BSP, LogP, and Oblivious Programs

Jörn Eisenbiegler Welf Löwe Wolf Zimmermann

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe,
76128 Karlsruhe, Germany,
{eisen|loewe|zimmer}@ipd.info.uni-karlsruhe.de

Abstract. We compare the BSP and the LogP model from a practical point of view. Using compilation instead of interpretation improves the (best known) simulations of BSP programs on LogP machines by a factor of $O(\log P)$ for oblivious programs. We show that the runtime decreases for classes of oblivious BSP programs if they are compiled into LogP programs instead of executed directly using a BSP runtime library. Measurements support the statements above.

1 Introduction

Parallel programming suffers from the lack of a uniform and commonly accepted machine model providing abstract programming of parallel machines and describing their costs adequately. Two candidates, the BSP model (Valiant [9]) and the LogP model (Culler et al. [4]), have been considered in an increasing number of papers. The comparison of the two models in [2] determines the delays for a simulation of LogP programs on the BSP machine and vice versa. For our observations we make two additional assumptions: we only consider oblivious programs and message passing architectures. A program is *oblivious* if source and destination processors of communications are statically determined[1]. The target machines are processor-memory-nodes connected by a communication network. We explicitly exclude shared memory machines. Virtual shared memory architectures are covered since they implicitly require communication via the interconnection network for remote memory operations. We only mention send and receive communications and deliberately ignore remote store and load operations.

The first part of our paper shows that oblivious BSP programs can be compiled to the LogP machine. We further show, that compilation reduces the delay of the simulation of oblivious BSP programs on the LogP machine by a factor of $O(\log(P))$ compared to the result in [2], i.e., there is asymptotically no delay for the compiled LogP program compared to the BSP program. Even better, it turns out that the compiled LogP program could outperform a direct execution of the BSP program on the same architecture. To sharpen this observation, we consider three classes of oblivious programs: first we study Multiple-Program-Multiple-Data (MPMD) solutions. Those programs are in general hard to partition into

[1] Many algorithms, especially those in scientific computing are oblivious, e.g. Matrix Multiplication, Discrete Simulation, Fast Fourier Transform, etc.

supersteps, i.e. in global phases of receive-, compute-, and send-operations. As an example, we discuss the optimal broadcast problem. The second and third classes of problems allow Single-Program-Multiple-Data (SPMD) solutions and there is a natural partition into supersteps. They differ in the data dependencies between the phases: in the second class there are sparse dependencies between the phases in third class these dependencies are dense. We call a data dependency sparse iff the BSP communication of each superstep last longer than the corresponding communication in the compiled LogP program. As representatives of the second class of problems, we discuss a numeric wave simulation. The fast Fourier transform is a representative of the third class. Measurement of the parameters and run time results for an optimal broadcast, the simulation and the FFT support our theoretical results.

2 The Machine Models

This section describes the two machine models a little more in detail. In order to distinguish between the parameters of both models, we use capital letters for the parameters of the LogP model and small letters for the parameters of the BSP model.

2.1 The LogP Model

The LogP model assumes a finite number P of processors with local memory, which are connected by a data network. It abstracts from the network topology, presuming that the position of the processor in the network has no effect on communication costs. Each processor has its own clock, synchronization and communication is done via message passing. All send and receive operations are initiated by the processor which sends or receives, respectively. From the programmers point of view, the network has no direct connection to the local memory. All communication is done via the processor.

In the LogP model, communication costs are determined by the parameters L, O, and G. Sending a message costs time O (overhead) on the processor. The time the network connection of this processor is busy with sending the message into the network is bound by G (gap). A processor can not send or receive two messages within time G, but if a processor returns from a send or receive routine, the difference between gap and overhead can be used for computation. The time between the end of sending a message and the start of receiving this message is defined as latency L. There are most $\lceil L/G \rceil$ messages in transit from any or to any processor at any time, otherwise, the communication stalls. We only consider programs satisfying this capacity constraint. If the sending processor is still busy with sending the last bytes of a message while the receiving processor is already busy with receiving, the send and the receive overhead for this message overlap. In this case the latency is negative. This happens on many systems especially for long messages or if the communication protocol is too complicated. L, O, and G have been determined for quite a number of machines; all works confirmed

runtime predictions based on the parameters by measurements. In contrast to [1] and [5], we assume the LogP parameters to be constants (as proposed in the early LogP works [4]). This assumption is admissible if the message size does not vary in a single program.

2.2 The BSP Model

The BSP (bulk synchronous parallel) machine was defined by Valiant [9]. We refer to its modification by McColl in [8,6]. Like the LogP model, the BSP model assumes a finite number of processors P with local memory, local clock, and a network connection to an arbitrary network. It also abstracts from the network topology. In contrast to the LogP model, the BSP machine can explicitly (barrier) synchronize all processors. The synchronization barriers subdivides the calculation into *supersteps*. All send operations in a superstep i are guaranteed to be completed before superstep $i+1$. In the BSP model as invented by Valiant, processor communicate via remote memory access. For oblivious programs we may focus on a message based communication: Consider the remote memory accesses at one superstep. If processor π_i reads from processor π_j, then π_i should send a request to π_j and π_j sends its answer for the general case. However, for oblivious algorithms, it is already known that π_i reads from the memory of π_j. Thus, the request can be saved in the case of oblivious algorithms: it is sufficient to send the result of the read request from π_j to π_i. A write of processor π_i to the memory of processor π_j, is equivalent to sending a message from processor π_i to π_j containing the memory address and the value to be written.

The cost model uses two parameters: the time for the barrier synchronization l, and the reciprocal of the network bandwidth g. With theses parameters, the time for one superstep is bounded by $l + 2h \cdot g + w$, where h is the maximal number of messages sent or received by one processors and w is the maximal computation time needed by one processor in this superstep. A BSP machine is able to route a $\lceil l/g \rceil$-relation in a superstep which is a capacity constraint analogous to the LogP model. The total computation time is the sum of the time for all supersteps. Like for the LogP model, we assume the parameters to be constant.

3 BSP vs. LogP for Oblivious Algorithms

In this section, we discuss the compilation of oblivious BSP programs to the LogP machine. First, we discuss how the communication between subsequent supersteps can be mapped onto the LogP machine without exceeding the LogP capacity constraints. Second, we define the actual compilation and prove execution time bounds for the compiled LogP programs. Third, we compare the direct execution of a BSP program on a target machine with the execution of the (compiled) LogP program. Therefore, we conclude this section by determining lower bounds for the BSP and LogP parameters, respectively, for the same target machine.

3.1 Communication of a Superstep on the LogP Machine

For simplicity, we assume that a h-relation is implemented, i.e. each processor sends and receives exactly h messages. It is well known that a "pipelined" communication can be computed using edge coloring on a bipartite graph (U, V, E) where U and V are the set of processors and $(u, v) \in E$, iff u communicates with v. Each color, represented by an integer $j \in \{0, \ldots, k-1\}$ where k is the number of required colors, defines set of non-conflicting communications that can be started simultaneously. A $send(v)$ on processor u is scheduled at time $j \cdot \max(O, G)$ and $recv(u)$ on processor v at time $L + O + j \cdot \max(O, G)$.

Since we consider oblivious BSP algorithms, the edge coloring of the communication graph for each superstep (and therefore the communication phase itself) can be computed prior to execution of the BSP algorithm. Thus, the time for edge coloring ($\mathcal{O}(|E| \log(|V| + |U|)$ due to [3]) can be ignored when considering the execution time of the BSP algorithm.

It is easy to see that the schedule obtained from the above algorithm does not violate the capacity LogP constraints if $L \geq (h-1) \cdot \max(O, G)$. If follows

Lemma 1. *If $L \geq (h-1)\max(O, G)$, then every fixed h-relation can be implemented on the LogP machine such that its execution time is $L + 2O + (h-1)\max(O, G)$.*

If $L < (h-1)\max(O, G)$ the scheduling algorithm must be modified to avoid stalling. First we discuss the simplified model where each channel is allowed to contain an arbitrary number of messages. In this case, the first receive operation is performed after the last send operation. The same approach as above then yields execution time $2O + 2(h-1)\max(O, G)$ since the first receive operation can be performed at time $O + (h-1)\max(O, G)$ instead of $O + L < (h-1)\max(O, G)$. If the number of messages is bounded, the message is received greedily, i.e. as soon as possible after a send operation on the processor is finished at the time when the message arrives. This does not increase the overall execution time of a communication phase since no new gaps are introduced. Thus, the following lemma holds:

Lemma 2. *If $L < (h-1)\max(O, G)$, then every fixed h-relation can be implemented on the LogP machine such that its execution time is $2O + 2(h-1)\max(O, G)$.*

3.2 Execution Time Bounds for the Compiled LogP Programs

The actual compilation of an oblivious BSP algorithms to the LogP machine is now straightforward: Each BSP processor corresponds one to one to a LogP processor. Beginning with the first superstep we map the tasks of BSP processor to a corresponding LogP processor in the same order. Communication is mapped as described in the previous subsection. Since a processor can proceed its execution when it received all its messages of the preceding superstep, there is no need for a barrier synchronization. Together with Lemmas 1 and 2, this observation leads to the

Theorem 1 (Simulation of BSP on LogP). *Every superstep of an oblivious BSP algorithm with work w and h remote memory accesses can be implemented on the LogP Machine in time $w + 2O + (h-1)\max(O,G) + \max(L,(h-1)\max(O,G))$.*

If we choose $g = \max(O,G)$ and $l = \max(L,(h-1)\max(O,G)) + 2O - \max(G-O,0)$ then, the execution time of a BSP algorithm in the BSP model and the compiled BSP algorithms in the LogP model is the same. Especially, the bound for the simulation in [2] is improved by a factor of $\log P$ for oblivious BSP programs.

3.3 Interpretation vs. Compilation

For the comparision of a direct execution of a BSP program with the compiled LogP program, the parameters for the BSP machine and LogP machine, respectively, cannot be chosen arbitrarily. They are determined by the target architecture, for comparable runtime predictions we must choose the smallest admissible values for the respective model.

A superstep implementing a h-relation costs in the BSP-model $w + l + h \cdot g$. According to Theorem 1, it can be executed in time $w+2O+(h-1)\max(O,G)+\max(L,(h-1)\max(O,G))$ on the LogP machine. The speedup is the ratio of these two numbers. Easy calculations prove the following

Corollary 1 (Interpreting vs. Compiling). *Let M be a parallel computer with BSP parameters l, g, and P and LogP parameters L, O, G, and P. Let $\mathcal{A}$ an oblivious BSP algorithm where each superstep executes at most h remote memory accesses. If $l \geq O + \max(L,(h-1)\max(O,G))$ and $g \geq \max(O,G)$ then the execution time of the compiled BSP algorithm on the LogP model is not slower than the execution time of the BSP algorithm on the BSP model. If one of these inequalities is strict, then the execution time of the compiled BSP algorithm is faster.*

3.4 Lower Bounds for the Parameters

Let g^* and l^* (resp. $\max^*(O,G)$ and L^*) be the smallest values of the BSP parameters (resp. LogP parameters) admissible on a certain architecture. Our simulation implies that $g^* = \mathcal{O}(\max^*(O,G))$ and $l^* = \mathcal{O}(L^*)$. Together with the simulation result of [2] that proves a constant delay in simulation of LogP algorithms on the BSP machine, we obtain

Theorem 2 (BSP and LogP parameters). *For the smallest values of the BSP parameters g^* and l^* (resp. LogP parameters $\max^*(O,G)$ and L^*) achievable on any given topology, it holds that*

$$g^* = \Theta(\max(O^*, G^*)) \quad \text{and} \quad l^* = \Theta(L^*),$$

provided that both machines run oblivious programs.

So far we only considered the worst case behavior of the two models. They are equivalent for oblivious programs in the sense that bi-simulations are possible with constant delay. We now consider the time for implementing a barrier synchronization and a packed router on topology networks.

Theorem 3. *Let d be the diameter of a point-to-point network with P processors. Let dg be the degree of the processors of the network. Each processor may route up to dg messages in a single time step but it receives and sends only one message at each time step*[2]*. On any topology, the time l^* of its BSP model is*

$$l^* = \Omega(\max(d(P), \log P)).$$

Proof. A lower bound for synchronization is broadcasting. Hence, $d(P)$ is obviously a lower bound for l^*. Assume an optimal broadcast tree [7] could be embedded optimally on the given topology. Its depth for P processors is $\Theta(\log P)$, a message could be broadcasted in time $\Theta(\log P)$. Hence, synchronization takes at least time $\Omega(\log P)$.

Remark 1. Actually $\Omega(P^{1/3})$ is a physical lower bound for l^* and L^* under the assumption that the processors must be layouted (in 3-dimensions) and signals have a run duration. Then the minimal average distance is $\Omega(P^{1/3})$. Due to the small constant factors of this bound, we may abstract from the layout and model signal delay by discrete hops from one processor to its neighbors.

For many packet routing problems (specific h-relations) and topologies, the latency is considerable smaller than l^* which could lead to a speed up of the LogP programs compared to oblivious BSP programs. This hypothesis is addressed by the next section.

4 Subclasses of Oblivious Programs

In general, there is a higher effort in designing LogP programs instead of BSP programs. E.g., the proof that the BSP capacity constraints are guaranteed reduces to simply counting the number of messages sent in a superstep. Due to the asynchronous execution model, the same guarantee is not easy to prove for the LogP machine. BSP programs cannot deadlock, LogP programs can. Hence, for all considered classes, we will have to evaluate whether:

- the additional effort in programming directly a LogP program pays out in efficiency, or
- compilation of BSP programs to LogP programs speeds up the program's execution, or
- such a compilation cannot guarantee a speedup.

For our practical runtime comparisons, we determine the parameters of the two machine models for the IBM RS6000/SP with 16 processors[3].

[2] This behavior lead to the LogP model where only receiving and sending a message takes processor time while routing is done by the communication network.

[3] The IBM RS6000/SP is a IBM SP, but uses other processors. Therefore, we compiled the BSP tools without processor specific optimizations.

Table 1. LogP vs. BSP parameters (IBM RS6000/SP), message size < 16 Bytes.

BSP	LogP
$l = 502\mu s$	$L = 17.1\mu s$
—	$O = 9.0\mu s$
$g = 30.1\mu s$	$G = 9.8\mu s$

4.1 The Parameters

We use the native message passing library for the LogP model and the Oxford BSP tools[4] for the BSP model. The results are shown in Table 1. For all measurements in this and the following sections we used compiler option -O3 -qstrict and for the BSP library option -flibrary-level 2.

4.2 MPMD-Programs

If an efficient parallel solution for a problem is hard to partition in supersteps, the BSP model is not appropriate. For those programs the LogP model seems preferable. Due to its asynchronous execution model, it avoids unnecessary dependencies of processors by synchronization. However, if the problem is low level, as our example broadcast is, the solution may be hidden in a library. It is not hard to extend the BSP model by a set of such library functions. Their implementation could be tuned using the LogP model.

Optimal Broadcast: A basic algorithm on distributed memory machines is the optimal broadcast of data from one processor to all others, which is used in many applications. For the LogP model, an optimal solution is given by Karp at al. in [7]. Each processor that received the item immediately initiates a repeated sending to processors which have not received the item until there is no such processors. We achieve the optimal BSP broadcast for our machine if the first processor sends messages to all others in one superstep. After synchronization the other processors receive their message. This is optimal for our target machine, since all other implementations would need at least two synchronization steps, which alone cost more than the algorithm given above. The measured runtime of broadcast for LogP and BSP on our machine can be seen in Table 2.

Remark 2. In the measurements for the BSP parameters, we used taged messages where the tag identifies the sender. For an optimal broadcast, this identification of the sender is not necessary. This explains the difference between estimation and measurements. In contrast to the LogP model, it is not possible on the IBM RS6000/SP to receive a message and send a it immediately without a gap. The LogP estimation ignore this property and are therefore too small.

However, even if we assume $\max(O, G) = g$, $L = l$ then the optimal LogP broadcast is a lower bound for the optimal BSP broadcast. The latter requires

[4] see http://www.BSP-worldwide.org

Table 2. Predictions vs. runtimes of broadcast, wave simulation, and FFT.

	BSP		LogP	
	measurement	estimation	measurement	estimation
broadcast	822 μs	980 μs	101 μs	99.6 μs
simulation (n=1,000)	6.84 s	6.35 s	1.50 s	1.56 s
simulation (n=100,000)	20.6 s	19.4 s	14.9 s	14.24 s
FFT (n=1024)	3.16 ms	3.51 ms	2.65 ms	2.67 ms
FFT (n=16384)	34.8 ms	35.6 ms	39.3 ms	35.2 ms

global synchronization barriers between subsequent send and receive operations. These synchronizations lead to delays on other processors if the LogP broadcast tree is not balanced by chance. For each send and receive pair *all* processors have to be synchronized instead of two.

4.3 SPMD-Programs with Sparse Dependencies

Assume that a BSP process sends at most h messages to the subsequent supersteps. The dependencies in the BSP program are *sparse* for M if the for LogP and BSP parameters for M it holds that

$$l + 2h \cdot g > 2O + (h-1)\max(O,G) + \max(L,(h-1)\max(O,G)) \qquad (1)$$

If a BSP programs communicates at most an h-relation from one superstep to the next, these communication costs are bounded by the right hand side of inequation (1) in the compiled LogP program (due to Theorem 1). Then we expect a speed up if the BSP program is compiled instead of directly executed.

However, if the problem size n is large compared to P, computation costs could dominate communication costs. Hence, if the problem scales arbitrarily, the speed-up gained by compilation could approach zero for increasing n.

Wave Simulation: For simulating a one-dimensional wave, a new value for every simulated point is recalculated in every time step according to the current value of this point(y_0), its two neighbors (y_{-1},y_{+1}), and the value of this point one time step before (y'_0). This update is performed by the function

$$\Phi(y_o, y_{-1}, y_{+1}, y'_o) \;=\; 2 \cdot y_0 - y'_o + \Delta_t^2/\Delta_y \cdot 2 \cdot (y_{+1} - 2 * y_0 + y_{-1}).$$

Since recalculation of one node needs the values of direct neighbors only, it is optimal to distribute the data balanced and block by block on the processors. Figure 1 sketches two successive computational steps and the required communication. The two programs for the LogP and BSP model only differ in the communication phase of each computation step. In both models, the processes communicate with their neighbors, i.e, the communication phase must route a 2-relation. The BSP additionally performs a synchronization of all processors. For our target machine, the data dependencies are sparse i.e., each communications phase is faster on the LogP machine than on the BSP, cf Table 2.

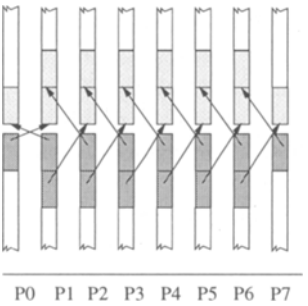

Fig. 1. A communication phase of the Wave Simulation.

4.4 SPMD-Programs with Dense Dependencies

We call data dependencies of a BSP program *dense* for a target machine if they are not sparse. Obviously for those programs compilation does not pay as the following example show.

Fast Fourier Transform: We consider an one-dimensional parallel FFT and assume the input vector v to be of size $n = 2^k, k \in \mathbb{N}$. Furthermore, let $n \geq 2 \times P$. Then the following algorithm requires only one communication phase: v is initially distribute block-by-block and we may perform the first $\log(n/P)$ operations locally. Then v is redistributed in a cyclic way, which requires an all-to-all communication. The remaining operations can be executed also locally. The computation is balanced and a barrier synchronization is done implicitly on the LogP machine with the communication. Hence, we expect no noteworthy differences in the runtimes. The measurements in Table 2 confirm this hypothesis.

Remark 3. For the redistribution of data from block-wise to cyclic, we used a LogP–library routine, which gathers data by memcopy calls with variable data length. For the BSP, such a routine does not exist and was therefore implemented by hand. This implementation allowed the compiler to use more efficient copying routines and explains the difference of the LogP runtime to its estimation and to the BSP runtime.

5 Conclusions

We show how to compile of oblivious BSP algorithms to LogP machines. This approach improves the best known simulation delay of BSP programs on the LogP machine [2] by a factor of $O(\log(P))$. It turns out, that the both models are asymptotically equivalent for oblivious programs. We identified a subclass of oblivious programs that are potentially more efficient if directly designed for the LogP machine. Due to a more comfortable programming, other programs are preferably designed for the BSP machine. However, among those we could identify another subclass that are more efficient if compiled to the LogP machine

instead of executed directly. Others are not. Our measurements determine the parameters for a LogP and a BSP abstraction of a IBM RS6000/SP with 16 processors. They compare broadcast, wave simulation, and FFT programs as representative of the described subclasses of oblivious programs. Predictions and measurements of the programs in both models confirm our observations.

Further work could combine the best parts of both worlds. We could use the same classification to identify parts of BSP programs that are executed directly, namely the non-oblivious parts and those which do not profit from compilation, and compile the other parts. Low level programs from the first class, like the broadcast, could be designed and tuned for the LogP machine and inserted to BSP programs as black boxes.

References

1. Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, 1995.
2. Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. Bsp vs logp. In *SPAA'96: 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32. ACM, acm press, June 1996.
3. R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *SIAM Journal on Computing*, 11(3):540–546, 1982.
4. 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 *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 93)*, pages 235–261, 1993.
5. Jörn Eisenbiegler, Welf Löwe, and Andreas Wehrenpfennig. On the optimization by redundancy using an extended LogP model. In *International Conference on Advances in Parallel and Distributed Computing (APDC'97)*, pages 149–155. IEEE Computer Society Press, March 1997.
6. Jonathan M. D. Hill, Paul I. Crumpton, and David A. Burgess. Theory, practice, and a tool for BSP performance prediction. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 697–705. Springer, August 1996.
7. R.M. Karp, A. Sahay, E.E. Santos, and K.E. Schauser. Optimal broadcast and summation in the logp model. *ACM-Symposium on Parallel Algorithms and Architectures*, 1993.
8. W. F. McColl. Scalable computing. In Jan van Leeuwen, editor, *Computer Science Today*, number 1000 in Lecture Notes in Computer Science, pages 46–61. Springer, 1995.
9. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.