Experimental Evaluation of QSM, a Simple Shared-Memory Model[*]

Brian Grayson, Michael Dahlin, and Vijaya Ramachandran
University of Texas at Austin
bgrayson@ece.utexas.edu, dahlin@cs.utexas.edu, vlr@cs.utexas.edu

UTCS Technical Report TR98-21

November 22, 1998

Abstract

Parallel programming models should attempt to satisfy two conflicting goals. On one hand, they should hide architectural details so that algorithm designers can write simple, portable programs. On the other hand, models must expose architectural details so that designers can evaluate and optimize the performance of their algorithms. Using both microbenchmarks and several representative algorithms, we experimentally examine the trade-offs made by a simple shared-memory model, QSM, to address this dilemma. The results indicate that analysis under the QSM model yields quite accurate results for reasonable input sizes and that algorithms developed under QSM achieve performance close to that obtainable through more complex models, such as BSP and LogP.

1 Introduction

A key goal of parallel language, compiler, and architecture designers is to support a programming model in which programmers and algorithm designers write high level descriptions of their algorithms that are then compiled into code optimized for different architectures. Designing a programming model to support that goal is challenging. On one hand, if the model is too abstract, it may hide important aspects of parallel architectures and cause algorithm designers to make poor design decisions. On the other hand, if the model is too detailed, it may complicate the programmer's task, and it may drive the programmer to write unportable code that optimizes performance on one architecture while making it hard for the compiler to optimize performance on other architectures. One step in resolving this dilemma is to develop a contract between programmers and compilers that specifies which architectural details should be explicitly handled in the high-level, architecture-neutral specification of an algorithm and which should be handled by its low-level architecture-specific implementation. This paper examines the trade-offs made by the Queuing Shared Memory (QSM) model [11]. Earlier theoretical analyses have suggested that despite the model's simplicity, it provides a good basis for designing high-performance algorithms. This paper takes an experimental approach to understanding under what conditions this model will yield good results.

[*] This work was supported in part by an NSF CISE grant (CDA-9624082) and grants from Intel, Novell, and Sun. Dahlin was also supported by an NSF CAREER grant (9733842). Grayson was also supported in part by an NSF Graduate Fellowship.

The QSM model provides a simple shared memory abstraction that attempts to reveal the most important aspects of parallel architectures to algorithm designers while hiding architectural details that have secondary performance impact and that interfere with portability. QSM provides a shared memory abstraction to simplify algorithm description and analysis, it models local memory and limited remote memory bandwidth to encourage locality, and it uses a bulk synchronous style to give the compiler[1] freedom to reorder, pipeline, and group messages to hide latency and per-message overhead. On the one hand, the QSM can be considered a more realistic version of the PRAM [9], since (1) it is shared-memory, (2) it models bandwidth limitations, and (3) it supports bulk-synchrony, thus avoiding excessive synchronizations. On the other hand, the QSM can be viewed as a simplification of more detailed distributed memory models such as BSP [21] and LogP [7] since it does not deal with the details of data layout, and it has a smaller number of parameters than these models. The theoretical results in [11] suggest that algorithms designed on the QSM should perform just as well on the BSP (to within a small constant factor) provided the input size is sufficiently large.

In this paper we use both simulation and measurements of actual parallel hardware to examine how well QSM tracks machine behavior in practice. In particular, we experimentally examine several ways in which QSM simplifies actual architectures to see if these simplifications are as benign as theory suggests. We examine QSM's decision to omit latency (l) and overhead (o) parameters by examining the behavior of several representative programs and find that, as predicted by theory, *programs written in a bulk-synchronous style are insensitive to network latency and overhead* as long as input sizes are large enough to permit sufficient pipelining and batching of messages. For the architectures and programs we examine, experiments suggest that this condition is achieved for essentially any problem size worth parallelizing. Finally, by examining microbenchmarks on an SMP (a Sun Enterprise 5000), a network of workstations (a cluster of Sun Ultra-1 workstations), and an MPP (a Cray T3E), we evaluate QSM's strategy of using randomization to avoid memory bank conflicts. We find that compared to a perfect memory layout with no contention, the random layout assumed by QSM does exhibit noticeable contention, but the contention appears tolerable even for these memory-intensive workloads, and randomization avoids the worst-case contention behavior when performance is much worse than the ideal layout.

The next section of this paper provides more details of the QSM model and discusses the contract it implies between programmer and compiler. Section 3 examines the performance of several representative algorithms running on a simulator that lets us vary network performance to determine the impact of omitting network latency and overhead parameters from QSM. Section 4 uses a synthetic benchmark on several actual machines to quantify the impact of omitting memory bank contention from the model. Section 5 surveys related work, and Section 6 summarizes our conclusions.

2 QSM Model

The Queuing Shared Memory (QSM) model [11] provides a simple shared memory abstraction that attempts to reveal the most important aspects of parallel architectures to algorithm designers while hiding architectural details that have secondary performance impact and that interfere with portability. A QSM 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

[1] In this paper, we use the term *compiler* in a broad sense to refer to the entity that translates an architecture-neutral program description into an optimized, architecture-specific implementation. This entity may be a human, library, or a program. In any case, the goal of our model is to make this translation a simple, mechanical process.

Architectural/Algorithmic Parameter	**Implementation contract**
Explicitly Modeled Factors	
p (number of processors)	QSM Parameter
g (gap)	QSM Parameter
κ (memory object contention) m_{op} (# of local operations) m_{rw} (# of remote operations)	Algorithm designer should minimize max($m_{op}, g \cdot m_{rw}, \kappa$)
Secondary Factors	
l (latency), L (barrier time)	Hide latency by pipelining
o (overhead of sending messages)	Use bulk synchronous style Minimize overhead by batching messages
h_r (memory bank contention)	Minimize contention by randomizing data layout
c (network congestion)	Use bulk synchronous style Limit contention by limiting network send rate

Table 1: QSM partitions architectural and algorithmic considerations into two categories: those that should be explicitly considered by the algorithm designer and those that should be handled by the low-level implementation.

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. This bulk synchronous model simplifies the analysis of algorithms as well as the translation of QSM descriptions into efficient architecture-specific implementations.

Table 1 summarizes a set of parameters that may affect the performance of parallel programs and indicates how a QSM programmer would account for those parameters. QSM essentially divides these parameters into two groups. First, the QSM performance model explicitly accounts for p, g, κ, m_{op}, and m_{rw}. These parameters represent fundamental characteristics of an algorithm on nearly any parallel architecture — p, the number of processors, represents the algorithm's concurrency, m_{rw}, the number of remote memory accesses, represents its locality (or lack thereof), and m_{op}, the number of local operations, represents its local computation time. The parameter κ represents the contention to any one remote memory object, which is fundamental to an algorithm because such contention cannot be hidden by, for instance, clever layout of data across banks. The key architectural parameter modeled by QSM is the gap, g, between the local instruction rate and the remote communication rate. This parameter reflects the limited communication bandwidth of most parallel architectures and thus encourages algorithms to exploit locality. If during a phase, the maximum number of local operations performed by any processor is m_{op}, the maximum number of remote reads or remote writes by any processor is m_{rw}, and the maximum number of reads *or* writes to any remote memory location during a phase is κ, QSM charges a time cost for that phase of max(m_{op}, $g \cdot m_{rw}$, κ). A related model, the s-QSM (symmetric QSM) charges a time cost of max (m_{op}, $g \cdot m_{rw}$, $g \cdot \kappa$).

QSM considers the second group of parameters in Table 1 — l, o, h_r, and c — to be secondary factors in algorithm design and contends that algorithm descriptions and analysis may generally be simplified by ignoring these factors. In practice, parallel programs reduce the impact of these factors using standard techniques: pipelining to hide latency, batching requests to reduce overhead, and randomization to avoid bank conflicts. Rather than complicate high-level, architecture-independent algorithm descriptions with these routine details, QSM assumes a contract in which the compiler is responsible for using such techniques when appropriate. In particular:

- When designing a QSM algorithm, a designer may ignore network latency (l) because she may assume that the low-level implementation will hide latency by pipelining requests. QSM's bulk-synchronous model facilitates this simplification by creating batches of requests that may be sent during a phase but that will not be used until the next phase. The QSM model thus predicts that l will not affect running time as long as the problem is relatively large. For instance, this condition holds if $(l/g) \cdot \pi << W/p$, where W is the amount of communication done by the algorithm, p is the number of processors in the target machine, and π is the number of phases in the QSM algorithm [19]. It also holds true if a QSM algorithm designed for p processors is mapped onto a p' processor machine where $(l/g) \cdot p' << p$ [11]. In our experiments, we find that in practice data sets large enough to be worth parallelizing easily meet these criteria for the algorithms and architectures we examine. Synchronization time, L, also increases with increasing latency (under the LogP model [7], synchronization takes $\frac{l/g \log p}{\log l/g}$), and QSM expects synchronization time to become insignificant under similar conditions.

- When designing a QSM algorithm, a designer does not explicitly account for o, the overhead of sending and receiving a message. Instead, the designer assumes that the compiler will take advantage of bulk synchrony to batch requests and thereby minimize overhead. By including g but not o in the network performance model, QSM tells algorithm designers to focus on limiting the amount of data sent by an algorithm, not on how many messages are used to send that data.

- When designing a QSM algorithm, a designer does not account for the contention of remote memory accesses to banks (h_r) except when there are many accesses to a specific remote object (κ). Instead, the designer assumes that the compiler will limit the performance impact of bank conflicts by randomizing data layout, for example by hashing remote memory addresses in hardware or software [11]. Three aspects of this model should be noted. First, randomization will not reduce conflicts when the conflicting accesses are to a single memory address, so QSM explicitly accounts for such hot-spot object conflicts with its κ parameter. Second, this aspect of the implementation contract should not be construed as indicating that QSM does not account for careful memory layout that improves locality; QSM's g parameter encourages algorithms to move data to their local memories when possible. Finally, the natural description of many algorithms provides a balanced or randomized data layout without requiring randomization from the implementation layer; in such cases, as a performance optimization the algorithm description should inform the compiler that it may safely omit randomization.

- When designing a QSM algorithm, a designer does not explicitly account for c, the network congestion. Brewer and Kuszmal [4] found that network congestion could significantly limit the performance of parallel machines. QSM expects compilers to address congestion in two ways, both based on Brewer and Kuszmal's techniques. First, the periodic synchronizations associated with a bulk-synchronous programming style can reduce congestion. Second, QSM expects compilers to limit the rate at which nodes send data so that they do not overrun receiving nodes and cause congestion in the network.

2.1 Comparison with other parallel architecture models

It is worthwhile to compare the QSM model to other popular models for parallel algorithm design. The traditional model is the PRAM [15] which is a synchronous shared-memory model with unit-time communication to shared-memory; different variants of this model restrict memory accesses to be *exclusive* or unit-time *concurrent*. While the PRAM is a simple model that aids in exposing high-level parallelism in algorithms, its cost measure has a significant mismatch to real machines in that it ignores issues of latency, bandwidth limitation, and memory granularity in parallel machines. As in the QSM, the latency mismatch can be addressed by pipelining if sufficient parallel slackness is present, but the synchronous nature of the PRAM model typically results in a larger number of phases in a PRAM algorithm for a given problem than in a QSM algorithm, and thus results in larger latency and synchronization costs than in the QSM. Also, the PRAM has no parameter to model bandwidth limitation, and hence the model does not encourage locality of reference. As in the QSM, the memory granularity issue can be addressed by hashing, provided the *exclusive* (*e.g.*, EREW) and not concurrent (*e.g.*, CRCW) memory access rule is used, but the exclusive memory access rule is more restrictive than the queuing memory access used in the QSM.

The BSP (*Bulk Synchronous Parallel*) [21] and the LogP [7] models each model a parallel machine as a collection of processor-memory units with no global shared memory. The processors are interconnected by a network whose performance is characterized by a gap parameter g and a latency parameter l (in LogP) or synchronization parameter L (in BSP). The LogP model also models the per-message overhead o for sending and receiving messages, and it limits network congestion by requiring that no more than l/g messages be in transit to a given destination processor in any interval of length l. There have been several algorithms designed and analyzed on the BSP and LogP models and their extensions (see, *e.g.*, [1, 3, 10, 14, 16, 23]). These algorithms tend to have rather complicated performance analyses, because of the number of parameters in the model as well as the need to keep track of the exact memory partition across the processors at each step.

In contrast to the BSP and LogP models, the QSM has only two architectural parameters—p and g—and it is a shared-memory model. This latter point is of importance since shared-memory has been a widely-supported abstraction in parallel programming [17], and additionally, the architectures of many parallel machines are either intrinsically shared-memory or support it using suitable hardware or software. Further, as indicated earlier, the shared-memory of the QSM can be hashed onto the distributed memory, and this strategy gives provably good performance on the BSP [11]. It is interesting to note that there are BSP algorithms for irregular problems that achieve good performance by randomly distributing elements across processors (see, *e.g.*, [3] for the multi-search problem).

In some special cases the QSM abstraction may not reveal the full power of a specific parallel architecture. In particular, algorithms that make use of fine-grained synchronization are not a good match with QSM's bulk synchronous programming style. Also, all QSM communication takes place through shared memory and all synchronization occurs at the end of phases, which is a simpler but less powerful mechanism than communication to activate computation on remote nodes (*e.g.*, Active Messages [22]).

3 Impact of omitting l and o

The QSM model predicts that network latency l and per-message overhead o will not impact running time for bulk synchronous programs assuming that (1) the compiler or run time system pipelines and batches messages and (2) the problem is sufficiently large to provide enough parallelism for these techniques to be effective. In this section, we test these hypotheses by running several representative parallel programs on a detailed simulator that lets us vary network performance.

3.1 Methodology

3.1.1 Workloads

We evaluate the performance of QSM algorithms for three fundamental problems: *prefix sums* (a basic primitive for most parallel algorithms, with an algorithm that displays parallelism with very little communication), *sample sort* (an important algorithm with some communication), and *list ranking* (the canonical problem for evaluating performance of parallel algorithms with large amount of irregular communication). As suggested by the QSM model, we optimized these algorithms to minimize computation and communication time, while keeping the number of phases small [19]. Note that we focus on providing simple algorithms that will be effective for practical problem and machine sizes, so our algorithms often place a minimum size on the problem size per processor. The running times are presented for the s-QSM, which assumes that the same gap parameter is encountered at processors and at memory.

This section summarizes the algorithms. More detailed descriptions of these algorithms can be found in the appendix.

Prefix Sums. The p-processor QSM prefix sums algorithm runs in $O(gn/p)$ time with just one synchronization when $p \leq \sqrt{n}$. Each node calculates the sum of its local elements, and broadcasts it to the remaining processors. Each processor then computes the offset for its elements, and follows that up with a computation of the correct prefix sums for the positions corresponding to its local elements. If the input is initially distributed evenly across the processors, the running time is $O(\frac{n}{p} + gp)$.

Sample Sort. The p-processor QSM sample sort algorithm is a simple one that runs in time $O(gp \log n + \frac{gn}{p})$ and 5 phases with high probability (*whp*) when $p \leq \sqrt{n/\log n}$. The algorithm uses over-sampling: it picks $c \log n$ random samples per processor for some constant c, sorts the $cp \log n$ samples and then picks a total of p pivots by using every $(c \log n)$th element in the sorted list of samples. The ith processor then sorts the elements in the ith 'bucket.'

List Ranking. The list-ranking algorithm we implemented is a randomized one that, on a p-processor QSM, runs in time $O(gn/p)$ time with $O(\log p)$ phases *whp*. This algorithm assigns each processor a random block of n/p elements, and in each phase the algorithm assigns each element a random bit. During a phase each processor eliminates those elements assigned to it whose random bit is 0 and whose successor's random bit is a 1. When the number of remaining elements is reduced to $O(n/p)$ all of the elements are sent to processor 0, which then completes the forward computation using a sequential list-ranking algorithm in time $O(n/p)$. A corresponding expansion phase then computes the list ranks of the eliminated elements within the same time bounds.

For all experiments, we ran each experiment 10 times and report the average. The standard deviation is less than 11% of the average for all of the sample sort runs, and less than 2% for all but the smallest problems sizes for the parallel prefix and list rank runs.

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.

3.1.2 Architecture

The Armadillo multiprocessor simulator [12] 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 `get`s and `put`s 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

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.

a high-performance communications interface such as Active Messages [22] and high-performance network hardware such as Myrinet [18]. 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. In Section 3.3 we will describe our experiments that vary these hardware parameters to examine the algorithms' sensitivity to them.

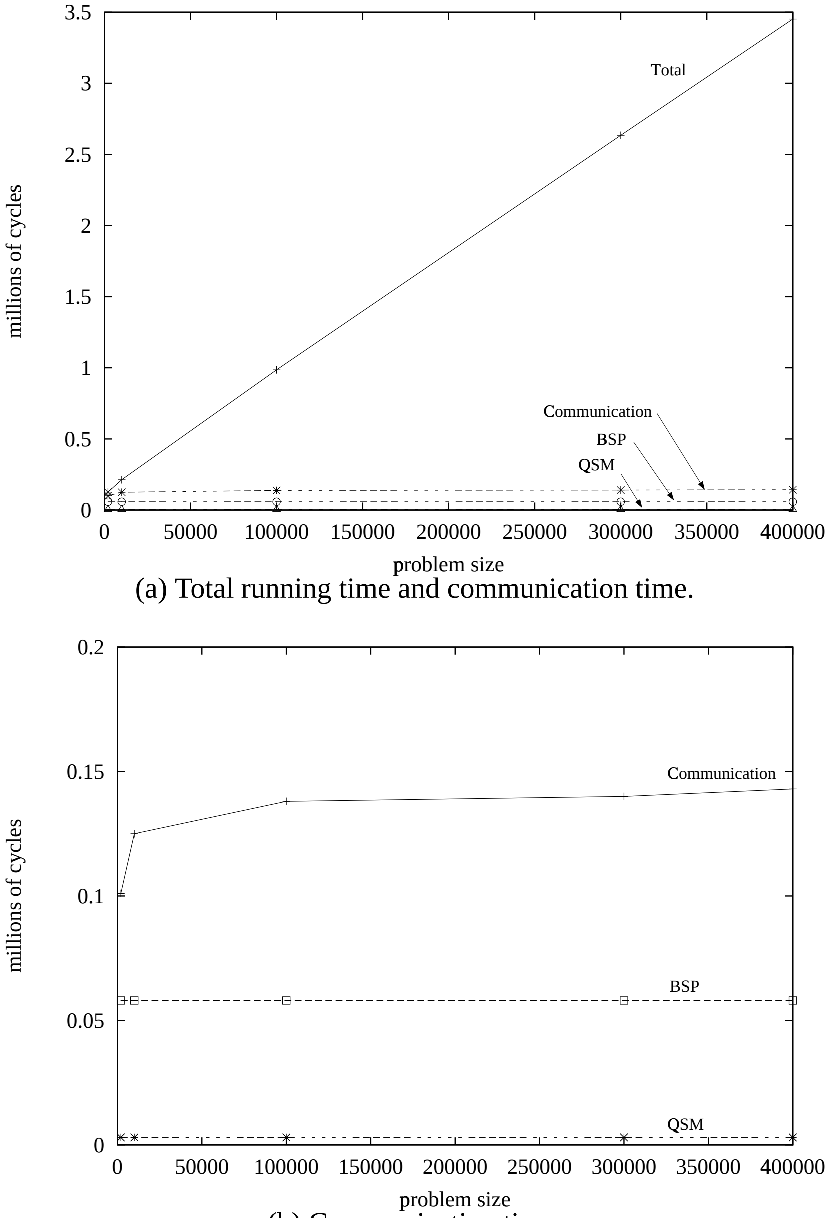
3.2 Results

Theory suggests that the bulk synchronous model will allow QSM analysis to safely ignore latency as long as there is sufficient parallelism to hide it by pipelining requests. In particular, it suggests that latency will be dominated by other factors when $(l/g) \cdot \pi << W/p$ where W is the amount of communication, p is the number of processors in the target machine, and π is the number of phases in the QSM algorithm. For our default system, l is 1600, g is 3, and p is 16. For the algorithms we examine, π ranges from 1 for prefix sum to 4 for sample sort to (4 + 16 log p) (which is about 68 for our default 16-node machine) for list ranking, and for the algorithms we examine W is linear with n. Thus, we would expect l to be hidden and QSM to predict performance for problem sizes where $\frac{n}{p}$ is larger than some constant times 37,000 for this system. Assuming that the constant hidden by the $O()$ notation is small, this analysis suggests that l will not significantly impact performance for problem sizes large enough to be worth parallelizing. Similarly, QSM analysis does not account for per-message overhead because it assumes that overhead will be amortized by batching requests.

Figures 1, 2, and 3 summarize the results of a set of experiments designed to test this hypothesis. In each figure we show the measured results of running one of the algorithms and compare the measured communication time to the communication time predicted by QSM and the more detailed BSP model. For all of these experiments, we find that QSM predicts communication performance well when n is reasonably large.

We focus on predicting communication performance rather than total running time for two reasons. First, all of the models abstract local computation in the same way, so comparisons of how the algorithms predict local computation will not be interesting. Second, for all of the models calculating appropriate constants for an algorithm on a particular architecture is nontrivial; imprecision at this step might overshadow the effects we wish to examine.

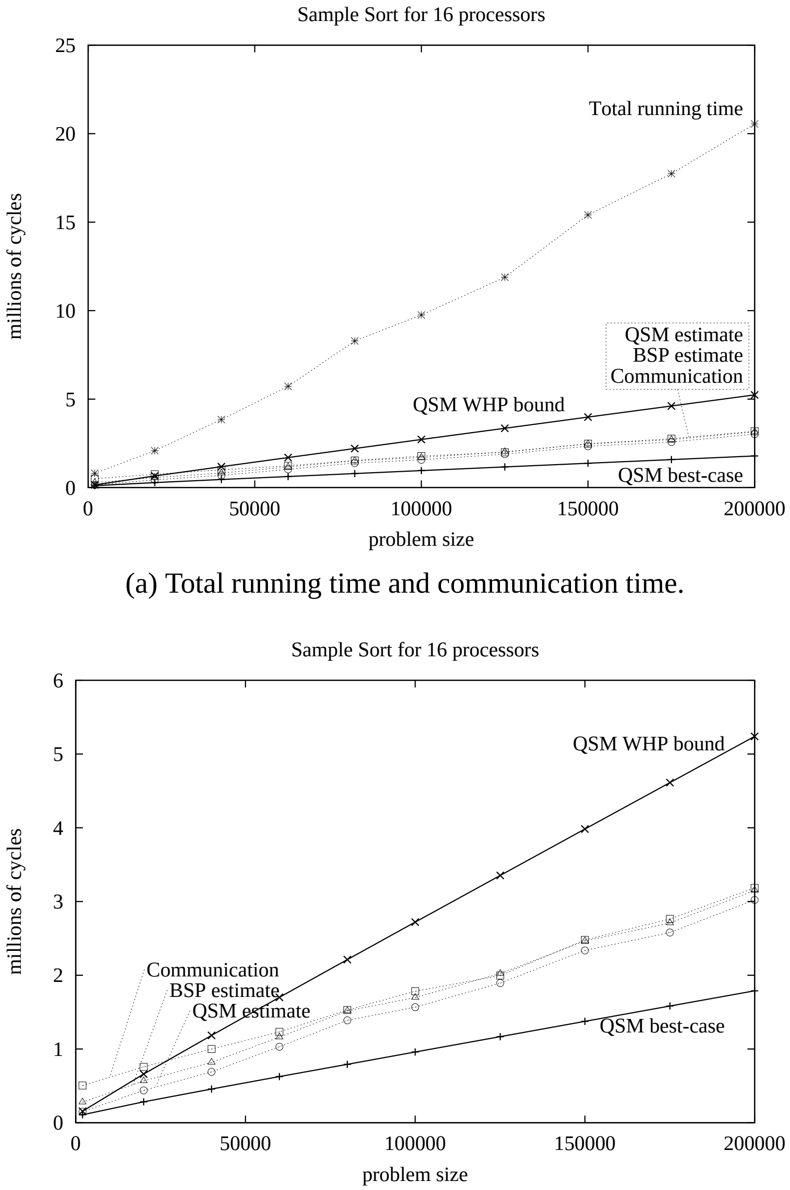
Prefix. Figure 1 shows the predicted and actual performance of the parallel prefix algorithm. A QSM analysis of the parallel prefix algorithm we implemented predicts that communication will take time $g(p-1)$.



(a) Total running time and communication time.

(b) Communication time.

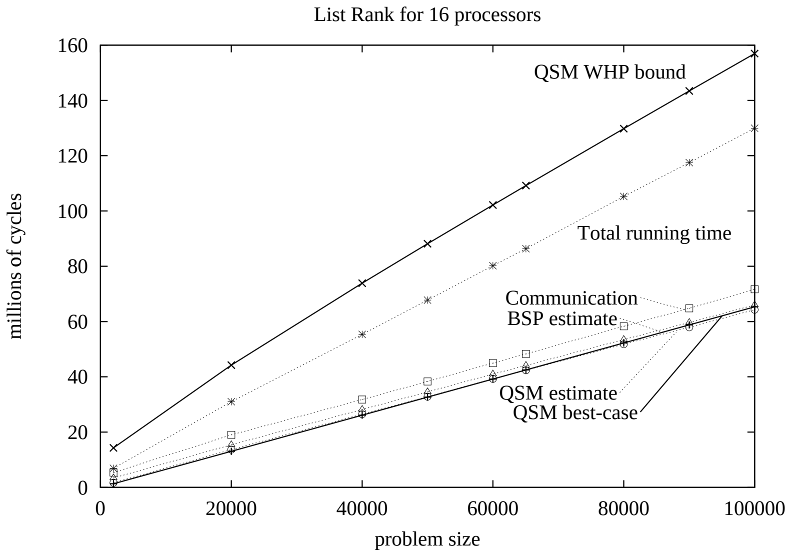
Figure 1: Measured and predicted performance for the prefix sums algorithm.



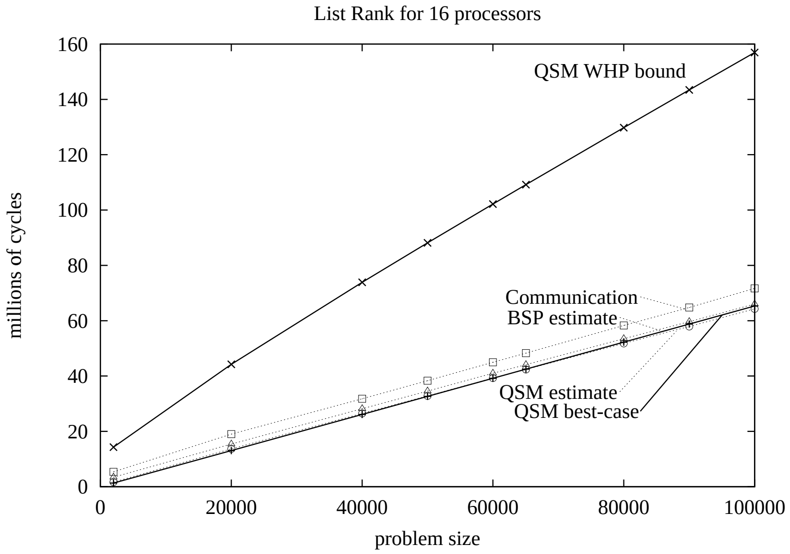
(a) Total running time and communication time.

(b) Communication time.

Figure 2: Measured and predicted performance for the sample sort algorithm.



(a) Total running time and communication time.



(b) Communication time.

Figure 3: Measured and predicted performance for the list ranking algorithm.

A BSP analysis includes the per-phase synchronization cost, thus adding a $1 * L$ term to that prediction.

Parallel prefix is an example where QSM and BSP models lead to good algorithm design but where they do not allow accurate prediction of communication time. Both the QSM and BSP analysis significantly underestimate communication time because they assume a bulk-synchronous model where communication overhead, o, is hidden by large messages; QSM's estimate is significantly lower than BSP's because it also ignores latency. For this algorithm, messages are small and communication time is dominated by overhead and latency, so these models do a poor job in predicting overall performance. In addition, the amount of communication does not increase as problem size increases. Note that although the relative error is large because communication time is tiny, the absolute error is still small, and the overall algorithm is still efficient in practice.

Sample Sort. For the sample sort algorithm in Figure 2, a QSM analysis predicts that communication will take time $4(p-1)g \log n + 3(p-1)g + gBr + gB$ *whp*. The algorithm is randomized, and the B and r terms represent how running time depends on the load balance achieved. 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 the bucket. In the figure, we plot three cases. In the best case $B = \frac{n}{p}$ and $r = \frac{p-1}{p}$, and all nodes have equal amounts of work to do. The *Best case* line shows this unreasonably optimistic case. By applying Chernoff bounds on B and r, we derived bounds for the algorithm's running time that hold for at least 90% of runs. The details of this derivation can be found elsewhere [19]. The *WHP bound* line shows this as an upper limit on typical performance. Finally, we experimentally measured the actual B and r skews experienced in each experiment and plot the resulting line as *QSM estimate.* This third line represents the type of performance estimate that could be achieved under a QSM model if either (a) an algorithm were oblivious and deterministic and an exact time bound were known or (b) a detailed analysis of probability distributions were available.

The *BSP estimate* line of the graph shows the results of a BSP analysis of the algorithm using the actual skews determined experimentally. BSP analysis includes the per-phase synchronization cost, for an additional $5L$ term over the QSM analysis. The best-case and upper-bound load balance analysis for BSP is the same as for QSM, and plots for these are omitted from the graph for clarity; they would be offset from the QSM lines by the same $5L$ term as the *BSP estimate* line.

For the case where load balance is known with precision, the simple QSM model successfully predicts communication performance when problem sizes are relatively large. By ignoring the cost of per-message overhead and network latency, QSM underestimates communication time by a constant amount. However, as problem size grows, this error becomes less important and the predictions become more accurate. Accuracies within 10% of the communication time are achieved for all problem sizes larger than about 125,000 elements total (or about 8,000 elements per processor.) (Note that because computation time represents a significant portion of running time, a 10% error predicting communication time translates into a much smaller error in predicting total running time.) We believe that problems smaller than this limit are unlikely to be worth parallelizing, so model inaccuracy for such problem sizes is not a large concern.

For the case where load balance is not known with precision, the *Best-case* and *WHP bound* lines in the graph bound predicted performance. Note that the slopes of the lines differ because a imbalance from the B and r terms means that some processor could be doing more communication than the average processor. The two lines bound actual performance over almost the entire range of problem sizes. Again, mismatches happen when problem sizes are probably too small to be worth parallelizing. This analysis suggests that the looseness of the bounds obtained using standard algorithm analysis and variations introduced by randomization may often be larger than the errors introduced by QSM's simplified network model.

List Ranking. Figure 3 shows the predicted and actual performance of the list ranking algorithm. The running time for the QSM list ranking algorithm is

$$\pi g(\frac{c_1}{2} + \frac{7c_2}{4}) \sum_{i=1}^{4\log p} x_i + 4\pi' gz$$

where x_i is the maximum number of elements at any processor in the ith phase, with $x_1 = \frac{n}{p}$, z is a bound on the number of elements sent to processor 0 for the sequential computation phase, 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 ith iteration, 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, 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.

In the unrealizable ideal case, we assume the randomized steps cause no skew in the work on the different processors. In that case, $x_i = \frac{n}{p}(\frac{3}{4})^{i-1}$, $z = n \cdot (\frac{3}{4})^{4\log p}$, $c_1 = c_2 = 1$, and $\pi = \pi' = \frac{p-1}{p}$. This is the *Best case* line in the graph. To obtain a bound on a running time that holds with probability at least 0.9, we used Chernoff bounds to obtain bounds on x_i, z, c_1, and c_2. As for our analysis of sample sort, these bounds are likely to be quite conservative. The resulting bound on expected performance is the *WHP bound* line in the figure. Finally, the *QSM estimate* line shows a calculation based on the actual problem-size compression achieved in each phase, and the *BSP estimate* line also corresponds to the actual work at each processor. BSP's lines for the ideal case and *whp* limit are not shown but would be offset from their QSM counterparts by a similar amount.

As with the sample sort algorithm, as problem size increases, prediction accuracy improves, and the model predicts performance well for problem sizes worth parallelizing. In particular, the BSP prediction is within 15% of the actual communication time as long as $n \geq 40,000$ elements and the QSM prediction is within 15% of the actual communication time as long as $n \geq 60,000$ elements. Again note that because communication time is only a portion of the total running time, errors predicting total running time will be even smaller.

3.3 Sensitivity to architectural parameters

The QSM model predicts that l and o are effectively hidden when $\frac{W}{p}$ or $\frac{n}{p}$ is large enough to allow sufficient pipelining and batching of messages. We would therefore predict that systems with larger l, o, or p would also require larger n before QSM predictions are accurate. In fact, as the first paragraph of Section 3.2 suggests, we would predict a linear relationship between l, o, or p and the minimum n required for good prediction. In Figure 4, we vary l, the hardware latency, over a range of values and compare the measured performance against QSM's predictions. Notice that QSM's predictions do not account for latency and are thus constant as l is varied.

As hypothesized, increasing l results in a linear increase in the problem size n required for QSM to accurately predict performance. Figure 5 shows this more clearly by plotting the points where the *WHP bound* line crosses the measured performance lines in Figure 4. If our simulator could accommodate larger problem sizes, we would expect a similar linear relationship if we compared where predictions first come within 10% of measured performance for each value of l. Figure 6 shows the results for another experiment where

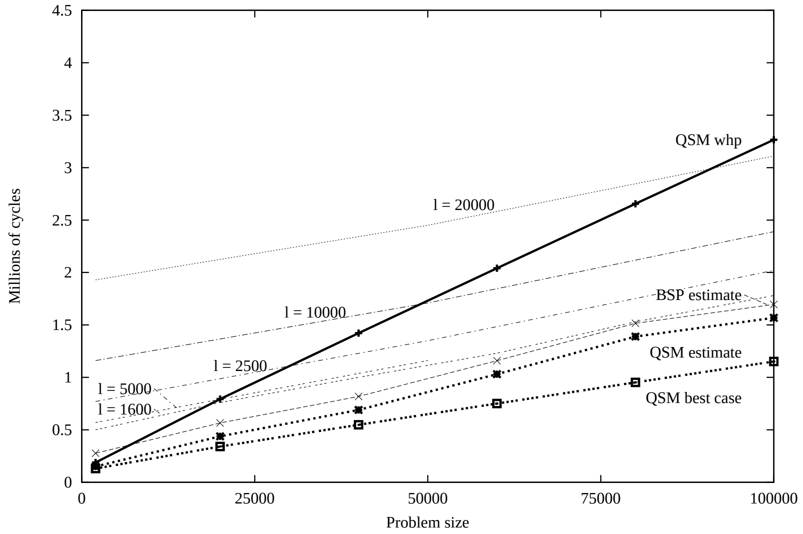


Figure 4: Measured communication performance vs. QSM predictions as latency is varied for sample sort.

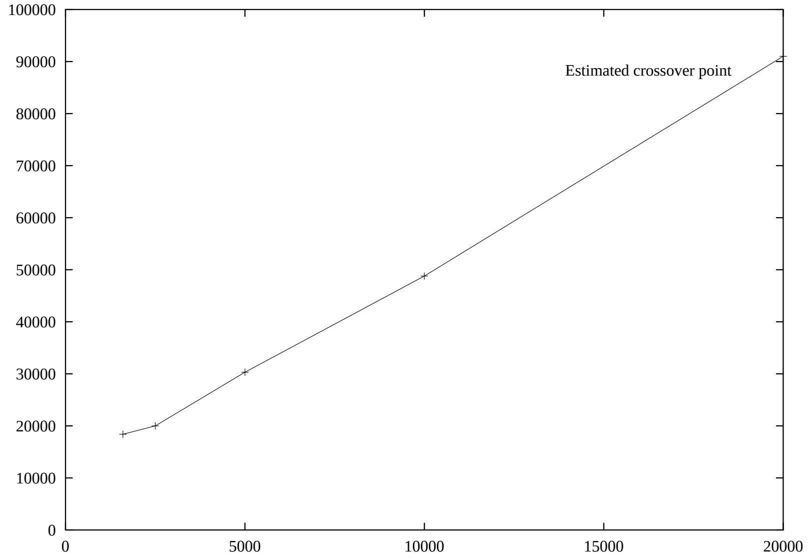


Figure 5: Problem size needed for actual communication time to fall within the range between the *WHP bound* and the *Best-case* lines as latency *l* is varied for sample sort.

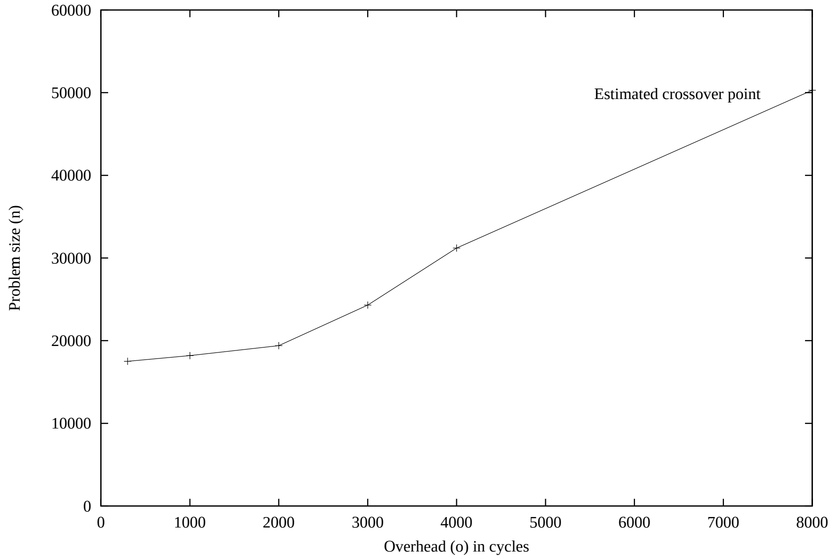


Figure 6: Problem size needed for actual communication time to fall within the range between the *WHP bound* and the *Best-case* lines as per-message overhead o is varied for sample sort.

we varied the machine's overhead, o. Due to memory limitations of our simulation infrastructure, we were not able to vary p over a wide enough range to examine this relationship for p.

These experiments suggest that QSM will predict communication performance of these algorithms for almost any reasonably sized problem. For example, in our default configuration QSM accurately predicts communication time for the sample sort algorithm when $n \geq 125,000$. On our 16-processor simulated machine, that corresponds to just 8000 elements per processor, which we believe is a small problem size for a modern machine with 64 MB or more of memory per processor.

The linear relationship between l, o, and p on the problem size needed for prediction accuracy suggests that we may be able to extrapolate from these results to predict when QSM will accurately model communication performance for other architectures. The predictions in Table 4 should be treated with caution since they represent an extrapolation from one set of experiments to a wide range of architectures. However, both the theoretical QSM model and our experimental results support this extrapolation. Even with these caveats, the data in this table suggest that QSM will predict performance well for this algorithm for modest sized problems.

4 Memory bank contention

QSM does not track how data are placed across global memory banks. QSM expects algorithms to maximize locality by utilizing local memory and to minimize remote-memory bank contention by randomizing data layout. This section examines how well that strategy will work in practice by examining the performance of a microbenchmark that was designed to stress the memory system of several modern parallel architectures.

Each processor running the microbenchmark accesses global memory as quickly as it can in one of three patterns. In the Random pattern, each access is to a random word in a random remote bank's memory.

Architecture	p	l	o	g	$\frac{n_{min}}{p}$
Default simulation parameters	16	1600	400	3	8000
Berkeley NOW [18]	32	830	481	4.3	(k * 4640)
300MHz Pentium-II TCP/IP, 100Mb Switched Ethernet	(32)	75000	150000	24	(k * 325000)
CRAY T3E [2]	(64)	126	(50)	1.6	(k * 1558)
Intel Paragon [8]	(64)	325	90	0.35	(k * 15429)
Meico CS-2 [8]	(32)	497	112	1.4	(k * 5325)

Table 4: The models examined in this paper predict that for problems larger than n_{min}, the QSM model should accurately predict running time for the Sample Sort benchmark. Most of the values for hardware parameters were taken from the articles specified above, after converting all parameters to be in units of clock cycles; values in parenthesis were not available in those articles and represent estimated values. Our estimates for n_{min} on the other architectures include the parameter k, which corresponds to differences in software implementation of communications primitives across the architectures.

This pattern represents the access pattern that a QSM runtime system would achieve by randomizing data layout. In the Conflict pattern, each access is to a random word in memory bank 0. It shows the case that might happen if an algorithm has hot spots and the run time system does not act to eliminate them. In the NoConflict pattern, each access by processor i is to a random word in memory bank $i + 1$ so that no two processors are accessing the same bank. This pattern represents a best case for remote memory access that might be achieved under a sophisticated algorithm developed under a more detailed model than QSM. In the results below, we report the average access time for a word when the shared memory array is too large to be cached in the machine's hardware processor cache.

We examine the performance of the microbenchmark on four systems that span a range of memory architectures.

- SMP-NATIVE is an 8-processor, 8-memory-bank Sun UltraEnterprise server. Each processor runs at 166 MHz, and each memory bank is 128 MB. The hardware distributes sequential 64-byte cache blocks to sequential memory banks. The benchmark shares memory using the cache consistent shared memory space provided by the hardware.

- SMP-BSPlib uses the same hardware, but the benchmark accesses shared memory using the shared memory subset of BSPlib version 1.3 [6]. We compiled the library to provide its global memory abstraction using SYSV shared memory. We show performance for both the "level-2" highly optimized library and the "level-1" less optimized version, and the microbenchmark uses the "high-performance" variants of the shared memory access functions that do less buffering than the standard functions.

- NOW-BSPlib uses a cluster of sixteen 166 MHz UltraSPARCs connected by a 10 Mbit/s ethernet. The benchmark uses the shared memory abstraction provided by the BSPlib runtime system, which uses TCP for global communication in this system. We show performance using the "level-2" optimized library and the "high-performance" shared memory access functions.

- Cray T3E uses 32 nodes of a 68 node Cray T3E. Processing elements are Digital Equipment Corporation EV5 RISC microprocessors, and the interconnect is a high-performance 3-D torus memory interconnect. We use the `shmem` shared memory library for data access to the shared array.

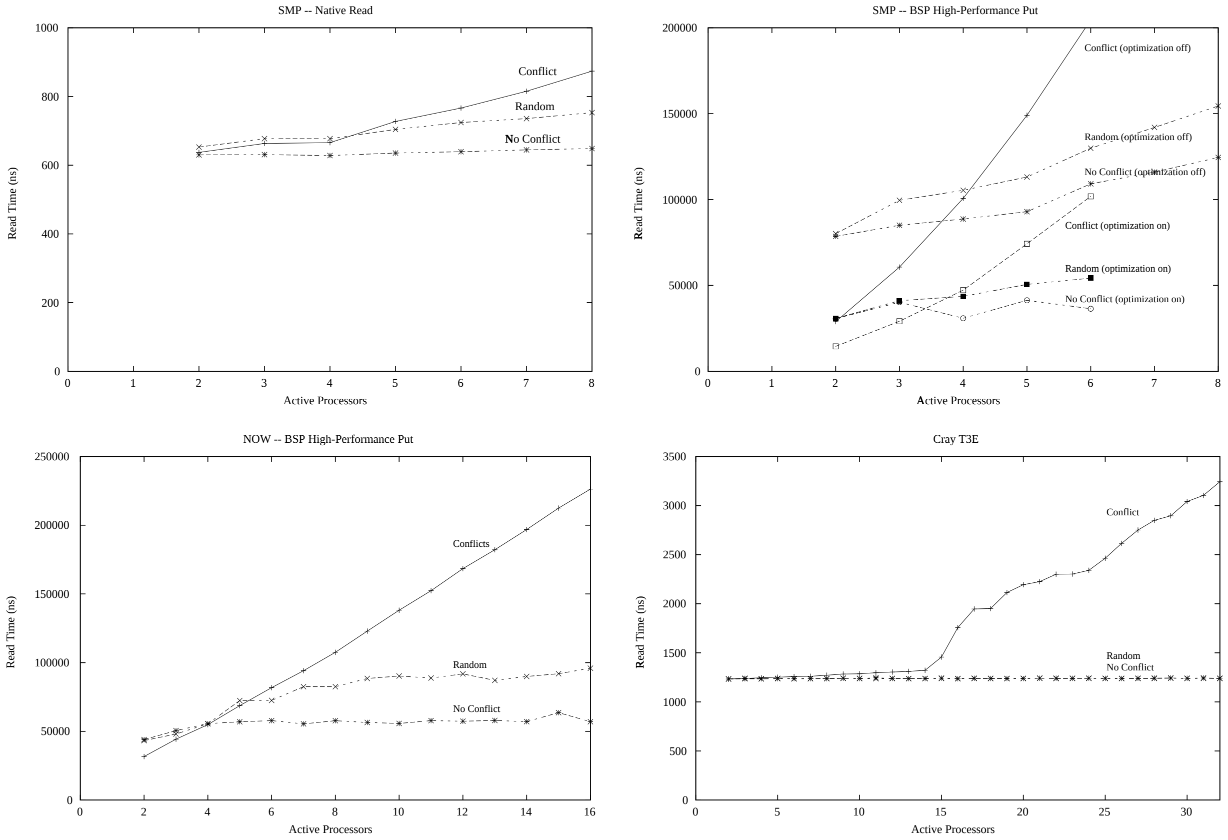


Figure 7: Remote memory access performance for the SMP, SMP-BSPlib, NOW-BSPlib, and Cray T3E architectures.

Figure 7 shows the performance of the benchmark on these architectures. The results conform to the assumptions of the QSM model. The careful memory layout of the NoConflict strategy performs modestly better than the Random approach with speedups of 0% to 68%. But randomization avoids the worst-case contention behavior seen in the Conflict cases when performance is generally a factor of two to four worse than the ideal NoConflict layout. Note that this microbenchmark was designed to stress test the memory systems' behavior under overload; access patterns for real programs may be less concurrent than shown here, and the performance differences among the patterns may be less pronounced than shown here.

5 Related work

Martin et. al [18] experimentally examined how the performance of parallel programs depended on the LogP parameters. They found the strongest dependency on per-message bandwidth (o) but less sensitivity to latency (l) and per-byte bandwidth (g). We found little sensitivity to per-message bandwidth for the problems we study. We believe this is because we assume a bulk synchronous model and assume that low-level compilers take care of details such as batching messages when possible.

Several studies have examined how different aspects of network performance affect program performance. Cypher et. al [5] examined the performance of several message passing scientific codes. Holt et. al [13] used simulation to examine the performance of the FLASH multiprocessor as its architectural parameters were varied and found that performance was heavily dependent on message latency and overhead. The Wisconsin Wind Tunnel was also built to examine the impact of different communication architectures on system performance [20]. A major difference between these studies and ours is that the workloads examined in these other studies do not generally follow a bulk-synchronous programming style. Although some of these other studies conclude that overhead and latency are important factors for performance for programs written under current programming models, our conclusions have a different focus: we conclude that it would be feasible to adopt a programming model in which l and o can be considered secondary factors.

6 Conclusions

A key goal of parallel language, compiler, and architecture designers is to support a programming model in which programmers and algorithm designers write high level descriptions of their algorithms that are then compiled into code optimized for different architectures. In this paper, we have experimentally evaluated whether the assumptions made by QSM are compatible with that goal. The results indicate that analysis under the QSM model yields quite accurate results for reasonable input sizes and that algorithms developed under QSM achieve performance close to that obtainable through more complex models.

References

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

[2] E. Anderson, J. Brooks, and S. Scott. Performance of the CRAY T3E multiprocessor. In *Proc. Supercomputing 97*, August 1997.

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

[4] E. Brewer and B. Kuszmaul. How to get good performance from the CM5 data network. In *Proc. of the 1994 International Parallel Processing Symposium*, April 1994.

[5] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.

[6] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, R. Bisseling. BSPlib: The BSP programming library. http://www.bsp-worldwide.org/standard/standard.htm, May 1997.

[7] 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.

[8] D. Culler, L. Liu, R. Martin, and C. Yoshikawa. LogP performance assessment of fast network interfaces. In *IEEE Micro*, 1996.

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

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

[11] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? In *Theory of Computing Systems* Special Issue on *SPAA '97*. To appear.

[12] B. Grayson. Armadillo: A high-performance processor simulator. Master's thesis, The University of Texas at Austin, May 1996.

[13] C. Holt, M. Heinrich, J. Singh, E. Rothberg, and J. Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

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

[15] 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.

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

[17] 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.

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

[19] V. Ramachandran, B. Grayson, and M. Dahlin. Emulation between QSM, BSP, and LogP: A framework for general-purpose parallel algorithm design. University of Texas at Austin Technical Report TR98-22, 1998. Summary to appear in *Proc. ACM-SIAM SODA*, 1999.

[20] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. 21st International Symposium on Computer Architecture* pages 325–336, April 1994.

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

[22] T. von Eicken, D. Culler, S. Goldstein, and K. E. Schauser, Active Messages: A mechanism for integrated communication and computation In *Proc. of the 19th International Symp. on Computer Architecture*, pages 256–266. May 1992.

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

Appendix: Detailed algorithm descriptions

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.
    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 gets 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 $i$th bucket
Major Step 4: Redistribution
    Each processor writes the sorted elements in its bucket into the appropriate locations
        in array $S$.
Barrier Synchronization
Un-register and deallocate temporary structures.
```

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, i.e. $I[i]$ points to the ith element in current linked list.

```
Initialization:
    Initialize $R$ to be all ones.
    Initialize $I[i] = i$, to give a one-to-one correspondence,
    Allocate and register temporary structures.
    since no element has been removed yet.
Major step 1: Each processor repeatedly removes some elements from its list,
until the list size is fairly small as follows.
    for $c \cdot logP$ 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 load 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.
    Perform a prefix-sum on the counts remaining at each processor.
    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.
```