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

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. In this paper, 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. If the model is too abstract, it may hide important aspects of parallel architectures and cause algorithm designers to make poor design decisions. Conversely, 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 details should be handled by its low-level architecture-specific implementation. This paper examines the trade-offs made by the Queuing Shared Memory (QSM) model [8]. 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.

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 one hand, the QSM can be considered a more realistic version of the PRAM [7, 13], since it is shared-memory, and additionally it (1) models bandwidth limitations and (2) supports computing in coarse-grained phases, 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 [18] and LogP [5] 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 [8] 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 a representative program and find that, as predicted by theory, programs written in coarse-grained phases 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

*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.

[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.

parallelizing. In addition, 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 adds negligible to modest delays 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 QSM on a simulator that lets us vary network performance to determine the impact of omitting network latency and overhead parameters from the model. 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 [8] 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 *bulk-synchronous* programming abstraction structured around coarse-grained *phases* in which (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 model of computing in phases 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, m_{op}, m_{rw}, κ, and g. 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 maximum number of remote reads or remote writes by any processor during a phase, represents its locality (or lack thereof); and m_{op}, the maximum number of local operations performed by any processor during a phase, represents its local computation time. The parameter κ, the maximum number of reads or writes to any remote memory location during a phase, represents the contention to any one remote memory object. κ 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 thereby encourages algorithms to exploit locality. Thus, for each phase, 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}, $g \cdot m_{rw}$, $g \cdot \kappa$).

Architectural/Algorithmic Parameter	**Implementation Contract**
Explicitly Modeled Factors	
p (number of processors)	QSM Parameter
m_{rw} (# of remote operations) m_{op} (# of local operations) κ (memory obj. contention)	Algorithm designer should minimize max($m_{op}, g \cdot m_{rw}, \kappa$)
g (gap)	QSM Parameter
Secondary Factors	
l (latency), L (barrier time)	Hide latency by pipelining
o (per-message overhead)	Use bulk synchronous style and batch messages
h_r (memory bank contention)	Randomize data layout
c (network congestion)	Use bulk synchronous style and limit network send rate

Table 1: System parameters.

QSM considers the second group of parameters in Table 1 — l, 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, large computation phases to amortize synchronization costs, 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 model of computing in coarse-grained phases 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.

- When designing a QSM algorithm, a designer does not explicitly account for L, the synchronization time. Instead, the designer follows the QSM cost model, which encourages minimizing the number of phases to amortize the cost of synchronization over a large amount of work [16].

- 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 concurrency within phases 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 h_r for accesses to remote memory banks except when there are many accesses to a specific remote object (captured by κ). Instead, the designer assumes that the compiler will limit the performance impact of bank conflicts by randomizing data layout [8].

- When designing a QSM algorithm, a designer does not explicitly account for c, the network congestion. Brewer and Kuszmal [2] found that network congestion could significantly limit the performance of parallel machines, and QSM expects compilers to address congestion by (1) using the periodic synchronizations associated with a phased programming style to reduce congestion and (2) limiting the rate at which nodes send data.

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 [13], which is a synchronous shared-memory model with unit-time communication to shared-memory. 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 and higher synchronization costs in a PRAM algorithm for a given problem than in a QSM algorithm. Also, the PRAM has no parameter to model bandwidth limitation, and hence the model does not encourage locality. 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) model 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*) [18] and the LogP [5] models each represent 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 its 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. Algorithms designed under these models tend to have rather complicated performance analyses because of the number of parameters in the model, the lack of a shared memory abstraction, and the need to keep track of the exact memory partition across the processors at each step.

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 input size 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 Workload

Our experiments studied three QSM algorithms, one with little communication, one with a medium amount of communication, and one with a large amount of irregular communication. As suggested by the QSM model, we optimized the algorithms to minimize computation and communication time, while keeping the number of phases small [16]. Due to space limitations, for this conference paper we present only the results for the algorithm with medium communication, *sample sort*. A detailed description of all three algorithms and experimental results on their performance can be found in the extended version of this paper [10].

We ran each experiment 10 times and report the average number of cycles. The standard deviation is less than 11% of the average for all of the runs.

3.2 Architecture

We used the Armadillo multiprocessor simulator [9] to simulate a distributed memory multiprocessor. For this set of

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-msg Overhead o	400 cycles (1 μs)	N/A
Latency l	1600 cycles (4 μs)	N/A
Synch. Barrier L	N/A	25500 cycles (p=16)

Table 2: Raw hardware performance and measured network performance (including hardware and software) for simulated system.

experiments, the processor and memory configuration parameters are set for an advanced processor in 1998 (with a 400MHz clock and 4 integer and 4 floating point functional units) and are not modified further.

The primary advantage of using a simulator is that it allows us to easily vary hardware parameters such as network bandwidth, latency, and overhead. Table 2 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. In Section 3.4 we will describe our experiments that vary these hardware parameters to examine the algorithm's sensitivity to them.

3.3 Results

Theory suggests that QSM's model of computing in phases 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 [16].[2] For our default system, l is 1600, g is 3, and p is 16. For sample sort π is 5, and W is linear with n. 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.

Figure 1 compares predicted and simulated communication performance in cycles. The *Communication* line shows measured communication performance. A QSM analysis predicts that communication will take time $4(p-1)g \log n + 3(p-1)g + gBr + gB$ with high probability (*whp*).[3] The algorithm is randomized, and the B and r terms represent how running time depends on the load balance achieved. B

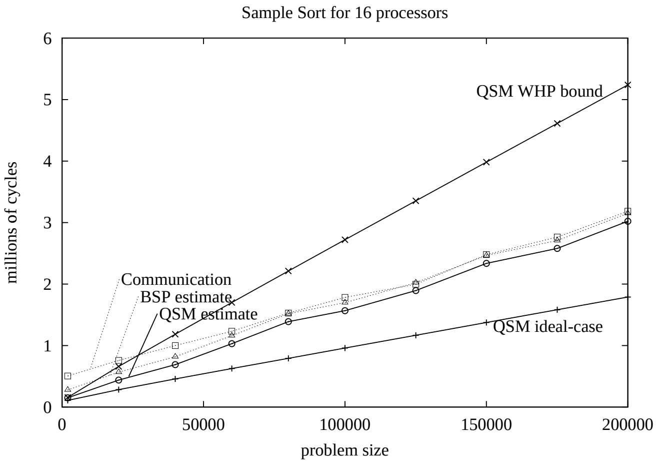

Figure 1: Measured and predicted communication performance for the sample sort algorithm.

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 ideal case $B = \frac{n}{p}$ and $r = \frac{p-1}{p}$, and all nodes have equal amounts of work to do. The *Ideal 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 [16]. The *WHP bound* line shows this as a conservative upper limit on typical performance. In addition, we experimentally measured the actual B and r skews experienced 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 measured B and r skews, and also including the experimentally-derived per-phase synchronization delays, for an additional $5L$ term over the QSM analysis. The ideal-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, synchronization, 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

[2]It also holds true if a QSM algorithm designed for p processors is mapped onto a p' processor machine where $(l/g) \cdot p' << p$ [8].

[3]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. For reference, the total running time including computation is about 20 million cycles for the 200,000-element problem size.

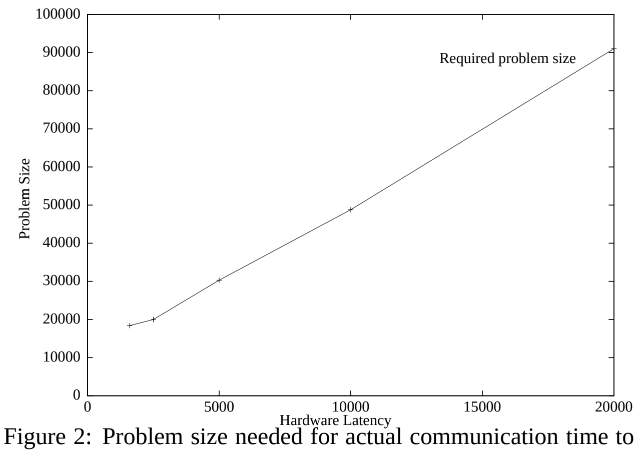

Figure 2: Problem size needed for actual communication time to fall within the range between the *WHP bound* and the *Ideal-case* lines as latency l is varied for sample sort.

per processor.) We believe that problems smaller than this limit are unlikely to be worth parallelizing, so model inaccuracy for such small problem sizes is not a major concern.

For the case where load balance is not known with precision, the *Ideal-case* and *WHP bound* lines in the graph bound predicted performance. 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.

3.4 Sensitivity to architectural parameters

The QSM model predicts that l and o are effectively hidden when $\frac{W}{p}$ 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 indicated above, we would predict a linear relationship between l, o, or p and the minimum n required for good prediction.

In Figure 2, we vary l, the hardware latency, over a range of values and show where the *WHP bound* line crosses the measured performance line for that latency. As hypothesized, increasing l results in a linear increase in the problem size n required for QSM to accurately predict performance. 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.

Although space limitations do not allow us to include the results here, we performed a similar experiment where we varied the machine's overhead, o, and we observed similar results. 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. As mentioned earlier, in our default configuration QSM accurately predicts communication time for sample sort when $n \geq 125,000$, which corresponds to just 8000 elements per processor. We believe this is a small problem size for a modern machine.

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 3 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. In the *Conflict* pattern, each access is to a random word in memory bank 0. 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.

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 with 166 MHz processors. 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 [4].

- NOW-BSPlib uses a cluster of sixteen 166 MHz UltraSPARCs connected by a 10 Mbit/s Ethernet. The benchmark uses the BSPlib runtime system, which uses TCP to simulate shared memory in this system.

Architecture	p	l	o	g	$\frac{n_{min}}{p}$
Default simulation parameters	16	1600	400	3	8000
Berkeley NOW [15]	32	830	481	4.3	(k * 4640)
300MHz Pentium-II TCP/IP, 100Mb Switched Ethernet	(32)	75000	150000	24	(k * 325000)
CRAY T3E [1]	(64)	126	(50)	1.6	(k * 1558)
Intel Paragon [6]	(64)	325	90	0.35	(k * 15429)
Meico CS-2 [6]	(32)	497	112	1.4	(k * 5325)

Table 3: 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.

- Cray T3E uses 32 nodes of a 68 node Cray T3E with DEC EV5 RISC microprocessors. We use the `shmem` shared memory library for data access to the shared array.

Figure 3 shows the performance of the benchmark on these architectures. The results conform to the assumptions of the QSM model. The careful 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.

5 Related work

Martin et al. [15] 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. [3] examined the performance of several message passing scientific codes. Holt et al. [11] 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 [17]. 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.

Juurlink and Wijshoff [12] provide an evaluation of several representative algorithms under variations of the BSP model. They found that the models generally predicted performance well and that prediction accuracy generally improved as problem size increased.

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

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

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

[4] 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.

[5] 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.

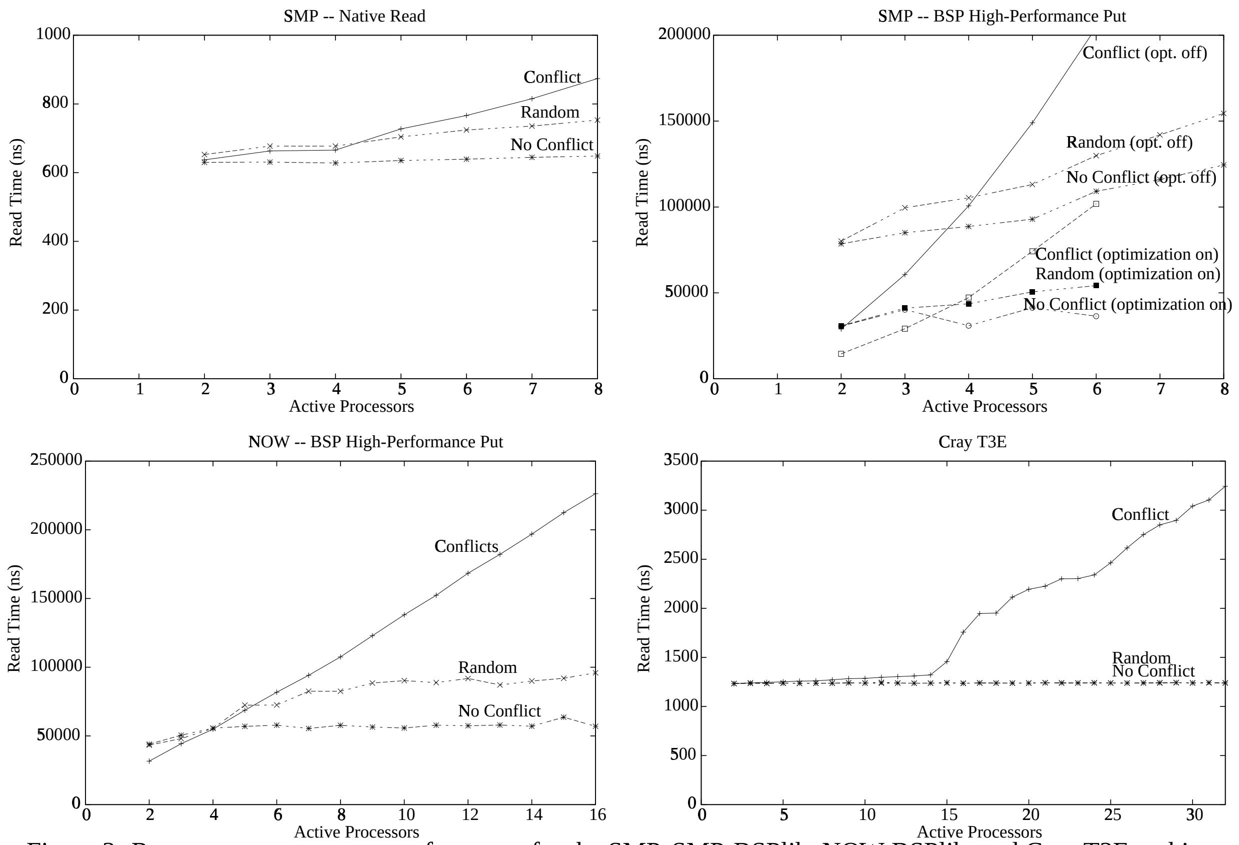

Figure 3: Remote memory access performance for the SMP, SMP-BSPlib, NOW-BSPlib, and Cray T3E architectures.

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

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

[8] 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.

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

[10] B. Grayson, M. Dahlin, and V. Ramachandran. Experimental evaluation of QSM, a simple shared-memory model. U. Texas at Austin Department of Computer Sciences Technical Report TR98-21, 1998.

[11] 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.

[12] B. Juurlink and H. Wijshoff. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 16(3) pages 271–318, August 1998.

[13] 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. 1990.

[14] 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.

[15] 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 Intl. Symp. on Computer Architecture*, pages 85–97, June 1997.

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

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

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

[19] 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 Intl. Symp. on Computer Architecture*, pages 256–266. May 1992.