location of these elements, to processor i.
  BARRIER SYNCHRONIZATION
  Each processor now fetches the other processors' contributions to its bucket.
  Each processor also participates in a parallel prefix of the total number of elements in each bucket.
  BARRIER SYNCHRONIZATION
Major Step 3: Local Sort
  for $1 \leq i \leq P$ in parallel
    processor i sorts the elements in the ith bucket.
Major Step 4: Redistribution
  Each processor writes the sorted elements of its bucket into the appropriate locations
    (calculated using the results of a prefix sums computation) in array S.
  BARRIER SYNCHRONIZATION

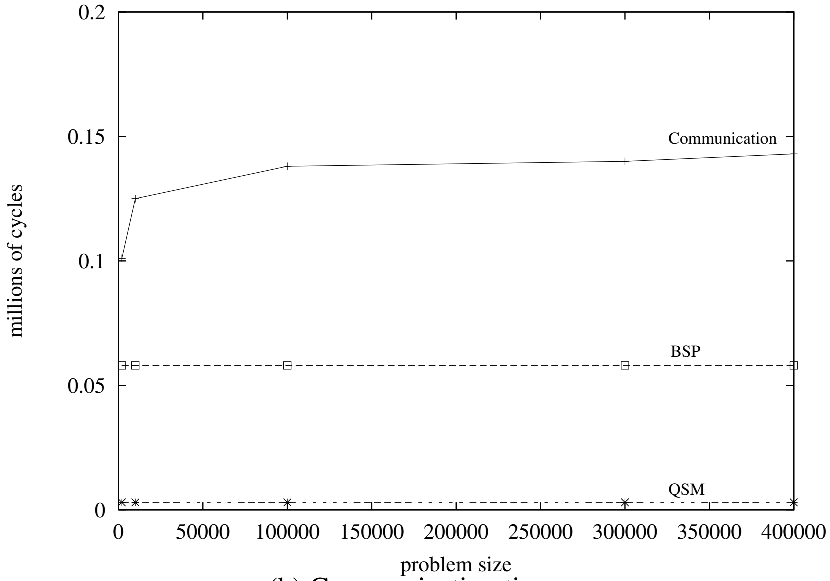
listrank(array S, array P, array R, size n)

Arrays: successor array S; predecessor array P; returned-ranks array R;
Local variables: indirection array I, flip array F, successor's flip array SF, removed element array RN, and temporary new ranks NR.
$Isize$ is the current number of elements, and $I[i]$ points to the ith element in the current linked list.

Initialization:
Initialize R to be all ones.
Initialize $I[i] = i$, to set up the initial indirection.
Allocate and register temporary structures.
Major step 1: Each processor repeatedly removes some elements from its list, until the list size is fairly small as follows.
for $c \cdot \log P$ iterations do
each active element i generates a flip (random bit), and stores it in $F[I[i]]$.
BARRIER SYNCHRONIZATION(to ensure shared-memory registrations have completed in the first loop, and to ensure that the updates from the previous loop have completed).
if i is not the head element, and i has a successor, and $F[I[i]]$ is 1 (i.e., i flipped a 1), then fetch its successor's flip into $SF[I[i]]$.
BARRIER SYNCHRONIZATION
if $F[I[i]] = 1$ and $SF[I[i]] = 0$ (i flipped 1, and i's successor's flip was 0), then i removes itself from the linked list by performing a doubly-linked list-remove using S and P. Get i's predecessor's rank.
if this is the last iteration of the loop, send our count of remaining elements to node 0 (doing this step now saves a BARRIER SYNCHRONIZATION).
BARRIER SYNCHRONIZATION
for each element i removed in the previous phase, look at the received rank of its predecessor, and increment its predecessor's rank $R[i]$ by i's current rank. (Barrier synchronization is not needed, as this can be done in parallel with the flip generation of the next iteration, or in parallel with the first phase of the step below.)
Major step 2: Processor 0 finishes the list reduction locally.
Node 0 uses the counts of remaining elements send by the other processors to perform a local prefix sum, and sets up temporary arrays to hold all of the remaining elements.
Node 0 tells each processor the offset to use for sending its remaining elements.
BARRIER SYNCHRONIZATION
All processors send the data for their currently-active elements (the predecessor pointers, the current ranks, and an appropriate indirection array) to processor 0.
BARRIER SYNCHRONIZATION
Processor 0 performs a local list-rank on the remaining active elements and puts the final ranks for these remaining active elements in their designated locations.
BARRIER SYNCHRONIZATION
Major step 3: Perform Major step 1 in reverse, inserting elements back into the list and patching things up.

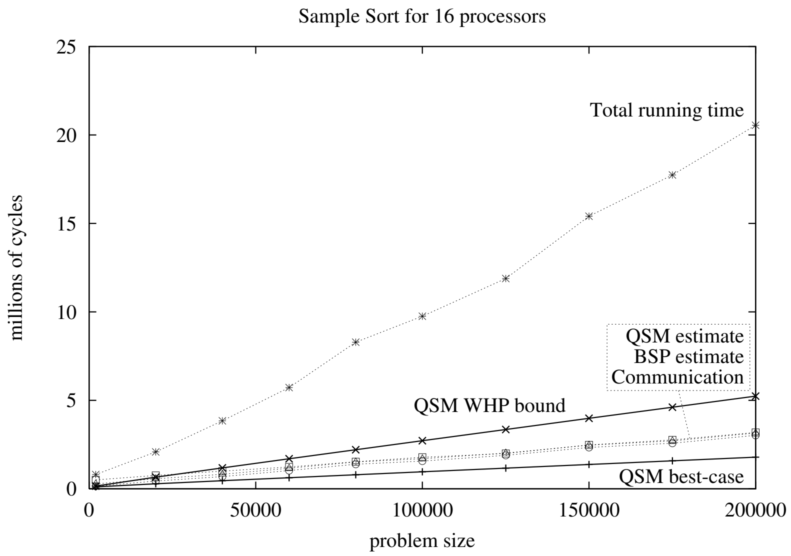


(a) Total running time and communication time.

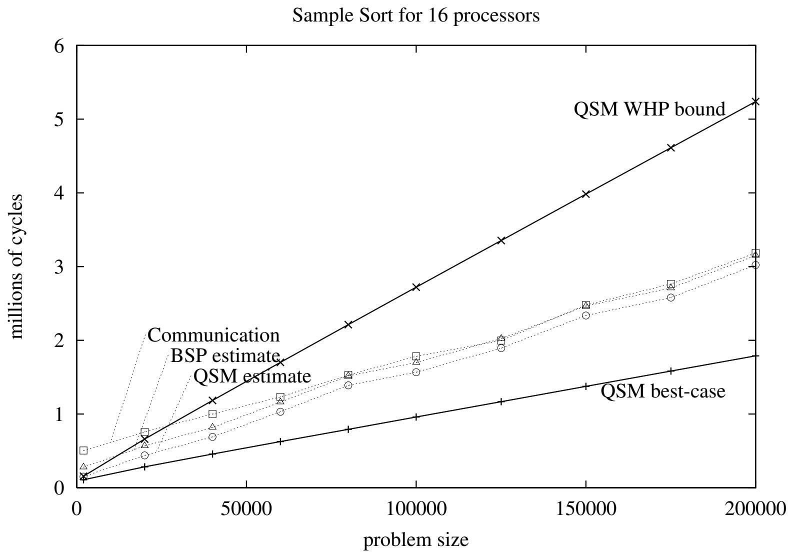


(b) Communication time.

Figure 8: Measured and predicted performance for the prefix sums algorithm on 16 processors.

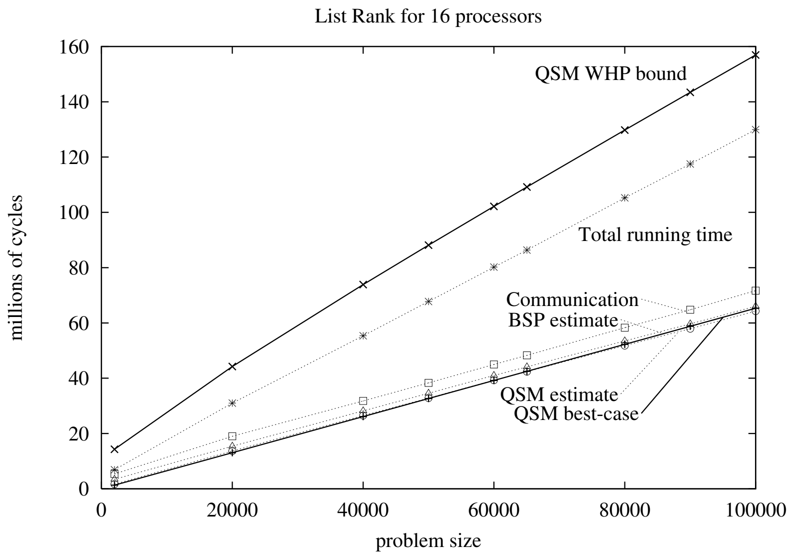


(a) Total running time and communication time.

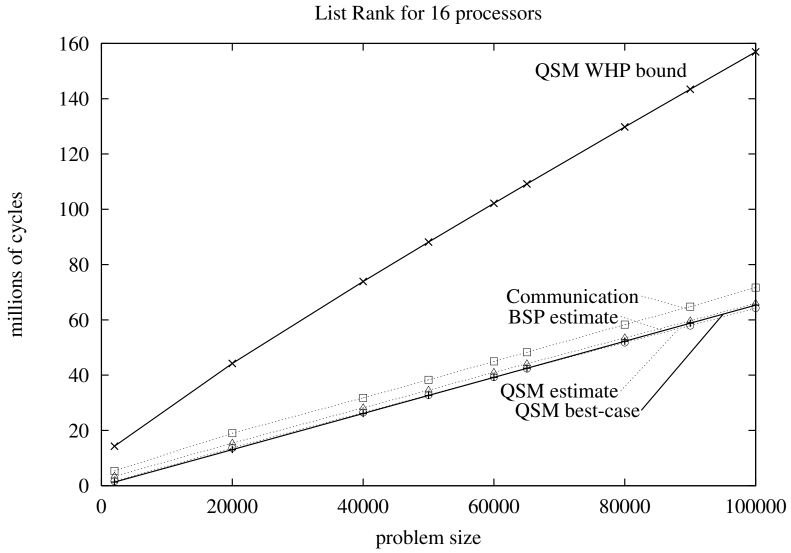


(b) Communication time.

Figure 9: Measured and predicted performance for the sample sort algorithm on 16 processors.

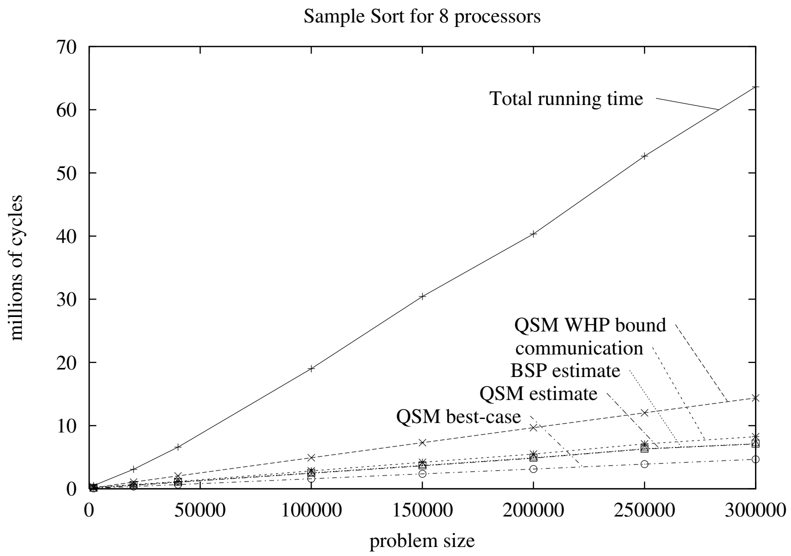


(a) Total running time and communication time.

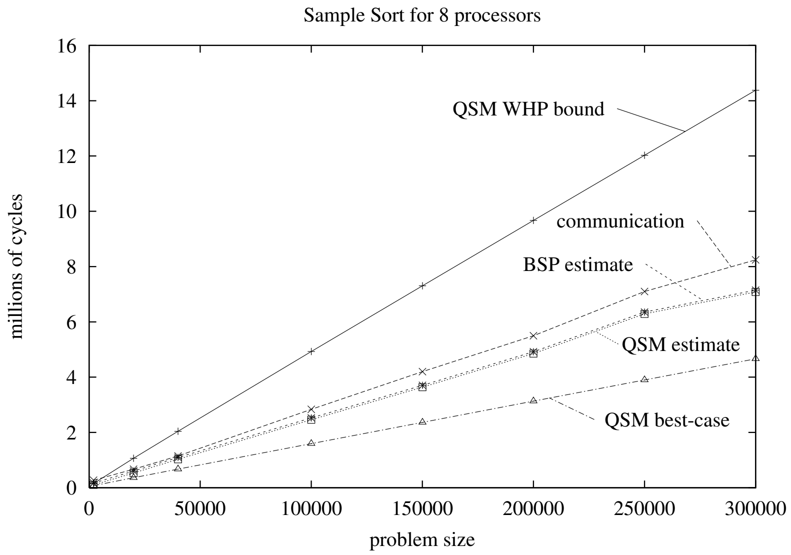


(b) Communication time.

Figure 10: Measured and predicted performance for the list ranking algorithm on 16 processors.

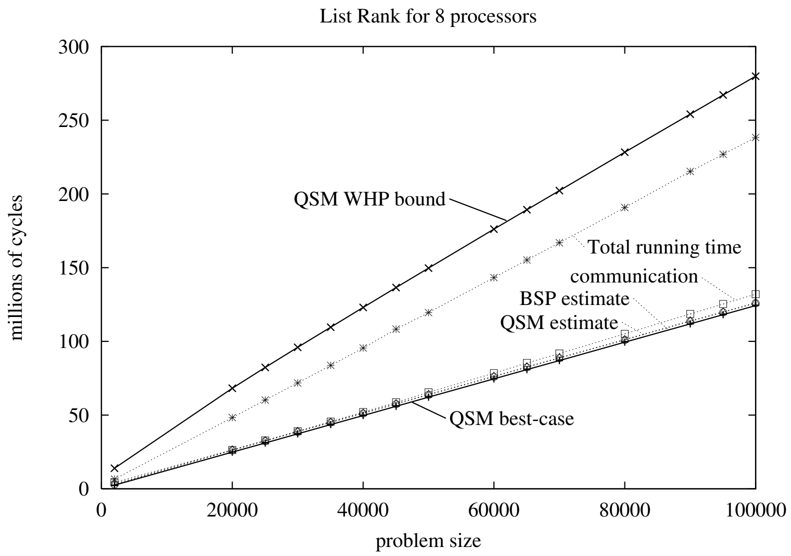


(a) Total running time and communication time.

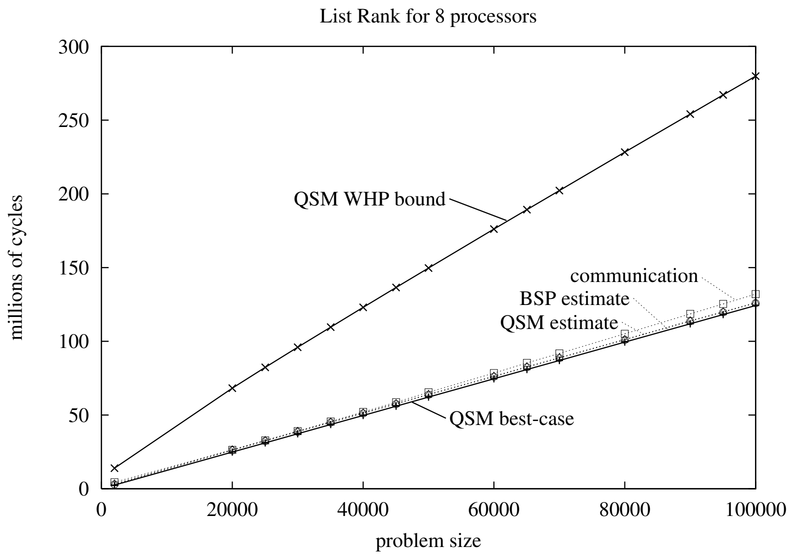


(b) Communication time.

Figure 11: Measured and predicted performance for the sample sort algorithm on 8 processors.

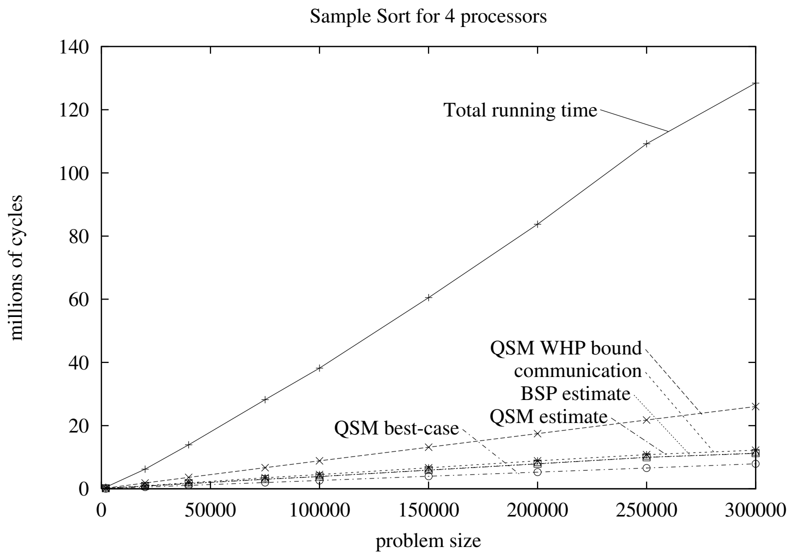


(a) Total running time and communication time.

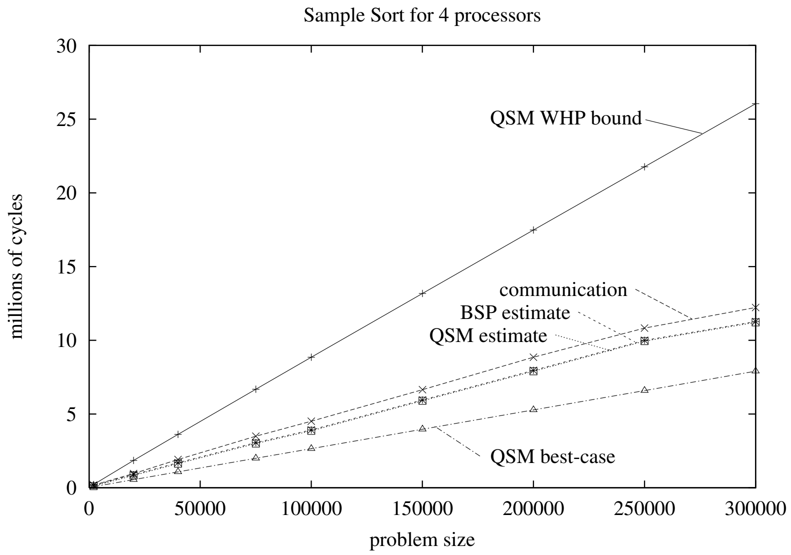


(b) Communication time.

Figure 12: Measured and predicted performance for the list ranking algorithm on 8 processors.

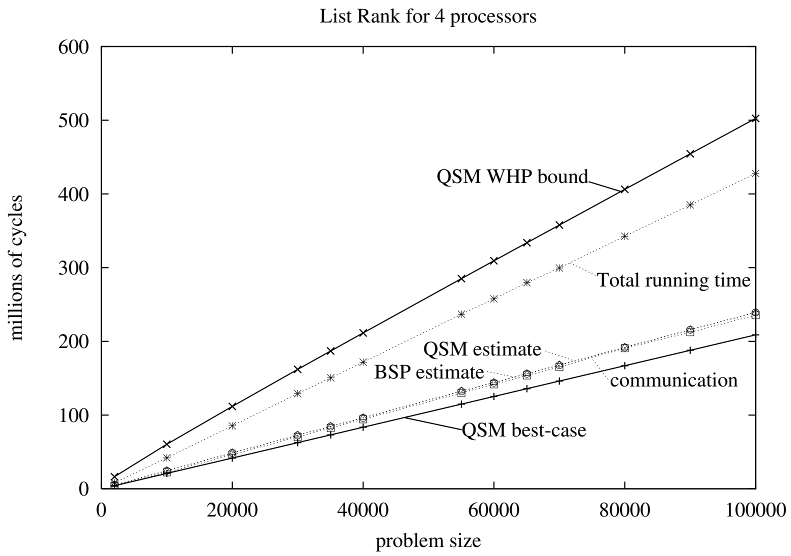


(a) Total running time and communication time.

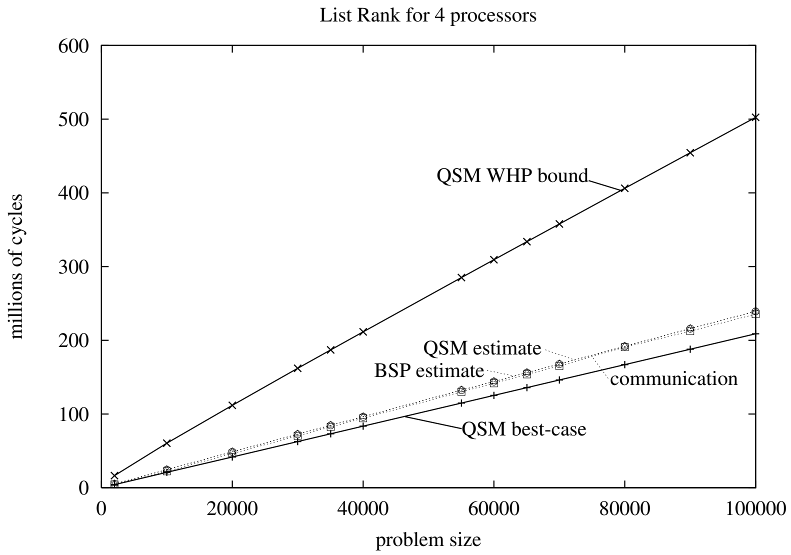


(b) Communication time.

Figure 13: Measured and predicted performance for the sample sort algorithm on 4 processors.



(a) Total running time and communication time.



(b) Communication time.

Figure 14: Measured and predicted performance for the list ranking algorithm on 4 processors.