

Parallel Hashing Algorithms on BSP and QSM Models

Hyunyoung Lee

Department of Computer Science, University of Denver
Denver, CO 80208, U.S.A.

hlee@cs.du.edu

Abstract

We study two parallel computing models – the Bulk Synchronous Parallel (BSP) and the Queued Shared Memory (QSM) – as alternatives to the PRAM model to provide more accurate performance predictions and analyses, and compares the two models in detail. As a case study, we consider a simple hashing problem, design the two versions – the message passing version and the shared memory version – of the algorithm, and compare their run time analytically. The message passing version of the algorithm is implemented and the experiments are performed to display the accuracy and the limitations of the predicted performance analysis.

1. Introduction

In this paper, we study two parallel machine models which have been proposed as alternatives to the PRAM: the BSP and the QSM. The PRAM – the most influential theoretical parallel machine model – allows algorithm designers to concentrate on the inherent parallelism of their algorithms without having to take machine details into account (e.g. [13]). However, the PRAM has received much criticism because the run time analysis in the PRAM model is often not a good indicator of an algorithm's run time on any existing parallel machine. This fact has led to the invention of a large number of alternative models. Their aim is to allow accurate performance predictions for existing parallel machines, while still offering a fairly abstract machine view to algorithm designers.

In general, parallel machine models have two purposes. Firstly, they should provide a simple and abstract basis for the design of parallel algorithms. In particular, they should free the algorithm designer from the need to take a large set of machine details into consideration, and enable the design of algorithms which work on a broad class of parallel machines.

A second goal is to allow a realistic analysis of the per-

formance of the designed algorithms. The term 'realistic' refers to the performance on existing or, at least, conceivable parallel machines. For example, the model should allow algorithm designers to estimate the running time of their algorithm when implemented to run on a parallel machine.

The two goals are somewhat conflicting. While the first goal appears to call for abstract models which ignore machine details, meeting the second goal appears to require some information about current parallel machine architecture in the model.

The large number of parallel machine models is, at least in part, due to a lack of agreement on the appropriate level of detail. Furthermore, the architecture of parallel machines keeps changing. For example, there are significant differences between older machines like the CM2 or MassPar (very large numbers of slow processors) and more modern machines like the IBM SP2 or the CM5 (moderate numbers of faster processors) or multiprocessors like the SGI Challenge or HP V-Class. These differences must be reflected in performance models.

This paper studies the following questions:

- How precisely and accurately do the alternative models present the run time analysis considering some degree of machine details?
- What are the limitations that the alternative models have?
- Which class of computing model – the message passing model or the shared memory model – represents the algorithms run time analysis more accurately?

We have selected two of a large number of alternative models in order to study and try to answer the above questions. They are the Bulk-Synchronous Parallel (BSP) model [19] and the Queuing Shared Memory (QSM) model [8]. As a case study, we consider a simple hashing problem. We design a parallel algorithm and analyze its run time in the BSP model and in the QSM model. In order to evaluate the accuracy of the run time predictions for the BSP model, we have

implemented the parallel algorithm using MPI and run the program on the parallel machine HP/Convex9000/V2200. A shared memory implementation for the QSM model will appear in the final version of the paper. While, overall, the models appear to make accurate predictions, our experiments also point out some limitations.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 studies the BSP model and the QSM model. Two versions of a parallel hashing algorithm are designed for the two models in Section 4. Section 5 presents the implementation of the message passing version of the algorithm, and discusses the experimental results. Section 6 concludes the paper with the future works.

2. Related Work

There has been a significant body of efforts to propose and justify various parallel computation models, regarding more realistic factors in parallel computation. It includes the BSP model, the LogP model, and the QSM model. There also have been numerous research results to compare those models quantitatively.

Maggs et al. [16] justify the objectives of having different models in parallel computation, comparing to sequential computation. Matias [17] addresses the issue of choosing a suitable model for parallel algorithm design and provides a high level overview of numerous models including the PRAM, the BSP, and the QSM.

The BSP model was first introduced by Valiant [19], that emphasized the separation of communication from computation by incorporating the bulk-synchrony with a distributed memory model over message passing. The idea was arrived by seeing the Von Neumann model of sequential computation as an efficient bridge between software and hardware, and by trying to adopt an analogous unifying bridging model for parallel computation, aiming to provide a general purpose parallel computing model.

The BSP* model was proposed by Bäumker et al. [3] as an extension of the BSP model, considering the significance of the set-up time for a message, by adding one more parameter B , which indicates the minimum message size in order to fully utilize the bandwidth of the router. Götz [10] provides a survey of algorithms in several models such as the BSP, the BSP*, and so on.

The E-BSP model by Juurlink and Wijshoff [14] is an extension of the BSP model, which takes unbalanced communication patterns and locality of memory access into account. In [15], Juurlink and Wijshoff compare quantitatively the BSP model, the E-BSP model, and the Message-Passing Block PRAM model that is a restrictive version of [1].

In [12], Hill and Skillicorn address some misconceptions on poor performance and expensiveness of the syn-

chronization operations in parallel computation in general, and show how the BSP model can provide improved performance based on their implementation experience of the BSP model.

The LogP model was proposed by Culler et al. in [6], having the BSP model as a starting point to address the realistic factors of a parallel model, yet trying to provide as simple a view of the machine as possible to algorithm designers. Alexandrov et al. [2] proposed the LogGP model as an extension of the LogP model, by incorporating long messages into the LogP model, with an additional parameter, G , which represents the bandwidth for the long messages. Various parallel sorting algorithms are extensively analyzed on the LogP model in [7].

Bilardi et al. [4] show a quantitative comparison of the BSP and LogP models through cross simulations between the two models.

The QSM model was first introduced by Gibbons et al. [8], serving as a bridging model with the shared memory paradigm. Adopting the QSM model as a general-purpose parallel computing model has been proposed by Grayson et al. [11, 18].

The uniqueness of our work compared to those previous works is that we designed the two versions of a parallel hashing algorithm, for the two models (BSP and QSM) and tried to predict their performance analytically. Such analysis allows us to justify the usage of the parameters of each model and the structure of each algorithm designed for each model. The similarities and differences between the estimates of the two models were also observed, based on the analysis.

Employing a simple application of parallel hashing in Section 4 was inspired by [9] in which a sophisticated theoretical work on a fast parallel hashing is described.

3. Parallel Machine Models

This section describes the two parallel machine models considered in this paper. The BSP model appears to be the most influential distributed memory model. The QSM model is an interesting shared memory model.

3.1. The BSP Model

In *bulk-synchronous* computing models, processors compute asynchronously between synchronization barriers. The BSP model was introduced by Valiant [19] in 1990. It views a parallel machine as a collection of p processor/memory modules which are linked by a fast interconnection network. The processor/memory modules can be viewed as independent workstations which are connected to a fast network. Architectures of this kind can be found in modern distributed memory machines such as the IBM SP2

whose processor/memory modules are essentially RS/6000 workstations.

The communication network or *router* is described by only two parameters: The *message latency* L and the *bandwidth factor* g . The latency is the time needed by a short message to travel across the network to its destination processor. The parameter g corresponds to a capacity constraint on the network. More precisely, it is defined as the ratio of local operations performed by all processors in one time unit to the total number of messages delivered by the router in the same time unit.

Beyond the parameters p , g and L , the BSP model does not take any other properties of the network into account. In particular, the interconnection topology of the network is ignored. This abstract view and omission of machine details makes BSP algorithms and their analysis applicable to a wide range of parallel machines. At the same time, the performance predictions made by the BSP model appear fairly accurate for existing parallel machines [12]. It has been argued extensively (e.g. [6]) that parameters like L and g are sufficient to model modern parallel machines such as the CM5 or the IBM SP2.

A BSP program is a sequence of *supersteps*. During each superstep, the processors (processor/memory modules) perform arbitrary local computations. At the end of each superstep, the processors synchronize and communicate by sending messages over the network (router). The router realizes supersteps in which each processor sends and receives at most h messages (*h*-relation). This pattern of independent computations followed by synchronization and communication steps is called *bulk-synchronous*.

The cost (time) of a superstep is defined to be

$$\max\{c, g \cdot h_s, g \cdot h_r, L\} , \quad (1)$$

where c is the maximum time spent by any processor performing local computations, and h_r , h_s are the maximum numbers of messages sent or received by any processor, respectively.

The basic BSP model has been extended in many directions by introducing additional parameters (e.g. the E-BSP model of [14] or the (d, x) -BSP model of [5]). Furthermore, there is a close similarity between the BSP model and the LogP model of [6]. The only two differences are an additional parameter (overhead o) in the LogP model (which is often ignored) and the omission of required synchronization steps in the LogP model. The LogP model allows fully asynchronous behavior of the processors rather than bulk-synchrony. The close relationship between the two models is formally analyzed in [4].

3.2. The QSM Model

Most of the models which were designed as more realistic alternatives to the PRAM are based on some form of message passing. The purpose of the QSM model is to present an alternative to these models: a shared memory model whose performance predictions are realistic.

Like the BSP model, the QSM model views a parallel machine as a collection of isolated processor/memory modules. However, the concept of a router is replaced by shared memory. That is, processors communicate by writing to and reading from shared memory.

QSM algorithms, like BSP algorithms, are bulk synchronous. The supersteps are called *phases* in the QSM model. During each phase, the processors can perform arbitrary local computations and arbitrary number of shared memory access. However, two important restrictions distinguish the QSM shared memory from that of a PRAM: The results of shared memory reads are available only after the end of the current phase (i.e. after the end of the phase in which the read operation is performed). Furthermore, there cannot be both read and write operations to the same shared memory location within a single phase. Multiple read or multiple write operations within one phase are allowed. These read or write requests are queued and executed sequentially.

The cost (time) of a phase is defined as

$$\max\{c, g \cdot h_r, g \cdot h_w, \kappa\} , \quad (2)$$

where c and g are defined as in the previous section (BSP model), κ – the *maximum contention* – is the maximum number of accesses to any shared memory location, and h_r , h_w are the maximum numbers of shared memory read or write operations executed by any processor, respectively.

4. The Algorithms

This section describes a simple application which we use to compare quantitatively the performance predictions of the two models (BSP and QSM): parallel hashing. In particular, we consider the parallel batch insertion of a large number of keys into a hash table which is distributed over p processor/memory modules.

Hashing is a fundamental search and data storage technique. It is used throughout computer science. It is one of the main implementation methods for abstract data types like dictionaries. It is also used extensively in the implementation of data base systems. Distributed database engines are concrete examples of applications of parallel hashing. Hashing is also a popular technique in the implementation of shared memory [8].

In this paper, we consider only standard forms of hashing. More sophisticated theoretical work on parallel hashing is described in [9]. The goal is to achieve constant access time on average. Quadratic probing is used for collision resolution. However, the analysis described in this section does not depend on any particular collision resolution method.

Given a large number n of keys, the task is to store the keys such that each key can be retrieved, on average, in constant time. In the sequential case, the algorithm uses a large array of size approximately $2n$. A random hash function H is used to map keys to array indices. If the corresponding array position is empty, the key is stored there. If the array position is already occupied by another key, a collision has occurred. A different array index must be found. Quadratic probing tries array index $H(x) + i^2$ in the i -th attempt, where x is the key to be stored.

The parallel algorithm is based on the idea of mapping each key to some processor/memory module, and letting each processor execute a hashing algorithm locally for the keys it is assigned. Figure 1 (a) shows the message passing version of the algorithm. Figure 1 (b) shows the shared memory version of the algorithm. Each processor is given n hash keys. These keys are processed in n/h rounds, where h is a parameter of the algorithm. Each round processes h keys. For simplicity, we assume that h divides n .

Each key is obtained from the function `nextKey()` which hides the input representation. The function `keyToProcessor` maps a given key into the range $\{1, \dots, p\}$. This function should behave like a uniformly distributed random function in the sense that the input key sequence should, with high probability, have approximately the same number of images j for any $j \in \{1, \dots, p\}$. The key is put into the local buffer which stores all keys destined for the same processor. This procedure is repeated for all h keys processed in the current round.

In the message passing version, the next step is to send the buffers to the processors which have to store the keys in them. That is, the j th buffer is sent to processor j for $j \in \{1, \dots, p\}$. Similarly, each processor receives all buffers that was sent by other processors. The last step is to execute a standard sequential hash algorithm on each processor, to extract each key from the buffers, and to store it in the local hash table.

In the shared memory version, the j th buffer is written to the shared memory location for processor j (X_j) for $j \in \{1, \dots, p\}$. And each processor j reads from its shared memory location X_j to read all the keys that were written by other processors, into the local buffer. In the algorithm of Figure 1 (b), queue operations (dequeue and enqueue) with shared queues are used instead of read and write operations on individual shared memory location, in order for a simpler presentation. The last step is to extract each key

from the local buffer and store it in the local hash table.

4.1. The BSP Version

The supersteps of the BSP model are delimited by the send and receive operations of each round. Thus, each superstep consists of (a) the local hashing step of the previous phase, (b) the key to processor assignment step of the current phase, and (c) the communication step of the current phase.

The next step is to estimate the running time of the algorithm in the BSP model. Let t_h be the local computation time needed to process a single key. This time includes the times taken up in steps (a) and (b).

Note that, even if each processor sends exactly h keys, it will, in general, receive not exactly h keys, because H may not partition the input sequence equally among the p buffers. However, if H is chosen appropriately, it will behave essentially like a random function. For random functions, it is assumed that large load imbalances are extremely unlikely as long as h is sufficiently large compared to p . In other words, with high probability no processor will receive significantly more than h keys, if h is sufficiently large. For simplicity, we will ignore the remaining minor imbalances.

The exact number of messages sent by each processor in each round is not h , but only $h(p-1)/p$ since one of the p buffers on each processor is destined for the processor itself and does not have to be sent. This leads to the following time estimate, based on (1):

$$t_{\text{superstep}} \approx \max\{t_h h, gh(p-1)/p, L\}$$

There are n/h rounds. Thus, the total running time of the algorithm is

$$\begin{aligned} t_{\text{BSP}}(n, h) &\approx \frac{N}{hp} \max\{t_h h, gh(p-1)/p, L\} \\ &= \begin{cases} NL/(hp) & \text{if } L > h \max\{t_h, g\} \\ (N/p) \max\{t_h, g(p-1)/p\} & \text{otherwise} \end{cases} \end{aligned}$$

where $N = np$ is the total number of hash keys processed on all processors.

4.2. The QSM Version

It is straightforward to convert the message passing algorithm of Figure 1 (a) into a shared memory algorithm. We only need to replace the message send and receive operations by write and read operations to shared memory, respectively. In the simplest case, shared memory will contain space for $p(p-1)$ buffers, so that each of the p processors can write its $p-1$ buffers, before it reads the $p-1$ buffers destined for it. Note that the required amount of

(a) Message passing version

Algorithm for Processor P_i
FOR *round* FROM 1 TO n/h DO
 FOR j FROM 1 TO h DO
 key = nextKey();
 buf[keyToProcessor(key)].insert(key);

 /* message send */
 FOR j FROM 1 TO p DO
 IF $j \neq i$ THEN
 send buf[j] to processor j ;

 /* message receive */
 FOR j FROM 1 TO p DO
 IF $j \neq i$ THEN
 receive message from processor j into buf[j];

 insert keys in buf[1], . . . , buf[p] into local hash table

(b) Shared memory version

Algorithm for Processor P_i
/* X_k : shared memory (queue) for processor k */
FOR *round* FROM 1 TO n/h DO
 FOR j FROM 1 TO h DO
 key = nextKey();
 buf[keyToProcessor(key)].insert(key);

 /* shared memory write */
 FOR j FROM 1 TO p DO
 IF $j \neq i$ THEN
 enqueue buf[j] to X_j ;

 /* shared memory read */
 bufForLocalKeys.insert(dequeue(X_i));
 bufForLocalKeys.insert(buf[i]);

 insert keys in bufForLocalKeys into local hash table

Figure 1. Message passing version and shared memory version of parallel hashing algorithm.

shared memory could be reduced to only p buffers by interleaving read and write operations. However, this would lead to a more complicated description of the algorithm and to an increase in the number of phases in the QSM model by a factor of p . In the algorithm of Figure 1 (b), queues are used instead of shared memory buffers to present the algorithm more precisely. The enqueue operation corresponds to many write operations for each element in each buffer, and the dequeue operation corresponds to many read operations for each element in the buffers which are destined for the processor.

The restrictions on shared memory accesses imposed by the QSM model force us to split each round of the algorithm into two phases. The first separation between phases takes place immediately after the shared memory write operations and right before the shared memory read operations. This separation is necessary because read and write accesses to the same shared memory location within one phase are not allowed in the QSM model. The next phase separation takes place between the last shared memory read operation and the beginning of the local hashing step. This separation is necessary because the results of shared memory read operations are not available in the phase which executed the shared memory read. We call the phase spanning the local hashing step of the previous round, the assignment of keys to buffers and the shared memory write operations of the current round, phase A. The phase comprising the shared memory read operations is called phase B.

Making the same assumptions about load imbalances as for the BSP algorithm and using (2), we arrive at the following estimates for the cost of each phase:

$$t_{\text{phaseA}} \approx h \max\{t_h, g(p-1)/p\}$$

$$t_{\text{phaseB}} \approx hg(p-1)/p$$

This leads to a total running time for all n/h rounds of

$$\begin{aligned} t_{\text{QSM}} &\approx \frac{N}{hp} (t_{\text{phaseA}} + t_{\text{phaseB}}) \\ &= \frac{N}{p} (g(p-1)/p + \max\{t_h, g(p-1)/p\}) . \end{aligned}$$

It is interesting to observe the similarities and differences between the estimates for t_{BSP} and t_{QSM} . The latter does not depend on h at all, whereas the former does. The reason is the absence of a latency term in the QSM model. Indeed, the comparisons between the QSM model and the BSP model in [8] are done for the special case $L = 0$. If L is set to zero in t_{BSP} , the estimates for t_{BSP} and t_{QSM} become almost identical. The only remaining difference is the additional term g . This term is an artifact of the shared memory restrictions in the QSM model, which make it necessary to partition each round of the algorithm into two phases.

5. Implementation and Experiments of the BSP Model

The goal of this section is to validate the run time predictions of the previous section by time measurements on an existing parallel machine. Based on some of these measurements, we have tried to estimate the machine parameters L and g . For this purpose, we have implemented the message passing algorithm of Figure 1 (a). The program is written in C++ and uses the MPI message passing library for the send and receive operations. In particular, the send operations call `MPI_Send`, and the receive operations call `MPI_Recv`.

The implementation deviates from the BSP specification in one important aspect. Rather than sending each key in a separate small message – as suggested by the BSP model – all keys in a given buffer are sent together in one large message. Thus, sending each buffer requires only two messages: a short message which specifies the number of keys in the buffer followed by a long message containing the keys themselves.

The experimental platform was a HP/Convex9000/V2200 with 16 200 MHz PA-8200 CPUs running in 4GB of physical memory. We have performed two kinds of experiments. The purpose of the first set of experiments was to measure the influence of h (number of keys per round) on the total running time. The data from this experiment are also used to estimate the machine parameters L and g . The goal of the second set of experiments was to investigate how the run time scales as the number of processors increases.

Experiment 1: Varying the number of keys per round.

In this set of experiments, the number of processors p and the total number of hash keys were fixed ($p = 16$, $np = 1120000$). The number of keys per round was varied from 1 to 70000, and the running times were measured. The results are displayed in Figure 2 (a). The number of keys per round is log-scaled base 2, and shown up to only 14000 to display more clearly the change of the total running times at the beginning.

The predicted and measured times show roughly similar behavior: they decrease in h . The predicted time t_{BSP} decreases initially, as the latency L is amortized over a larger number h of keys (messages). After the point at which the amortized latency becomes smaller than the time taken up by local computations (c) or sending messages, t_{BSP} is constant. Similarly, for $h > 80$, the measured time decreases only insignificantly.

We have used these measurements to obtain a rough estimate of the machine parameters. If $h = 80$ is taken as the point at which the running time becomes constant, we have,

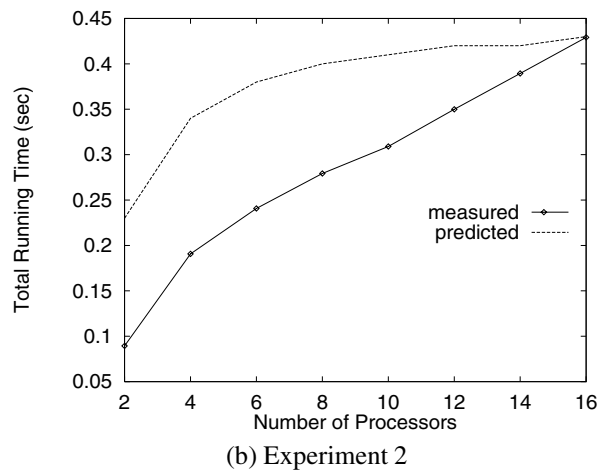
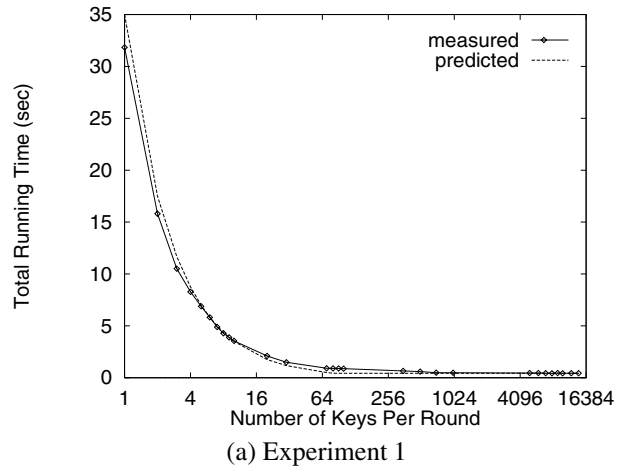


Figure 2. Experiments 1 & 2: Measured and predicted running time.

based on the estimate for t_{BSP} ,

$$t_{\text{measured}} \approx NL/(hp) \quad \text{for } h \leq 80$$

and

$$\begin{aligned} t_{\text{measured}} &\approx (N/p) \max\{t_h, g(p-1)/p\} \\ &= (N/p)g(p-1)/p \quad \text{for } h > 80, \end{aligned}$$

since t_h is sufficiently small. The two equations can be solved for L and g . The results are $L \approx 500\mu\text{s}$ and $g \approx 6.5\mu\text{s}/\text{key}$.

However, the result for L does not appear to account for the true machine latency. It is derived from the formula for t_{BSP} which is based on the BSP estimate (1). Equation (1), charges only one latency term L for sending h messages, assuming that the h messages can be perfectly pipelined. However, this is not the case in our implementation, which uses `MPI_Send` – a blocking send operation. Thus, our implementation takes one latency term L for each of the $2(p-1)$ messages it sends per round. Hence, we should use the following equation to estimate L :

$$t_{\text{measured}} \approx 2(p-1)LN/(hp) \quad \text{for } h \leq 80.$$

We obtain the following estimates for the machine parameters:

$$L \approx 16.7\mu\text{s} \quad g \approx 6.5\mu\text{s}$$

These estimates are used to plot the predicted curves in figures 2 (a) and 2 (b).

Experiment 2: Varying the number of processors. For this set of experiments, n (the number of hash keys per processor) was set to 70000, and p (the number of processors) was varied from 2 to 16. For each value of p , the total running time was measured. We took care to ensure that there was no other load on the machine in order to avoid measurement errors.

The results are plotted in Figure 2 (b). Because of the way L and g were estimated, the measured and predicted curves coincide at $p = 16$. However, they differ for smaller values of p . The estimate t_{BSP} predicts an increase in the running time of less than a factor of 2, as p is increased from 2 to 16. However, the measured times increase by more than a factor of 4.

This discrepancy can have several reasons. Firstly, the BSP model assumes that g (the per processor bandwidth parameter) is constant, i.e. independent of p . It is not clear if the HP Convex 9000 has this property. The total amount of data which need to pass through the router (network) is proportional to the number of processors. It appears possible that an increase by a factor of 8 in the number of processors – and, thus, in the total data volume – could result in an increased communication time. While g is near constant

in certain high-performance machines like the CM5 or the IBM SP2, it is clearly not constant for simple bus architectures (e.g. workstations connected by an Ethernet).

A second reason could be the fact that our implementation sends all keys in a given buffer together in a single long message, rather than sending each key in a separate short message. Bundling messages in this way leads to a much more efficient program. However, the basic BSP model does not foresee the possibility of increasing efficiency by sending long messages.

Several authors have suggested ways to incorporate long messages into parallel machine models [2]. The basic observation is that, for most parallel machines, the bandwidth factor g is not constant, but that it depends on the message length. In most machines, the de facto bandwidth for long messages is much higher than for short messages. A simple example of a model which accounts for this phenomenon is the LogGP model [2]. It adds a second bandwidth parameter G to the basic LogP model, such that G specifies the bandwidth for long messages and g specifies the bandwidth for short messages. Clearly, similar parameters could be introduced to the BSP model. The details of such extensions would require further work.

6. Conclusions

Two models were selected for comparison in this research: the BSP model and the QSM model. As a case study, we have selected a parallel hashing problem and formulated and analyzed parallel algorithms in the two models. In order to validate the run time predictions made by the models, we have implemented the BSP version of the algorithm and measured its run time on the parallel machine, HP/Convex9000/V2200. While the predictions were qualitatively correct, the experiments also show a limitation of the model which makes its prediction somewhat imprecise.

The work of this paper can be extended in several ways. A more detailed validation of the QSM model and, in particular, a special shared memory implementation is an ongoing work. Furthermore, it would be interesting to study ways to extend the models in order to account for different message sizes.

Acknowledgments: We thank Nancy Amato for helpful direction of this research and Jennifer Welch for valuable comments.

References

- [1] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in pram computations. *Proceedings of Symposium on Parallel Algorithms and Architectures*, ACM, pages 11–21, 1989.

- [2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures, ACM*, pages 95–105, 1995.
- [3] A. Bäumer, W. Dittrich, and F. M. auf der Heide. Truly efficient parallel algorithms: c -optimal multisearch for an extension of the bsp model. *Proceedings of European Symposium on Algorithms*, pages 17–30, 1995.
- [4] G. Bilardi, K. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. Bsp vs. logp. *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures, ACM*, pages 25–32, 1996.
- [5] G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagna. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures, ACM*, pages 84–94, July 1995.
- [6] D. Culler, R. Karp, D. Patterson, A. Sahay, R. S. K. Schauser, E. Santos, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *Proceedings of the fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, 1993.
- [7] A. C. Dusseau. Modeling parallel sorts with LogP on the CM-5. *Technical Report: UCB/CSD-94-829, Dept of EECS, University of California, Berkeley*, 1994.
- [8] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures, ACM*, pages 72–83, 1997.
- [9] J. Gil and Y. Matias. Simple fast parallel hashing by oblivious execution. *SIAM Journal on Computing*, pages 1348–1375, 1998.
- [10] S. Götz. Algorithms in CGM, BSP and BSP* model. *Project Report 95.574 Parallel Algorithms and their VLSI Implementation, Carleton University*, Fall 1996.
- [11] B. Grayson, M. Dahlin, and V. Ramachandran. Experimental evaluation of qsm, a simple shared-memory model. *Technical Report: CS-TR-98-21, University of Texas, Austin*, November 1998.
- [12] J. Hill and D. Skillicorn. Lessons learned from implementing bsp. *High-Performance Computing and Networking, Lecture Notes in Computer Science 1225, Springer-Verlag*, 1997.
- [13] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.
- [14] B. H. H. Juurlink and H. A. G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. *Proceedings of the second International Euro-Par Conference*, 11, 1996.
- [15] B. H. H. Juurlink and H. A. G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 16(3):271–318, August 1998.
- [16] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: A survey and synthesis. *Proceedings of the 28th Annual Hawaii Int. Conf. on System Sciences, IEEE Computer Society Press, II: Software Technology*:61–70, 1994.
- [17] Y. Matias. Parallel algorithms column: On the search for suitable models. *SIGACT News*, September 1997.
- [18] V. Ramachandran, B. Grayson, and M. Dahlin. Emulations between qsm, bsp and logp: A framework for general-purpose parallel algorithm design. *Technical Report: CS-TR-98-22, University of Texas, Austin*, December 1998.
- [19] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.