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 $g(p-1)$.