Performance Relevant Issues for Parallel Computation Models

Ben Juurlink
Electrical Engineering Department
Delft University of Technology
Delft, The Netherlands

Ingo Rieping*
Computer Science Department
Paderborn University
Paderborn, Germany

Abstract

It is argued that the most important property of a suitable model of parallel computation is that it correctly predicts which algorithms is the most efficient in practice, and it is shown that models that do not reward block transfers do not always fulfill this requirement. We also discuss if the algorithm developer should have to specify the order in which messages are sent, and how these issues are dealt with differently in two BSP libraries.

1 Introduction

It has been argued by many researchers that one of the reasons why parallel computing has not been very successful is the absence of a standard parallel computation model (PCM) suitable for designing and analyzing parallel algorithms (see, e.g., [20, 19, 16, 7]). Such a model must fulfill several crucial requirements. First, it should be *simple* enough to facilitate the design and analysis of parallel algorithms. Second, it should be *accurate* so that efficient algorithms developed in the framework of the model translate to efficient programs. Third, it should lead to *portable* parallel algorithms, i.e., algorithms that run efficiently on a wide variety of parallel architectures.

The most important property for a suitable model is that its runtime predictions are accurate. However, we do not believe that it is possible to formulate a PCM whose predictions are always within, say, 10% of the actual execution times. A model that achieves this is likely to be too complex to serve as the basis for parallel algorithm design. Nevertheless, the model should be able to correctly predict which algorithm is the most efficient in practice. In other words, the most crucial property of a suitable model is that when it predicts that algorithm A is faster than algorithm B, then A should run faster than B when implemented on an actual platform. This property enables algorithm developers to pick the fastest algorithm from several alternatives without having to implement them all.

Many models have been proposed (see, e.g., [15] for an overview) but hardly ever empirical evidence is provided that demonstrates their predictive capabilities. In this paper we advocate a systematic study of all performance relevant issues for PCMs. We concentrate on two issues. First, should a suitable model reward sending large messages? Second, should the model capture endpoint contention, i.e, the phenomenon that when several processors send to the same processor at the same time, stalls occur. We also discuss how these issues are dealt with differently in two communication libraries based on the BSP model [20, 16]: BSPlib [10] and the Paderborn University BSP (PUB) library [6], and shortly discuss how different models lead to different algorithmic techniques.

2 Block Transfers

It is well-known that on many parallel computers, there is a high startup cost associated with each message transmission. For example, in [9] it is shown that, typically, the messages must be at least 1KB to achieve only half the maximum bandwidth and in many cases even much longer. This is *not*, however, sufficient evidence that a suitable PCM should reward sending large messages. Some models, in particular BSP [20], allow messages destined for the same processor to be combined in a single message at runtime.

*Supported in part by DFG-SFB 376 "Massive Parallelität" and EU ESPRIT Long Term Research Project 20244 (ALCOM-IT).

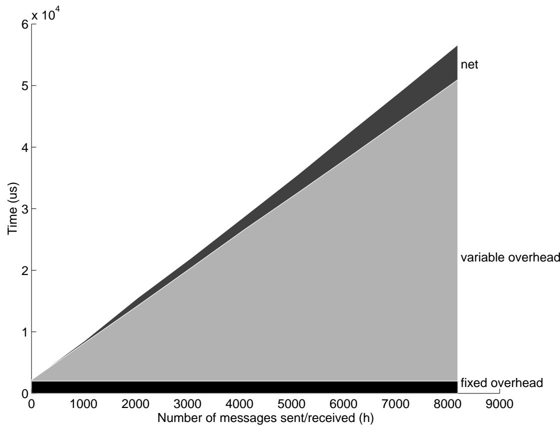

Figure 1. Breakdown of the time required for routing h-relations on the Intel Paragon.

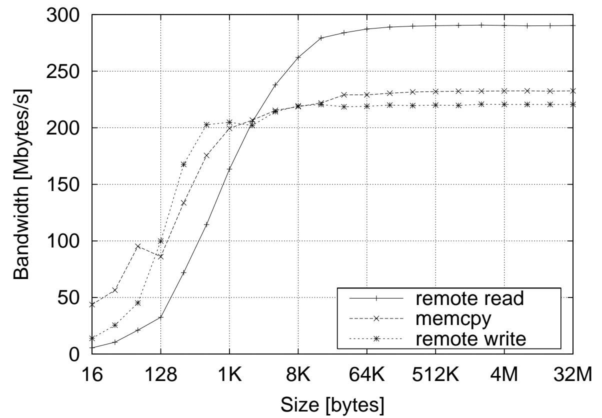

Figure 2. Comparison of local and remote bandwidth of the Cray T3E.

Briefly, computations in the BSP model are organized in supersteps, separated by barrier synchronizations. In each superstep, a processor can perform local operations, send some messages and (implicitly) receive some messages. Two parameters are used to model communication cost: (1) the latency/synchronization cost $\mathbf{L}$, and (2) the bandwidth reciprocal $\mathbf{g}$. It is assumed that an arbitrary *h-relation* (a communication pattern in which each processor sends/receives at most h messages) followed by a barrier synchronization takes $\mathbf{g} \cdot h + \mathbf{L}$ time. All messages have a fixed, *short* size (essentially the word size). An extension of BSP that rewards block transfers is BSP* [3, 4]. In this model, time $\mathbf{g}^* \cdot h + \mathbf{s} \cdot m + \mathbf{L}^*$ is charged for the communication part of a superstep, where h is the maximum amount of data in or out off each processor, $\mathbf{s}$ is the startup cost of message transmissions, and m is the maximum number of messages sent/received by any processor[1].

The semantics of the BSP model allow all communication to be postponed until the end of the superstep so that messages destined for the same processor can be sent in a single message. This approach is taken in BSPlib [10]; a communication library based on the BSP model. In [18] it is argued that this reduces the importance of sending large messages. This technique has several drawbacks, however.

First, sending large messages is still beneficial because a larger part can be used for user data.

[1]The BSP* cost model used here differs slightly from [3, 4].

Second, on high-bandwidth systems where communication is (nearly) as fast as local memory accesses, buffering messages contributes significantly to the communication overhead. To illustrate this, Figure 1 (taken from [11]) divides the time needed for routing h-relations on the Intel Paragon into three component: (1) fixed overhead, which includes the startup costs of message transmissions, (2) variable overhead, which is the time needed for buffering messages, and (3) the time needed for sending the messages through the network. It can be seen that the actual communication time is only a small fraction of the total time. As another example, Figure 2 compares the local bandwidth of the Cray T3E with its remote bandwidth. This figure shows that writing to a remote memory location is as fast as writing to a local memory location. In fact, reading from a remote memory location is actually faster than a local `memcpy`!

Third, this technique requires a lot of memory for buffering. If one processor has 1 MB of data to send to every other processor and does so by simply sending each processor a copy (admittedly not the best implementation under any reasonable model), it needs p times as much memory. In [10] it is stated that this is the reason why BSPlib also has unbuffered variants of its remote memory access primitives.

However, the main reason for rewarding block transfers is that a model that does not sometimes provides wrong incentives to the algorithm developer. Consider the problem of broadcasting $n \geq p$ data items from one processor to all others. The BSP model

suggests the following 2-phase algorithm [12]:

1. The source processor distributes its items among all other processors.
2. Every processor sends its items to all other processors.

The BSP and BSP* costs of this algorithm are approximately

$$T_{\text{BSP}} = 2 \cdot (\mathbf{g} \cdot n + \mathbf{L})$$
$$T_{\text{BSP*}} = 2 \cdot (\mathbf{g}^* \cdot n + \mathbf{s} \cdot p + \mathbf{L}^*).$$

Because the root processor has to send out n messages anyhow, the algorithm is within a factor of 2 from optimal under the BSP cost model.

The BSP* model might encourage a tree-like algorithm in which the root processor first sends its items to one other processor, then both send the message to two other processors, etc. This algorithm incurs only $\lceil \log p \rceil$ startups, and its BSP and BSP* costs are approximately

$$T_{\text{BSP}} = \log p \cdot (\mathbf{g} \cdot n + \mathbf{L})$$
$$T_{\text{BSP*}} = \log p \cdot (\mathbf{g}^* \cdot n + \mathbf{s} + \mathbf{L}^*).$$

So, BSP model clearly predicts that the 2-phase algorithm is superior in all cases, whereas BSP* predicts that the tree-like algorithm is preferable when the startup costs dominate.

Figure 3 compares the performance of both broadcast algorithms on a 64-processor Intel Paragon. It can be seen that for a wide range of input sizes the tree-like algorithm is actually faster than the 2-phase algorithm, in contradiction to what the BSP model predicts. For one very large input size ($n = 256\text{KB}$), the 2-phase algorithm is indeed more efficient, but BSP* also predicts this. It needs to be mentioned that the implementations of the algorithms do not synchronize using barrier synchronization. A barrier synchronization incurs many startups and this would slow down both algorithms, but especially the tree-like algorithm (since it consists of more supersteps) and thus it would bias the 2-phase algorithm.

So, even for this simple problem, the BSP model[2] is incapable of predicting the relative performance of two

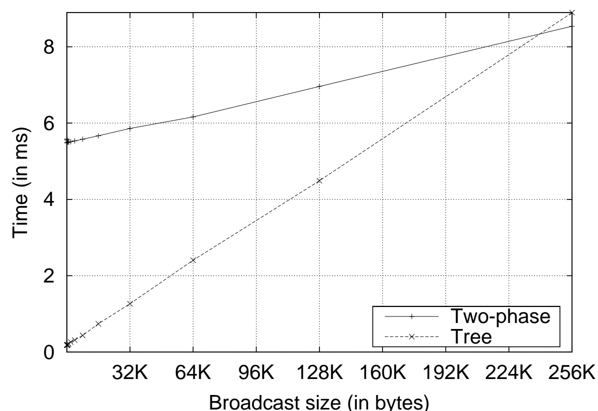

Figure 3. Comparing the performance of two broadcast algorithms on the Paragon.

algorithms. This seems to be strong evidence that parallel computation models should reward sending large messages, at least a suitable model of this architecture and also of others that have a significant startup cost. The main argument is that sometimes higher performance can be achieved by *increasing* the communication volume, which clearly contravenes the incentives provided by the BSP model.

The Paderborn University BSP (PUB) library [6] provides primitives for collective communication operations like broadcast, parallel prefix, etc. for the following reasons. First, they are used frequently in parallel applications and often determine the overall efficiency. For example, the sole communication routine in the Linpack benchmark performing LU decomposition is a broadcast. Second, as shown above, high-level algorithmic models like BSP are unable to predict which algorithm is the most efficient in practice.

Algorithms employing "blockwise communication" for fairly regular problems like matrix multiplication, FFT, and sorting are described in, e.g., [2]. It is an interesting open question if algorithmic techniques to utilize block transfers can also be developed for many irregular problems. Some positive results have been reported in, e.g., [3, 4] and [17].

In [3, 4], the (m, n)-*multisearch* problem is considered: given a search data structure consisting of n keys and m search keys or *queries*, determine for each query if it is contained in the search structure and if not, where it should be inserted. This problem is similar to the familiar binary search problem but with m

[2] We have taken the BSP model as an example, but this remark applies to all models that do not reward block transfers

queries instead of one. Without going into detail, models that do not reward block transfers urge the algorithm developer to distribute the successors of a node mapped to a certain processor as randomly as possible over all processors to avoid load imbalance. This means, however, that when m' queries are located in a certain processor at a certain phase, the average message length will be m'/p. The work in [3, 4] has shown that when the nodes are distributed over $\Theta(\log p)$ processors, load imbalance is still avoided, and, moreover, the average message length will be much longer, namely $\Theta(m'/\log p)$.

In [17], one of the most fundamental graph problems is considered: *list ranking*. The experimental results provided also show that sometimes higher performance can be achieved by increasing the communication volume if it reduces the number of startups.

3 Endpoint Contention

In the BSP model, the algorithm developer does not specify the *order* in which messages are sent. Consider, for example, the total exchange communication pattern in which each processor has a message of size m that needs to be sent to all other processors. A naive implementation might let each processor first send to processor 0, then to processor 1, etc. This causes first contention at the input communication port of processor 0, then at the input port of processor 1, etc. We call this *endpoint contention*, to distinguish it from contention within the network. Alternatively, one can use a staggered communication schedule in which processor i first communicates with processor $(i+1) \bmod p$, then with $(i+2) \bmod p$, and so on.

The issue of endpoint contention was raised by the LogP model [7]. This model has a capacity constraint such that at most $\mathbf{L}/\mathbf{g}$ messages can be in transit from or to any processor at any time, where $\mathbf{L}$ is the latency parameter of the LogP model and $\mathbf{g}$ is the gap.

In [18] it is argued that the naive communication schedule can take p times as long as the staggered one. However, we have performed experiments on several platforms that showed that the naive schedule "only" takes up to twice as long. The data provided in [18, Table 3] also confirms this. To explain this, consider Figure 4, which shows a Gantt-like diagram that illustrates how message transmissions are scheduled in the naive implementation. There is a box labeled P_j in the row labeled P_i at time t if processor P_j's message for P_i is being received by P_i at time t. If several processors send a message to the same processor at the same time, we assumed that priority is given to the processor with the lowest ID, but by symmetry, this is irrelevant for the discussion. It can be seen that once a processor has succeeded in sending its message to processor P_0, it will not stall again because some kind of pipelining effect occurs. The last message will be received at time $(2 \cdot p - 1) \cdot \mathbf{g} \cdot m$, which is indeed almost twice the time taken by the staggered schedule: $p \cdot \mathbf{g} \cdot m$.

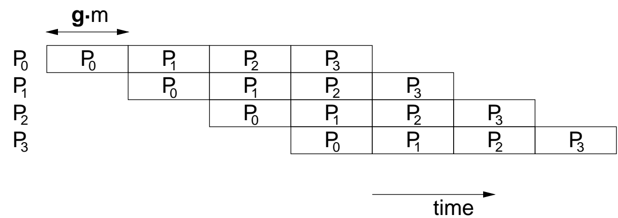

Figure 4. Gantt-like chart illustrating why the naive schedule takes twice as long as the staggered one.

In BSPlib implementations, a Latin square is used in an attempt to avoid endpoint contention [18]. Since communication occurs at the end of the superstep (to reduce the importance of sending large messages), the order in which the messages are sent can also be changed. In this approach, processor P_i sends its messages in the order specified by the ith row of the Latin square. However, this may work for *perfectly balanced* communication patterns like the total exchange, it will **not** avoid endpoint contention in all cases.

To support this claim, we have written a simulator based on the following assumptions, which is the implicitly assumed model in all discussions concerning endpoint contention:

1. There are no collisions within the network. Only endpoint contention can deteriorate the performance of the communication network.

2. Each processor has one input communication port and one output port. If a processor sends a message to another processor, its output port is busy until the message has been accepted. Furthermore, once a processor starts accepting a message from another processor, its input channel is engaged until the entire message has been received.

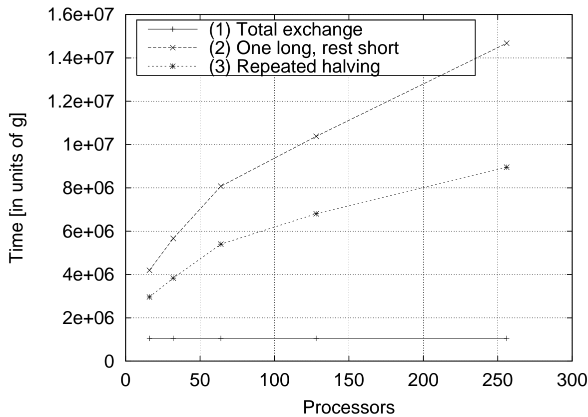

Figure 5. Time needed for routing different communication patterns using a Latin square.

3. If several messages are sent to the same processor at the same time, it will receive one of them at random. The other messages are stalled.
4. The order in which messages are sent is determined by a Latin square generated at random.

We could have used real machines but they do not fulfill the above assumptions. Furthermore, current platforms are often relatively small.

Figure 5 shows the results obtained by simulating three communication patterns: (1) total exchange, (2) "one long, rest short" – each processor sends a very long message (of $h - P + 1$ words) to a partner processor and a very short message of one word to all other processors, and (3) "repeated halving" – processor i sends a message of size $h/2$ to processor $(i+1) \bmod p$, a message of size $h/4$ to $(i+2) \bmod p$, and so on. The figure shows the total routing time for an $h = 2^{20}$-relation as a function of the number of processors. It can be seen that time is very dependent on the communication pattern, even though a Latin square is used to schedule the communication. Moreover, the routing time for communication patterns (2) and (3) grows with the number of processors p.

In fact, in a theoretical paper, Adler et al. [1] showed that no on-line (meaning without knowledge of the communication pattern to be routed) can route an arbitrary h-relation in $h + o(h)$ time.

So, using a Latin square does not guarantee contention freedom, and for the total exchange the difference between a good schedule and a bad one is at most a factor of two. The PUB library [6], therefore, does not change the order in which messages are sent. However, it is an issue (we believe) the programmer, not so much the algorithm developer, should be aware of. In most cases it can be avoided relatively easy by using, e.g., a staggered schedule, as long as the actual schedule is not hidden. In many other cases, the communication pattern is "sufficiently random" so that endpoint contention plays only a minor role. It is remarkable that the developers of the LogP model decided in [8] to ignore endpoint contention and nevertheless obtained accurate predictions.

4 Other Issues

In this paper we mainly focus on two issues: block transfers and endpoint contention. In this section, we touch on some other issues.

4.1 Synchronization

On many contemporary architectures barrier synchronizations are expensive, especially if it is implemented by performing a total exchange at the end of each superstep, since then it includes the cost of $p - 1$ startups. PUB, therefore, provides another form of synchronization besides barrier synchronization. If it is known how many messages a processor is due to receive, it can synchronize by calling the primitive `bsp_oblsync(bsp,n)`, which simply blocks the calling processor until `n` messages have been received. This primitive incurs no communication cost and is, therefore, faster than a barrier synchronization. We do not agree with the view that this form of synchronization should be excluded from a BSP library simply because it was not present in the original BSP model.

4.2 Unbalanced Communication

The BSP model (as well as others) assumes that the communication time is independent of the network load. It charges the same amount of time for a *full* h-relation in which all processors send and receive exactly h messages as for, e.g., a communication pattern in which only one processor sends h messages to another processor. In other words, the communication *volume* is not a factor in the BSP cost model.

The E-BSP model [13] extends the basic BSP model to deal with unbalanced communication, i.e., communication patterns in which the processors send and/or

receive different amounts of data. The cost functions used in this work are generally nonlinear functions that depend on the network topology. In [14] we, therefore, employed a simplified version of E-BSP. Let a (V, h)-relation be a communication pattern in which each processor sends and receives at most h messages, and in which the total number of messages being routed does not exceed V (the communication volume). In the E-BSP model, every communication pattern is viewed as an instance of a (V, h)-relation with cost $\max\{\mathbf{g} \cdot V/p, \mathbf{g}' \cdot h\} + \mathbf{L}$. Note that the parameter $\mathbf{g}'$ captures node-to-network bandwidth, whereas $\mathbf{g}$ captures intranetwork bandwidth. Also note that this cost model essentially differentiates between communication patterns that are insensitive to the bisection bandwidth and those that are not.

To get an idea of the importance of unbalanced communication, Figure 6 shows the bandwidth obtained on a 64-processor Cray T3E for different communication patterns. In every experiment each processor sends at most $p - 1$ messages of length m. The figure depicts the obtained bandwidth as a function of the message length. The communication patterns are:

- *One-to-one*: each processor i sends $p - 1$ messages to a partner processor ($(i + p/2) \bmod p$).
- *All-to-all*: staggered total exchange.
- *One-to-all*: processor 0 sends to all processors.
- *All-to-one*: all processors send to processor 0.
- *All-to-one-to-all*: *all-to-one* and *one-to-all* simultaneously.

It can be seen that the performance is very dependent on the actual communication pattern. However, this should only be partially attributed to bisection bandwidth limitations. In *one-to-all* and *all-to-one* the maximum amount of data sent *and* received by any processor is $(p - 1) \cdot m$, whereas in the other communication patterns it is $2 \cdot (p - 1) \cdot m$. This is the main reason why *one-to-all* and *all-to-one* attain higher bandwidths. So, on this platform it is better to use a one-port model, where processors cannot send and receive simultaneously. Moreover, when comparing *one-to-one*, *all-to-all*, and *all-to-one-to-all*, one can observe that *all-to-one-to-all* performs comparable to *one-to-one*, even though it is the communication pattern with the smallest volume (the volume for *all-to-one-to-all* is $V = 2 \cdot (p - 1) \cdot m$, for the other two it

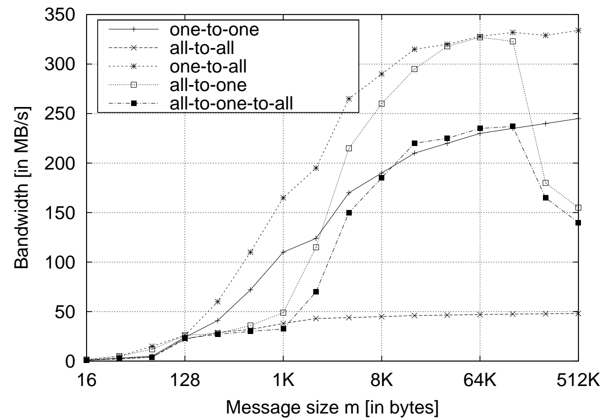

Figure 6. Time needed for routing different communication patterns on the T3E.

is $V = p \cdot (p - 1) \cdot m$). However, further experimentation revealed that communication patterns in which processor i communicates with $(i + p/2) \bmod p$ are remarkably cheap on the T3E. If the processors communicate with different partners, performance drops tremendously, sometimes my more than a factor of 5.

So, for systems with a moderate number of processors, unbalanced communication is of minor importance, even though it cannot be completely ignored. But it is clear (to us) that when moving to a large number of processors, the bisection bandwidth will affect the performance of different communication schemes. Recent theoretical work [5] has shown that models of mesh-connected parallel computers that allow network proximity to be exploited deal efficiently with unbalanced communication. So, for these architectures, unbalanced communication may be ignored, at least from a theoretical standpoint.

4.3 Network Proximity

Most models give up network proximity, mainly because models that allow network proximity to be exploited are more complex and less general than models that do not. Furthermore, "wormhole" routing techniques have made message communication time in lightly loaded networks rather insensitive to the distance between communicating processors. However, it is still the case that local communication patterns can be routed faster than global ones, since less data has to cross the bisection. Even so, for current systems with a moderate number of processors, it seems reasonable

to give up network proximity, although experimental results should be provided to validate this claim.

5 Conclusions

In this paper we have attempted to start a systematic study of all performance relevant issues for parallel computation models. We have concentrated mainly on two issues: (1) should a suitable model reward block transfers, and (2) should the model capture endpoint contention? Both questions have been the subject of much debate, but no definite answers have been given.

To the first question, our tentative answer is yes. The most important argument is that sometimes higher performance can be achieved by increasing the communication volume if this reduces the number of message transmissions. Models that do not reward sending large messages fail to predict this. It may be the case that future parallel systems will have much smaller startup costs, but today this conclusions seems premature. It is remarkable that the data provided in [9] shows that especially high-bandwidth systems suffer from a (relatively) high startup cost.

To the second question, our (again tentative) answer is no. In the total exchange, the difference between a good schedule and a bad schedule is at most a factor of two. Furthermore, it was shown that the Latin square technique does not guarantee a contention free schedule. In those cases where contention is a significant problem, the programmer should be given the opportunity to avoid it.

References

[1] M. Adler, J. W. Byers, and R. M. Karp. Scheduling Parallel Communication: The h-relation Problem. In *Proc. of Mathematical Foundations of Computer Science*, 1995.

[2] A. Aggarwal, A. Chandra, and M. Snir. On Communication Latency in PRAM Computations. In *Proc. Symp. on Parallel Algorithms and Architectures*, 1989.

[3] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly Efficient Parallel Algorithms: 1-Optimal Multisearch for an Extension of the BSP Model. *Theoretical Computer Science*, 203, 1998.

[4] A. Bäumker and F. Meyer auf der Heide. Communication Efficient Parallel Searching. In *Proc. IRREGULAR*. Springer LNCS 1253, 1997.

[5] G. Bilardi, A. Pietracaprina, and G. Pucci. A Quantitative Measure of Portability with Application to Bandwidth-Latency Models for Parallel Computing. In *Proc. Euro-Par*, 1999. LNCS 1685.

[6] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library – Design, Implementation and Performance. In *Proc. IPPS/SPDP*, 1999.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Comm. of the ACM*, 39(11):78–85, 1996.

[8] D. Culler, R. Martin, A. Dusseau, and K. Schauser. Fast Parallel Sorting Under LogP: Experience with the CM-5. *IEEE Trans. on Parallel and Distributes Systems*, 7, 1996.

[9] J. J. Dongarra and T. Dunigan. Message-Passing Performance of Various Computers. *Concurrency: Practice and Experience*, 9(10), 1997.

[10] J. Hill, W. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24(14), 1998.

[11] B. Juurlink. Experimental Validation of Parallel Computation Models on the Intel Paragon. In *Proc. IPPS/SPDP*, 1998.

[12] B. Juurlink and H. Wijshoff. Communication Primitives for BSP Computers. *Information Processing Letters*, 58(6), 1996.

[13] B. Juurlink and H. Wijshoff. The E-BSP Model: Incorporating Unbalanced Communication and General Locality into the BSP Model. In *Proc. Euro-Par*. Springer LNCS 1124, 1996.

[14] B. Juurlink and H. Wijshoff. A Quantitative Comparison of Parallel Computation Models. *ACM Trans. on Computer Systems*, 16(3), 1998.

[15] B. Maggs, L. Matheson, and R. Tarjan. Models of Parallel Computation: A Survey and Synthesis. In *Proc. Hawaii Int. Conf. on System Sciences*, 1995.

[16] W. McColl. Universal Computing. In *Proc. Euro-Par*. Springer LNCS 1123, 1996.

[17] J. Sibeyn. Better Trade-offs for Parallel List Ranking. In *Proc. Symp. on Parallel Algorithms and Architectures*, 1997.

[18] D. Skillicorn, J. Hill, and W. McColl. Questions and Answers About BSP. *Jnl. of Scientific Programming*, 6(3), 1997.

[19] L. Snyder. Experimental Validation of Models of Parallel Computation. In J. van Leeuwen, editor, *Computer Science Today*. Springer LNCS 1000, 1995.

[20] L. Valiant. A Bridging Model for Parallel Computation. *Comm. of the ACM*, 33(8), 1990.